Université Bretagne Sud

Master 1 Ingénierie de Systèmes Complexes

Spécialité Cybersécurité des Systèmes Embarqués

Promotion 2019-2020

Étude du protocole BLE

Stage Master 1

Gidon Rémi

Avril/Juin 2020

Sommaire

Tableaux

Figures

Dû aux conditions exceptionnelles imposées par la pandémie du COVID-19, j'ai réalisé un sujet fourni par le laboratoire d'informatique LAB-STICC affilié à l'université dans le cadre du stage de première année de master CSSE à l'UBS de Lorient.
Initialement prévu sur l'étude les protocoles de communication domotique (`ZigBee`, `Z-Wave`, `Thread` ...) à l'aide d'une carte `HackRF`, j'ai dû m'adapter avec le confinement et ait bifurqué sur du matériel et un protocle accessible: le *Bluetooth Low Energy*. Son intégration dans nombre d'appareils de bureautique en ont fait un choix pour la communication avec les systèmes embarqués constituant les objets intelligents. L'étude du protocole est facilitée par cette popularité, disposant de matériel dédié à moindre coût sur le marché ainsi qu'une floppée d'outils logiciels et d'audits de sécurité révélant et expliquant les vulnérabilités du protocole.

Dans le cadre du master CSSE nous étudions l'internet des objets (IoT) et leurs aspects sécurité. Le protocole réseau sans fil *Bluetooth Low Energy* (`BLE`) permet une consommation réduite pour les objets fonctionnant sur batterie, visant notamment les objets connectés. Aujourd'hui integré dans la plupart des appareils de bureautique, il est rapidement devenu populaire dans l'internet des objets.

La première itération du BLE ne répond plus aux exigences de sécurité contemporaine et même si le protocole à su évoluer depuis pour répondre à ces besoins, beaucoup d'appareils utilisent encore la version originale n'intégrant pas ces mécanismes.
Ce sont pour la plupart des appareils conçus pour fonctionner sur batterie et communiquer en point à point. On va retrouver les capteurs corporels pour santé ou fitness mais également des mécanismes plus sensibles tels des cadenas ou serrures. Les communications (incluant parfois des données personnelles) peuvent êtres interceptées voir modifiées pour permettre des actions aux dépends de l'utilisateur.

1 Objets connectés

Avec l'explosion de l'internet de objets au cours de ces dernières années, toute une floppée d'objet du quotidien ont été augmentés pour permettre la communication avec d'autres systèmes informatique dont nos *smartphones* ou encore des serveurs distants (via notre `Wi-Fi`). Ces objets dits intelligents étendent leur équivalent mécanique en intégrant des composants éléctroniques, permettant notamment le contrôle à distance.
Face à l'engouement du public, les constructeurs s'efforcent de proposer des objets toujours plus *intelligents* et connectés, souvent au détriment de la sécurité. Ces améliorations engendrent une augmentation de la surface d'attaque: les objets connectés sont confrontés aux mêmes challenges que ceux des systèmes informatiques traditionnels en plus de leur fonction primaire.

1.1 Architecture

Les objets connectés ont commencés par proposer des communications avec nos *smartphones*, notre routeur `Wi-Fi` ou notre ordinateur. Celles-ci permettent d'utiliser ces objets comme télécommande de contrôle (via une application dédiée la plupart du temps) ainsi que de communiquer aux services distants du constructeur en s'appuyant ces fonctionnalités (`Wi-Fi`, données mobiles).
Ces architectures point à point connectent un appareil directement à un contrôleur (*smartphone*, *PC*) duquel il est dépendant pour accéder aux services distants. On retrouve cette architecture dans les appareils autonomes qui réalisent une fonction d'augmentation sur un produit existant (comme les cadenas connectés). Ces appareils peuvent se passer d'un service distant puisqu'ils proposent un modèle d'interactions simple et local avec un seul utilisateur.

Les objets connectés ont rapidement été utilisés pour la mise en place de réseaux de capteurs. Cette utilisation à été largement introduite pour les particuliers avec la domotique. Les interactions avec ces réseaux ont d'abors été locales, au sein du domicile, puis globales pour permettre un contrôle depuis n'importe quel emplacement. Cette tendance pousse les constructeurs à exposer leurs objets connectés au réseau mondial: certains ont optés pour incorporer des puces `Wi-Fi` directement dans leurs objets quant à d'autres ont proposés une solution plus long terme en mettant en place une passerelle (appelée *hub*) gérant les interactions avec le monde extérieur.
L'architecture *hub* est aujourd'hui vastement utilisée dans la domotique. Un appareil dédié est considéré comme *hub* par lequel les capteurs, beaucoup plus simples, communiquent. Des protocoles spécialisés ont fait leur apparition comme `Zigbee`[1], `Z-Wave`[2], `ANT+`[3] ou encore `Thread`[4].

1.2 Protocoles

Comme évoqué précédemment, beaucoup d'objets connectés ont profité des protocoles intégrés dans les appareils utilisés comme contrôleur. Cela englobe le `Wi-Fi`, le `Bluetooth` et dernièrement le `NFC`. De ces trois, le `Bluetooth` à largement pris le dessus car plus adapté avec son standard *Low Energy*. Concu en tant que *WPAN* (réseau sans fil personnel) il permet de communiquer dans un rayon de 10 mètres, suffisant pour les interactions locales. Le `NFC` est assez réçent, il à été conçu pour les interaction proches (une dizaine de centimètres) et intentionnelles comme les paiements. Enfin le `Wi-Fi` aurait dû être le plus populaire, puisque chaque foyer dispose d'une box internet, mais c'est un *WLAN* (réseau sans fil local) permettant une couverture et des débits beaucoup plus grands. Sa

consommation est telle qu'un objet sur batterie ne tient que quelques heures au maximum (voir les ordinateurs portables, disposants pourtant de larges batteries). Il n'est pas adapté pour ce besoin puisqu'il permet de hauts débits avec faible latence pour une consommation élevée la ou les objets connectés utilisent de faibles débits sur des transmission occasionnelles pour économiser leurs batteries.

La démocratisation des objets connectés à permit l'émergence de nouveaux protocoles spécialisés. Même si le `BLE` s'adapte pour répondre aux besoins de ce marché, il n'a pas été conçu pour les objets connectés mais plutôt pour permettre des interactions humaines avec les objets intelligents.
Beaucoup de constructeurs ont developpés leur protocole, le vantant et l'imposant avec leurs produits et architectures propriétaires. Des géants l'ont même intégré à leur écosystème: *Apple Home* utilise *Darwin* (*iOS*, *macOS*) comme contrôleur ainsi que son propre protocole (`HAP`) pour ses objets connectés, Google a mis en place le protocole `Thread`, interopérable avec *Google Home* (*hub*) et *Android* (contrôleur).
Beaucoup de protocoles se battent pour avoir accès à un marché juteux encore instable car en plein développement. Cependant tout les objets connectés ne recherchent pas les améliorations qu'apportent ces protocoles, leur caractère spécialisé demande une intégration particulière dont l'intéroperabilité restreinte est bien à l'encontre de la standardisation et l'omniprésence (dans les appareils de bureatique) offerte par le `BLE`.

2 Bluetooth Low Energy

Le protocole a principalement été designé par Nokia pour répondre au besoin d'un protocole sans fil peu gourmand en énergie permettant la communication avec les périphériques personnels (téléphone portable, montre, casque audio). Nommé *Wibree*, il a été intégré au standard `Bluetooth` sous le nom *Low Energy*.
Le `Bluetooth` ne comprend pas seulement un protocole mais une multitude d'entre eux (`BR`, `EDR`) qui ont en commun de permettre la communication (et l'échange de données) sans fil avec des périphériques personnels. Ils font partit des protocoles *WPAN* et leur distance d'émission varie de quelques mètres jusqu'à 30 mètres.
La spécification `Bluetooth` 4.0, sortie en 2010, intégre le protocole `LE` (*Low Energy*) et permet au `Bluetooth` de toucher le marché des systèmes embarqués fonctionnants sur batterie.

2.1 Différences

Les autres protocoles du `Bluetooth` sont principalement connus et utilisés pour le transfert de contenu multimédia, que ce soit des fichiers entre ordinateurs comme de la musique avec un casque ou encore une voiture. Ils fonctionnement avec une connexion continue et un transfert en mode flux.
Le `BLE`, visant à réduire la consommation d'énergie, n'établie pas de connexion continue. L'appareil reste la plupart du temps en mode veille, pouvant émettre des annonces dans l'attente d'une connexion. Pour chaque requête reçu, une réponse pourra être renvoyée ou une notification mise en place périodiquement.
Les appareils `BLE` et `Bluetooth BR/EDR` ne sont pas compatibles, n'utilisant pas les mêmes technologies/protocoles et répondant à des besoin différents (tbl. 1).

Tableau 1: Cas d'utilisation et protocole Bluetooth adapté

Besoin	Flux données	Transmisson données	Localisation	Réseau capteurs
Appareils	ordinateur, smartphone, casque, enceinte, voiture	accessoires bureautique ou fitness, équipement médical	beacon, IPS, inventaire	automatisation, surveillance, domotique
Topologie	point à point	point à point	diffusion (1 à N)	mesh (N à N)
Technologie	Bluetooth BR/EDR	Bluetooth LE	Bluetooth LE	Bluetooth LE

2.2 Protocole

Pour permettre une interopérabilité maximale entre les appareils `BLE`, le standard défini 4 profils en fonction du rôle de l'appareil: *Peripheral*, *Central*, *Broadcaster*, *Observer*. Chaque appareil se conformant au standard ne doit implémenter qu'un seul de ces rôles à la fois.
Le *broadcaster* ne communique qu'avec des annonces, on ne peut pas s'y connecter. Ce mode est très populaire pour les balises (*beacons*). L'*observer* est sont opposé, il ne fait

qu'écouter les annonces, n'établiera jamais de connexion.
Le *peripheral* et le *central* forment la seconde paire et permettent la mise en place d'une architecture client-serveur. Le *peripheral* joue le rôle du serveur et est dit esclave du *central* qui endosse le rôle du client et maître.
Le *peripheral* transmet des annonces jusqu'à recevoir une connexion d'un *central*, après quoi il arrête de s'annoncer car ne peut être connecté qu'à un *central* à la fois. Le *central* écoute les annonces de *peripheral* pour s'y connecter, puis interroge ses services via les requêtes *ATT/GATT*.

Couche physique

Le `BLE` opère dans la bande ISM 2.4GHz tout comme le `Wi-Fi`. Contrairement aux canaux `Wi-Fi` de 20MHz, le `BLE` découpe le spectre en 40 canaux de de 2MHz (plage de 2400 à 2480MHz).
Le protocole met en place le *saut de fréquence* qui consiste à changer de canal d'émission tout les laps de temps donné pour réduire le risque de bruit sur les fréquences utilisées (la bande ISM 2.4Ghz étant libre d'utilisation).
Sur les 40 canaux que compose le spectre, 3 sont utilisés pour la transmission d'annonce. Ils sont choisit pour ne pas interférer avec les canaux `Wi-Fi` car les deux protocoles sont amenés à coexister (fig. 1).
Les 37 autres canaux sont utilisés pour les connexions. Chaque connexion va utiliser un sous-ensemble des 37 canaux (appelé carte des canaux) pour éviter les interférences avec les autres protocoles et connexions `BLE`. Un seul canal transmet des données à la fois mais tous les canaux de la carte sont utilisés pour le saut de fréquences.

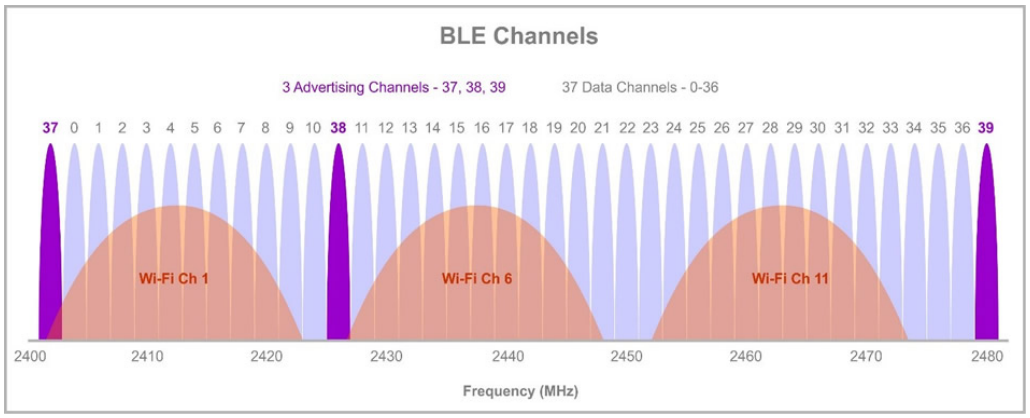

Figure 1: Répartition du spectre BLE en canaux[5]

Couche logique

1. Annonces

Le *peripheral* indique sa présence avec des annonces émises périodiquement. Ces annonces contiennent sont addresse `Bluetooth` (permettant une connexion) et des données qui consituent un profil (appelé *GAP*[6]). Ces données permettent aux *centrals* de savoir si il est capable de réaliser les fonctionnalités recherchées.
La spécification `Bluetooth` définit des profils type pour des applications communes dans les appareils `BLE`[7]. Cela inclus par exemple les capteurs corporels pour le sport, les capteurs médicaux de surveillance (pour les diabètiques notamment), la domotique (termomètres, lampes), etc.

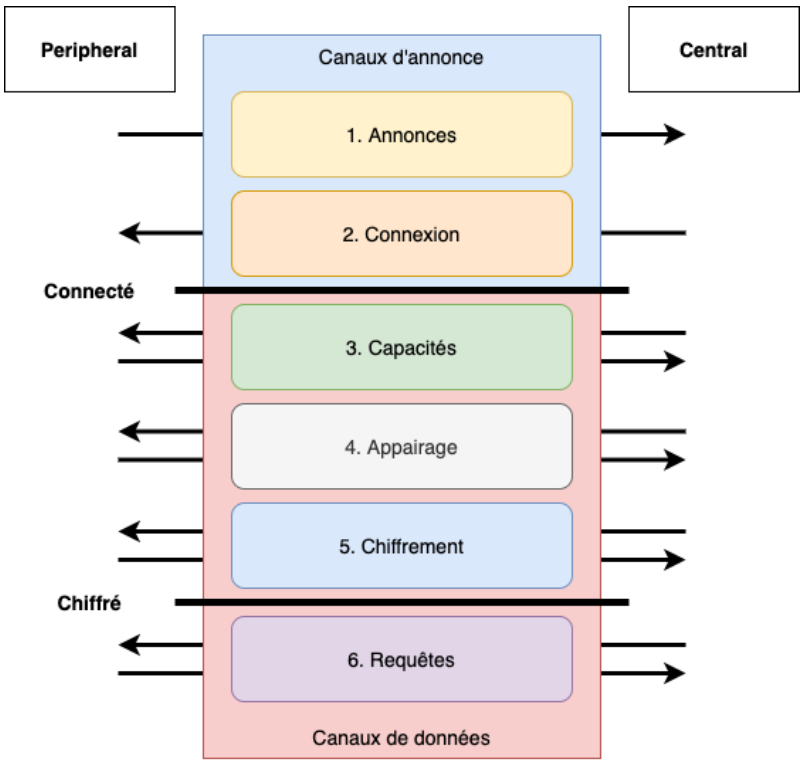

Figure 2: Étapes d'un échange BLE

Dans un environnement `BLE`, les *centrals* ne peuvent pas reconnaître leurs *peripherals* à part avec une addresse `Bluetooth` fixe, mécanisme de moins en moins utilisé car vulnérable à l'usurpation. Les *peripherals* génèrent donc des adresses aléatoires et l'identification se fait via les données du *GAP* contenues dans l'annonce. Ce mécanisme permet à n'importe quel *central* de s'appairer à n'importe quel *peripheral* proposant le profil recherché.
Par exemple, une application de *smartphone* `BLE` pouvant gérer la température pourrait s'appairer et utiliser n'importe quel appareil `BLE` qui implémente le profil standardisé pour les termomètres dans le *GAP*.
Les profils ne sont certes pas exhaustifs mais permettent une intégration fonctionnelle avec un maximum d'appareils et prévoient un moyen d'intégrer des données propriétaires non standardisées[8].

2. Connexion

Lorsqu'un *central* reçois une annonce d'un *peripheral* auquel il souhaite se connecter, il lui envoit une intention de connexion sur les canaux d'annonce. Ce message contient tout les paramètres communs pour établir une connexion sur les canaux de données: carte des canaux utilisés, temps entre chaque saut de fréquence, nombre de canaux sautés par saut, adresse unique de la connexion (appelée *Access Adress*).
Ce message (nommé *CONNECT_REQ*) est crucial lors d'attaques car il permet la synchronisation avec une connexion pour l'écoute passive et est donc jugé sensible puisque transmit sur les canaux d'annonces avant la mise en place du chiffrement.

3. Capacités

Le `BLE` voulant garder une interopérabilité maximale entre les appareils et ceux-ci ne

disposant pas des mêmes fonctionnalités embarquées, il est définit plusieurs méthodes d'appairage en fonction des capacités disponibles sur les deux appareils.
Chaque appareil va transmettre ses capacités à l'autre ainsi que ses exigences sur la connexion à établir. Les capacités sont déduites des fonctionnalités présentes physiquement sur l'appareil et les exigences dépendent de la version du procole actuellement supportée par celui-ci.
Les exigences comprennent la protection aux attaques *MITM* par l'authentification de l'appairage, l'établissement d'une connexion sécurisée (*LE secure connection*), la mise en place d'une session (*Bonding*) pour une reconnexion future ainsi que l'utilisation d'un canal autre que le `BLE` (comme le `NFC`) pour la transmission de secrets menant au chiffrement (*Out Of Band* ou *OOB*).

Tableau 2: Capacités d'entrée possibles[9]

Capacité	Description
No input	pas la capacité d'indiquer *oui* ou *non*
Yes/No	mécanisme permettant d'indiquer *oui* ou *non*
Keyboard	claver numérique avec mécanisme *oui*/*non*

Tableau 3: Capacités de sortie possible[9]

Capacité	Description
No output	pas la capacité de communiquer ou afficher un nombre
Numeric Output	peut communiquer ou afficher un nombre

Tableau 4: Capacité d'entrées/sorties de l'appareil[9]

	No output	Numeric output
No input	NoInputNoOutput	DisplayOnly
Yes/No	NoInputNoOutput	DisplayYesNo
Keyboard	KeyboardOnly	KeyboardDisplay

4. Appairage

En fonction des capacités et des exigences émises par chacun des appareils, une méthode d'appairage est sélectionnée (voir tbl. 5).

Tableau 5: Méthode d'appairage utilisée en fonction des capacités échangées[10]

	DisplayOnly	**DisplayYesNo**	**KbdOnly**	**NoIO**	**KbdDisplay**
DisplayOnly	JustWorks	JustWorks	PassKey	JustWorks	PassKey
DisplayYesNo	JustWorks	JustWorks	PassKey	JustWorks	PassKey
KbdOnly	PassKey	PassKey	PassKey	JustWorks	PassKey
NoIO	JustWorks	JustWorks	JustWorks	JustWorks	JustWorks
KbdDisplay	PassKey	PassKey	PassKey	JustWorks	PassKey

Je m'intéresse principalement à la methode *JustWorks*. C'est celle par défaut lorsque deux appareils ne disposent pas des capacités nécessaires pour une autre. Elle est notamment beaucoup présente pour les objets connectés puisque n'intégrant pas de mécanismes pour un appairage plus complexe (claver ou écran).
Passkey permet d'authentifier l'appairage pour se protéger des usurpations d'identité (*Spoofing* et *MITM*) puisque partageant un secret via l'utilisateur (ou un autre canal dans le cas du *OOB*). *JustWorks* ne permet pas d'authentifier les appareils et le chiffrement est moins robuste que les autres méthodes mais permet tout de même d'établir une communication chiffrée.

La méthode d'appairage choisie permet de transmettre un des matériel cryptographique: la clef temporaire (*Temporary Key*). Cette phase est plus ou moins sensible à l'écoute passive en fonction de la méthode d'appairage et des exigences émises lors de l'échange des capacités.
JustWorks avec connexion `BLE` 4.0 (dite *legacy*) est le mode le plus sensible puisque la clef temporaire est tout simplement zéro, ne disposant pas de moyen de transmettre une donnée par autre voie, elle peut donc être trouvée rapidement par *brute-force*.
La connexion *LE secure*, introduite à partir de la version 4.2, utilise l'algorithme Diffie-Hellman sur courbes elliptiques (*ECCDH*) pour l'échange du matériel cryptographiques et est donc résistante à l'écoute passive (*eavesdropping*) mais toujours vulnérable à l'usurptation d'identité avec *JustWorks*.

5. Chiffrement

L'établissement du chiffrement de la connexion est ensuite réalisé par dérivation de la première clef temporaire transmise via la méthode d'appairage choisie et d'autres matériel cryptographique échangés via le protocole `BLE`. La clef obtenue est dite court terme (*Short Term Key*) car elle ne sera utilisée que pour cette connexion et devra être re-générée à chaque nouvelle connexion.
Il est cependant possible de mettre en place une session en stockant une clef partagée dite long term (*Long Term Key*) si cela à été exigé lors de l'échange des capacités. La clef long terme (*LTK*) est stockée et associée à l'appareil communiquant pour rétablir une connexion future sans avoir à refaire une phase d'appairage.

À partir de la compréhension actuelle du protocole `BLE` et du fonctionnement de l'appairage, il semble recommandé de mettre en place une connexion sécurisée dès que possible. Il est également judicieux d'éviter la méthode *JustWorks* au maximum et stocker une session pour éviter d'avoir à réitérer l'appairage.
Cependant, il est assez simple de forger un échange de capacités pour rétrograder la connexion en *legacy* et forcer *JustWorks* via les capacités échangées. C'est pourquoi certains appareils attendent des capacités et exigences minimales pour établir une connexion, sans quoi celle-ci est avortée. C'est notamment le cas d'appareils propriétaires conçus pour fonctionner ensemble.

6. Requêtes

Les échanges sont réalisés sur la base d'une architecture client-serveur. Le *central* (client) interroge le *peripheral* (serveur) avec le protocole *ATT* (*ATTribute Protocole*). Chaque requête mène soit à une réponse du serveur, soit à la mise en place d'une notification lors d'un évènemment (valeur changée ou disponible).
Les requêtes et réponses possibles sont standardisées dans le *GATT* (*Generic ATTributes*)

pour permettre une interoperabilité maximale entre les appareils (comme pour le *GAP*). *GATT* et *GAP* partagent les mêmes profils, seul la structure change. Le serveur *GATT* peut être interrogé pour établir une liste exhaustive de toutes les fonctionnalités d'un appareil la ou le *GAP* choisit ce que contient l'annonce mais est limité par la taille du paquet (31 octets).

GAP

Dans le cas des *peripherals* et *centrals*, le *GAP* est principalement utilisé pour établir un profil du *peripheral* permettant la décision de connexion de la part du *central*.
Pour les *Boardcasters* et *Observers* il permet la communication unidirectionnelle (*Broadcaster* vers *Observer*) via les annonces, ceux-ci utilisant la diffusion plutôt qu'une connexion point à point. On retrouve cette utilisation pour les balises publicitaires ou de localisation intérieur.

GATT

Pour l'échange de données lors de connexion point à point, le *GATT* est utilisé en mode client-serveur. L'architecture du serveur *GATT* est en entonnoir, la plus haute couche s'appelle un *service*, il encapsule des *caractéristiques*, chacune contenant une *valeur* et un ou plusieurs *descripteurs* fournissants des informations additionnelles sur la valeur (voir fig. 3).
À chacune de ses couches (service, caractéristique, valeur, descripteur) est attribué un identifiant unique appelé *handle*. La plage des indentifiant est partagée entre toutes les couches donc si un service a l'identifiant `0x01` aucun autre service/caractéristique/descripteur ne peut l'utiliser.

Un service correspond généralement à un profil (standardisé ou non) comme par exemple un termomètre. Ce service exposerait des caractéristiques comme la température ou l'humidité. Chacune de ces caracatéristique contient la valeur (donnée brute) et des descripteurs pour indiquer l'unité ou encore un facteur ou formule pour convertir la valeur donnée en résultat exploitable.

À moins de connaître exactement l'appareil et de l'interroger à l'aveugle via les *handles* (ce qui peut être le cas entre des appareils propriétaires), il faut procéder par étape en découvrant d'abors les services disponibles, puis chaque caractéristique par service et enfin les descripteurs.
Pour procéder à la découverte d'un appareil, le protocole *ATT* dispose d'un type de requête par couche à interroger (voir fig. 3). Une fois le service voulu trouvé (ou la cartographie totale de l'appareil réalisée), on peut lire, écrire ou souscrire à des caractéristiques directement par *handle*. Le *GATT* met en place un système de droits par valeur pour protéger la lecteur, l'écriture et la souscription par le client.

Le *GATT* définit égalemet des services standardisés, appelés primaire et secondaire, censés êtres présents sur tous les appareils `BLE` afin de connaître les fonctionnalités standardisées (service primaire) et propriétaires (service secondaire) de l'appareil. Comme les *handle* sont définies arbitrairement par le serveur *GATT*, les profils standards et leurs services/-caractéristiques sont identifiés par un *UUID* standardisé identique dans tout les appareils `BLE`[11].

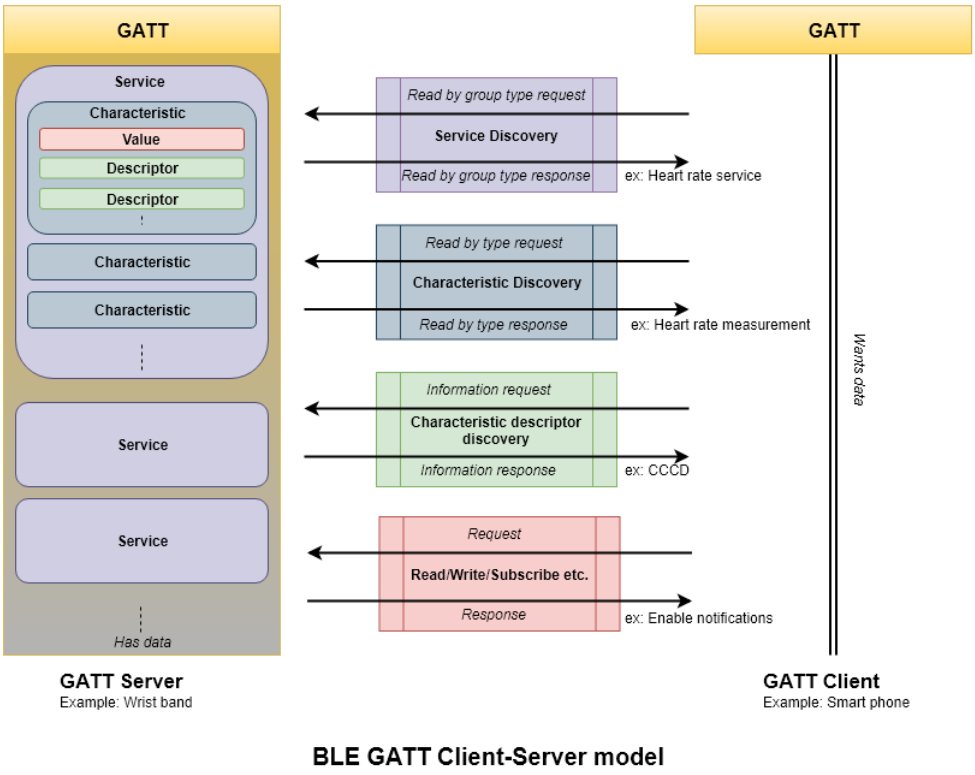

Figure 3: Client et serveur GATT[12]

Évolution

Depuis sa première itération en 2010 dans la version 4.0 des spécifications `Bluetooth`, le `BLE` à évolué pour intégrer des mesures de sécurité avec l'ajout des connexions sécurisées *LESC* en 4.2 puis la diversification des topologies avec l'introduction du *mesh* pour les réseaux de capteurs en `5.0` et dernièrement l'amélioration de la localisation intérieur (*Indoor Positionning System*) pour une précision de l'ordre du centimètre grâce aux systèmes angle d'arrivée et de départ (*AOA/AOD*).

3 Preuve de concept

Sujet: Étudier puis mettre en place des attaques sur le protocole *Bluetooth Low Energy* (également appelé *Bluetooth Smart*).

Le but est d'exposer puis abuser des failles dans le protocole BLE 4.0 (première version). Ces failles sont connues et ont pour la plupart été corrigées dans les versions ultérieures du protocole (aujourd'hui en version 5.1). Néanmoins cela permet d'étudier et comprendre les mécanismes du BLE depuis sa création, puis voir les alternatives qui ont été proposées pour mitiger ces attaques.
Je ne cible que les appareils supportant l'appairage en BLE 4.0 (dit *legacy*) avec méthode d'appairage *JustWorks*. C'est le niveau de sécurité minimal prit en charge par le protocole et très répandu dans les appareils connectés BLE. La majeure partie des appareils connectés sont simples, ne réalisant qu'une fonction d'augmentation, ne disposant ainsi pas de clavier ou d'écran et ne permettent donc pas l'utilisation de méthode d'appairage autre que *JustWorks*.
Les mécanismes proposés à partir de la version 4.2 du BLE sont beaucoup plus robustes. Ils apportent une connexion securisée (*LE Secure Connections* ou LESC) se basant sur Diffie Hellmann pour l'échange de clefs ainsi qu'une nouvelle méthode d'appairage authentifiée (Comparaison numérique).

Même si le BLE à toujours proposé des mesures de sécurité, la majorité des constructeurs préfèrent mettre en place un chiffrement au niveau de la couche application (le BLE chiffre depuis la couche lien).
Ces chiffrements propriétaires sont souvent basés sur des algorithmes reconnus comme AES et des méthodes telles *challenge-response* pour authentifier un appareil. Une clef unique est integrée dans chaque appareil, celle-ci sera soit distribuée au propriétaire de l'appareil lors de la création du compte ou le téléchargement de l'application associée, soit gardée par le constructeur qui transmettra directement les commandes depuis l'utilisateur à l'appareil (via l'application, ou directement si l'appareil est connecté au réseau mondial).
Ces mécanismes exposent cependant beaucoup plus d'informations que le chiffrement BLE, réalisé depuis la couche lien. Les requêtes ATT et $GATT$ transittent en clair et pour pallier à cette fuite d'informations les constructeurs évitent les requêtes standardisées dans le BLE et préfèrent utiliser des protocoles personnalisés dans la couche application, celle-ci étant chiffrée.
Ces chiffrements propriétaires sur la couche application sont hors de portée de mon sujet mais ont fait couler beaucoup d'encre. Plusieurs présentations et leurs *whitepaper* sont disponibles dans des conférences de sécurité offensive comme *black hat*[13], *Defcon*[14] ou encore *SSTIC*[15].

Maintenant il s'avère que beaucoup d'appareils autonomes simples ne mettent aucune mesure de sécurité en place, quelles soient standardisées ou propriétaires, car les données qui transittent ne sont pas jugées sensibles. C'est notamment le cas des objets domotiques autonomes comme les télécommandes pour lampes dites connectées.

3.1 Travail demandé

Mettre en place un outil basé sur un framework offensif permettant de répertorier et faciliter l'analyse des appareils et connexion BLE alentours. Cet outil est facilement portable sur diverses cartes de développement comme la Raspberry Pi car conteneurisé avec Docker et se basant sur du matériel USB pour l'étude du protocole.

3 ports USB suffisent pour permettre de conduire toutes les attaques proposées par le framwork offensif utilisé: 2 dongles USB `BLE` 4.0 et une carte `BBC micro:bit`.
Le projet suppose la mise en place de 3 attaques dont une nouvelle qui ne fait actuellement pas partie de `Mirage`: - Inventaire des appareils et connexions à proximité + localisation des appareils (*scan* et *sniffing*) - Usurpation et mise en place d'un *Man In The Middle* sur un appareil selectionné (*spoofing*) - Synchronisation puis détournement par brouillage d'une connexion précédement identifiée (*hijacking*)

3.2 Architecture

Le *back-end* en `python` permet le pilotage de `Mirage` via une interface logicielle (*Application Programmable Interface*, en rouge sur fig. 4). Cette *API* fait le lien entre le serveur `HTTP/Websocket Flask` (violet) et `Mirage` (bleu) qui pilote le matériel nécéssaire pour les attaques (vert). J'ai opté pour une interface web côté *front-end* car les technologies (surtout du `javascript`) ont explosées ces dernières années, rendant l'intégration d'intefaces plus simples et flexibles que les interfaces graphiques applicatives. Cette page se voulant un panneau de controle, un framework `javascript` (`Hyperapp`, en bleu clair) permet l'interactivité recherchée. Concernant les communications, il me fallait un protocole à double sens puisque l'on doit pourvoir suivre l'avancée d'une attaque en temps réel, ce que ne permet pas `HTTP`. Les *websockets* etant tres populaires dans les applications `javascript` comme les mini-jeux, jai intégré `Socket.IO` (jaune) en front et *back-end*: c'est une librairie de communication evenementielle basee sur les *websockets*. `Hyperapp` ou `Flask` souscrivent aux evenements de `Socket.IO` pour lancer des attaques sur `Mirage` ou modifier le `DOM` en adéquation (*Document Object Model*, c'est la vue `HTML` représentée en rouge sur fig. 4).

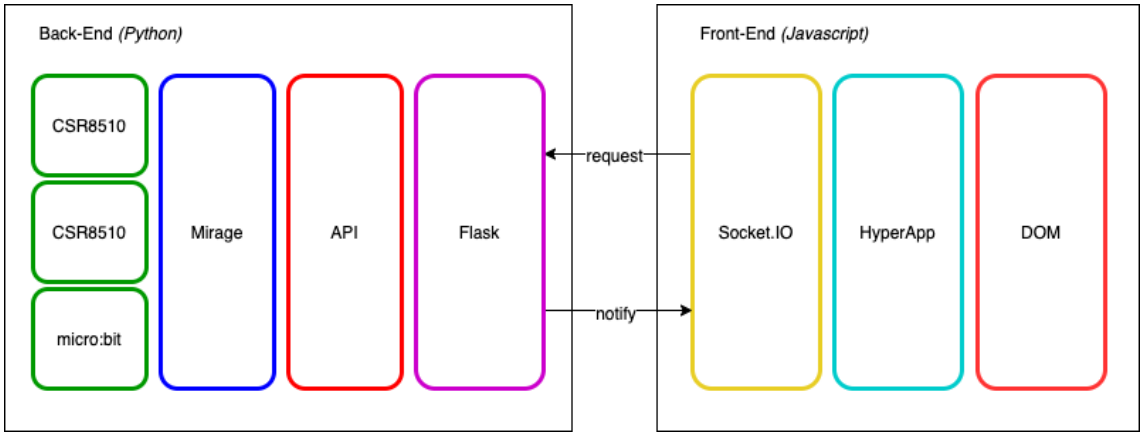

Figure 4: Architecture du système

3.3 Interface

Elle sert de panneau de contrôle pour la supervisation du *scan* et *sniffing* des appareils et connexions alentours. Le but est d'augmenter `Mirage` avec une interface graphique (*Graphical Uuser Interface*) pour les attaques implémentées dans le projet. C'est un environnement plus familier que le terminal et ces ligne commandes (*Command Line Interface*), rendant plus accessible les démonstrations de sensibilisation.
On retrouve les appareils et connexions identifiées par le *scan* sur la colonne de droite (fig. 5). La colonne de gauche est découpée en trois parties: les contrôles permettant de lancer ou interrompre une attaque, une carte affichant les appareils localisés ainsi que leur distance et les messages remontés par `Mirage` (*logs*).

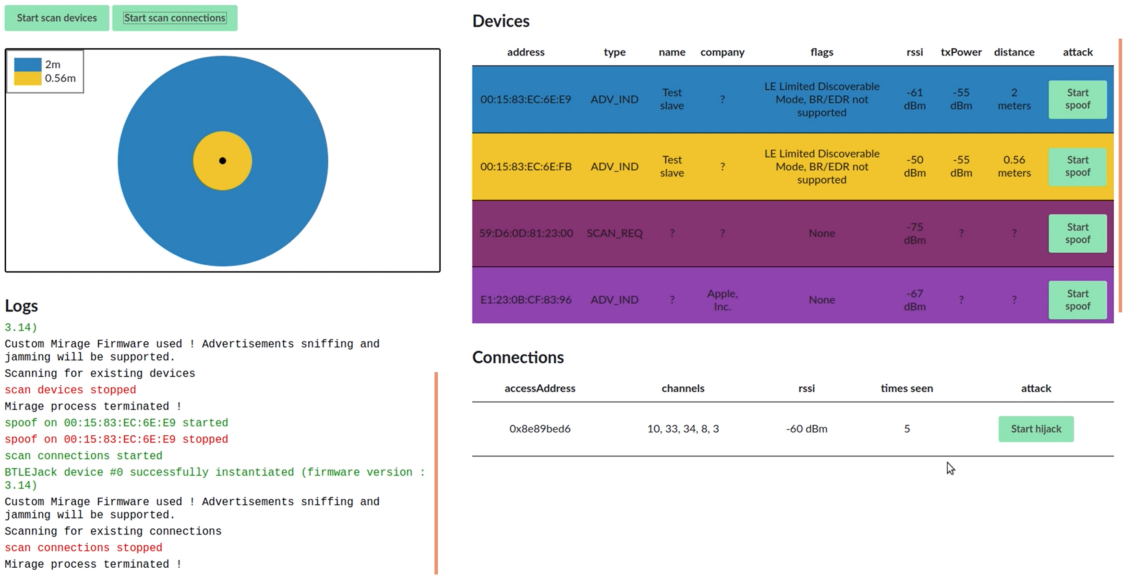

Figure 5: Interface du système

4 Étude de l'existant

4.1 Outils

Les outils offensifs sur le protocole `BLE` permettent de récupérer, analyser et modifier les échanges entre appareils, ce qui permet de réaliser des audits de sécurité ou encore mettre en place des attaques pour exposer des vulnérabilités. L'analyse du traffic sans fil BLE demande une antenne couvrant la bande utilisee par les 40 canaux du protocole ainsi qu'un systeme assez rapide pour scanner puis suivre les communications lors des sauts de fréquence. Les radio-logiciels (*SDR*) ne sont donc pour la plupart pas adaptés car trop lents ou trop cher pour les fonctionnalités voulues: des outils spécialisés dans l'analyse et l'attaque du `BLE` sont disponibles pour une fraction du prix.

Le premier outil, utilisé dans tout nos appareils `BLE`, est la puce intégrée pour les communications `BLE`. Ces puces sont limités par un *firmware* spécifique qui ne permet pas la récupération (*sniffing*) et analyse ou modification d'un traffic sans fil, cela étant interdit. Même si il reste possible d'analyser le traffic entre la puce et un appareil `BLE` (notamment en utilisant *Wireshark*[16]), il n'est pas possible d'étendre les capacités de celle-ci sans modifier le *firmware*.
Tout les appareils ne disposant pas d'une puce `BLE` dédiée, les constructeurs ont développés des *dongles* qui intègrent ces puces et permettent de communiquer avec tout appareil USB via une interface nommée *HCI* (*Host-Controller Interface*). Alliés aux outils standard du protocole `BLE` comme `BlueZ`, la pile protocolaire BLE du noyau Linux, ces *dongles* permettent de découvrir les appareils à proximité et d'endosser le rôle de *peripheral* ou *central* pour établir une communication avec n'importe quel autre appareil `BLE`. Les utilitaires `hcitool`, `hciconfig` et `gatttool` de `BlueZ` permettent par exemple de manipuler les annonces et extraire le profil *GATT* d'un appareil `BLE`. Même si certains de ces *dongles* proposent des fonctionnalités intéressantes comme le changement d'adresse `Bluetooth` (*Bluetooth Address* ou *BD*), ils n'ont pas été conçus dans une optique de sécurité, et sont peu flexibles pour un usage offensif.

La plupart des attaques sur le protocole BLE requièrent un moyen d'intercepter le traffic. Puisque les *dongles* et puces ne permettant pas cette fonctionnalité car destinés au grand public, plusieurs outils spécialisés ont emergés. On va retrouver des outils d'analyse de protocole sans fil généralistes comme la HackRF ou sa version spécialement conçue pour le BLE nommée Ubertooth One. Ces cartes sont assez cher mais hautement personnalisables depuis les couches bas niveau. Elles demandent un certain background de connaissances sur le protocole et les modulations sans fil pour arriver à un résultat précis comme la réalisation d'une attaque.
Viennent ensuite les *sniffers* sous forme de dongle USB arrangés et plus ou moins personnalisables. Beaucoup sont basés sur les mêmes puces de *Nordic Semiconductor* ou *Texas Instrument* qui eux meme proposent leurs sniffers[17, 18] et logiciels[19, 20] pour l'analyse du protocole BLE. Dans les initiatives plus open-source, mais pas encore totalement personnalisable sans reprogrammation de la puce, on peut citer le Bluefruit[21] de *Adafruit*.
Enfin, un outils open-source appelé BTLEJack[22] permet non seulement l'étude mais également la mise en place d'une multitude d'attaques sur le protocole BLE via reprogrammation d'une carte de développement avec un *firmware* personnalisé. Cet outils à été développé pour la carte *BBC micro:Bit*[23], une carte de developpement bon marché à but éducatif, puis porté sur plusieurs autres cartes intégrant une puce nRF51 (notamment la Bluefruit). Basé sur les travaux de BTLEJack et d'autres librairies BLE en python, Mirage[24] permet des fonctionnalités identiques en supportant encore plus de cartes, de protocoles et d'attaques. Il comble le manque de flexibilité des précédents outils en intégrant plusieurs mécanismes qui permettent la mise en place d'attaques entièrement personnalisées, dites scénarisées, depuis les couches protocolaires basses et facilite l'ajout de fonctionnalités au sein du *framework*.

4.2 Attaques

Scanning

Le *scanning* consiste à répertorier des appareils BLE à proximité, on peut étendre l'inventaire aux connexions établies entre 2 appareils BLE. Là ou les *dongles HCI* suffisent pour intercepter les annonces diffusées, l'analyse des communications établies requiert un *sniffer* capable de suivre les 37 canaux de données.
La pile protocolaire *BlueZ* permet le *scan* des *advertisements* (annonces) tandis que plusieurs outils precedements evoques comme smartRF ou nRFSniffer suffisent pour repérer une communication.

Spoofing

C'est l'une des étape du *Man-In-The-Middle* qui permet d'usurper un *peripheral* BLE. Après identification de la victime (via annonces ou adresse *BD*), l'attaquant la clone en s'y connectant et extractant son profil *GATT*. L'attaquant peut rester connecté pour garder la victime silencieuse (un *peripheral* connecté n'émettant pas d'annonces) puis, via un *dongle HCI*, s'annonce comme étant l'appareil précédemment cloné.
Cette attaque est réalisable en utilisant simplement un *dongle HCI* et l'utilitaire *BLueZ*. Les librairies et *frameworks* d'attaque discutés plus auparavant (*GATTAcker*, *BTLEJack*) intègrent également ces mécanisme.

Sniffing

Le *sniffing* est l'analyse voir le suivis d'une connexion `BLE` (suivant les capacites du *sniffer* utilisé). Un premier cas de figure est l'attente d'une nouvelle connexion pour se synchroniser avec afin de suivre les échanges. La seconde option, et la plus courante, est la synchronisation avec une connexion auparavant établie: la difficulté ici réside en la récuperation des paramètres de connexion. Il est nécessaire de retrouver la carte des canaux utilisés (*channel map*) ainsi que le *hop increment* (nombre de canaux sautés) et *hop interval* (temps entre chaque saut) pour se synchroniser, sans quoi il est impossible de suivre une connexion à cause des sauts imprédictibles et trop fréquents.
`BTLEJack`, et par conséquent `Mirage`, mettent en place un mécanisme permettant de retrouver ces informations de connexion à partir des échanges interceptés lors du *scanning*. Le *sniffing* de communications sans synchronisation quant à lui est une fonctionnalité très répandue et integrée à tout les sniffers `BLE` vu antérieurement.

Man-In-The-Middle

L'attaque *Man-In-The-Middle* concerne n'importe quelle communication: l'attaquant peut modifier les échanges en venant se placer entre l'émetteur et le récepteur ciblé, se faisant passer pour l'un après de l'autre en usurpant leurs identités. Dans le cas du `BLE` on utilisera deux *dongles*, un pour usurpé le *peripheral* cible et un autre pour maintenir une connexion avec celui-ci. On doit d'abors usurpé le *peripheral* cible via du *spoofing* puis attendre la connexion d'un *central*. Une fois le *central* connecté à notre *dongle* usurpateur on se retrouve en situation de *Man-In-The-Middle* entre le *peripheral* et le *central*: on peut suivre et modifier le traffic avant de le retransmettre (fig. 6). A noté que l'on peut également usurpé le *central* si les appareils ciblés attendent un profil précis pour s'appairer.

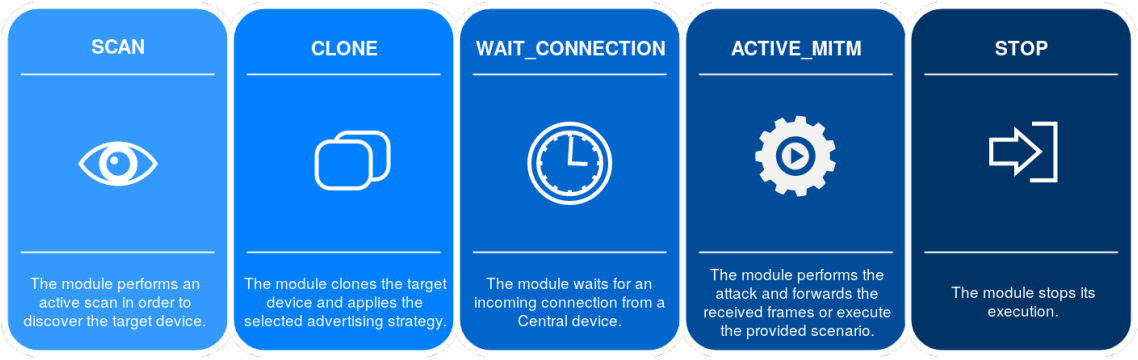

Figure 6: Étapes d'une attaque *MITM*[25]

Plusieurs outils sont dediés à cette attaque car populaire et simple à mettre en oeuvre: `GATTAcker` et `BTLEJuice` facilitent la mise en place en automatisant les étapes à partir de la cible choisie. Ce sont d'assez ancien outils qui aujourd'hui souffrent de lacunes dûes aux technologies utilisées. Basés sur `Noble`[26] et `Bleno`[27], des librairies `javascript` pour `NodeJS` et permettant de manipuler le `BLE`, ils manquent de flexibilité et ne permettent pas entre autre la coexistence d'appareils `BLE`, obligant l'utilisation de machine virtuelle pour chaque *dongle*[28]. `Mirage` reprend le fonctionnement de ces outils, l'intégrant en tant que module, mais basé sur de nouvelles librairies, notamment `PyBT`[29] permettant de simuler le comportement d'un appareil BLE en s'affranchissant des contraintes imposées par leurs équivalent `javascript`, `Noble` et `Bleno`.

Jamming

Le brouillage de communication est également une attaque assez populaire et implémentée dans bon nombre d'outils sur le marché. Le but est de créer du bruit sur le canal au moment de la transmission pour corrompre le message, le rendant inutilisable par le récepteur. Concernant le `BLE`, `Ubertooth One` dispose des capacités nécessaire pour brouiller les canaux d'annonce ainsi qu'une communication établie par retransmission simultanée. `BTLEJack` implémente également le brouillage au sein de son firmware personnalisé et, ayant déjà un mécanisme permettant de se synchroniser avec une communication, peut également brouiller une communication établie.

Hijacking

Le principe est de voler une connexion entre 2 appareils en forcant une déconnexion de l'un pour prendre sa place. Cette nouvelle attaque, implémentée par `BTLEJack` et reprise dans `Mirage`, utilise les différences de *timeout* entre le *central* et le *peripheral* pour forcer le *central*, via l'utilisation de brouillage sur les paquets du *peripheral*, à se déconnecter et prendre ainsi sa place au sein de la communication.

4.3 Mirage

Après comparaison entre les outils disponibles (tbl. 6), j'ai choisi `Mirage` car il dispose de la flexibilité voulue pour implémenter des attaques scénarisée: accès aux couches bas niveau pour récupérer dds informations comme force du signal et calibrage (nécessaires pour calculer la position d'un appareil lors de la localisation). Il dispose également d'une implementation d'un *central* et *peripheral* personnalisables permettant la mise en place d'un réseau de tests sur lequel vérifié l'implémentation des attaques. Enfin, il supporte une variété de composants matériel[30] ainsi que toutes les attaques nécessaires[25] pour la preuve de concept, rendant le développement plus simple.

Tableau 6: Comparaison des outils pour l'étude offensive du BLE

Logiciel	*scan*	*sniff*	*mitm*	*jam*	*hijack*	*locate*	Matériel
nRFSniffer	oui	oui	non	oui	non	non	puce `nRF51`
TIsmartRF	oui	oui	non	non	non	non	puce `CC25xx`
BTLEJuice	oui	non	oui	non	non	non	*dongle HCI* + `Bleno`/`Noble`
GATTAcker	oui	non	oui	non	non	non	*dongle HCI* + `Bleno`/`Noble`
BTLEJack	oui	oui	oui	oui	oui	non	`BBC micro:bit`, cartes basées sur puce `nRF51`
Mirage	oui	oui	oui	oui	oui	possible	*dongle HCI*, `Ubertooth`, `nRF kit`, cartes compatibles avec `BTLEJack`

`Mirage` fournit des briques logicielles pour communiquer avec des appareils (*dongles*, *sniffers*) ainsi que décortiquer les protocoles, ce qui constitue le coeur du *framework*. Ces briques logicielles de base sont utilisées par des *modules* dans un but précis, par exemple réaliser le brouillage d'une communication `BLE`. `Mirage` fourni un certain nombre de modules qui mettent en place les attaques retrouvées dans les autres outils d'audit du `BLE` (`BTLEJack`, `GATTAcker`, ...) précédemment discutés. Similaire à la philosophie d'`Unix`, ces modules sont spécialisés dans une tâche précise et peuvent êtres assemblés entre eux pour réalisé des fonctionnalités plus complexes telle la connexion avec le module

`ble_connect` puis l'extraction d'informations avec `ble_discover`. Enfin, il est possible de modifier les étapes d'une attaque via les *scénarios*: chaque module accepte un scénario surchargeant son flux d'exécution et ses méthodes.

Concernant le matériel necessaire, la preuve de concept nécessite d'abors deux *dongles HCI* compatibles avec `Mirage` et supportant le changement d'adresse *BD* pour le *spoofing*. `Mirage` se base sur les numéros de constructeurs des *dongles*, définit par le *Bluetooth SIG*[31], pour savoir s'ils sont compatibles. Il supporte une variété des constructeur dont le `CSR8510`[32] de Qualcomm (fig. 8), puce très populaire dans les *dongles HCI* basiques et permettant le changement d'adresse *BD*.
Concernant le *sniffer* nécessaire pour la pupart des attaques, `Mirage` se basant sur `BTLEJack`, les cartes supportées par celui-ci le sont également. Même si `Mirage` supporte d'autres cartes comme des kits de développement de *Nordic Semiconductor* ou l'`Ubertooth One`, elles ne sont pas adaptées pour mon projet car trop onéreuses pour les fonctionnalités exploitées dans la preuve de concept.
Je me suis donc tourné vers les cartes compatibles avec `BTLEJack`[33]. `Bluefruit` d'Adafruit, Waveshare `BLE400`[34] et les kits `nRF51`[35] demandent une reprogrammation via un périphérique externe utilisant le port *SWD*. Ne disposant pas du matériel nécessaire pour la reprogrammation, et celui-ci étant assez onéreux, j'ai choisit la `BCC micro:bit` (fig. 7): carte avec laquelle `BTLEJack` a été originalement développé.

Figure 7: Carte BBC micro:bit

Figure 8: Dongle HCI BLE CSR8510

Intégration

`Mirage` est un "*framework offensif pour l'audit des protocoles sans fil*"[24]. Il a été pensé pour le *pentest* (audit de sécurité) donc un usage exclusivement en ligne de commandes (*CLI*). Mème si le framework se veut beaucoup plus modulaire et extensible que ces prédécesseurs (`BTLEJack` notamment), cette modularité est exclusive à l'interface en ligne de commandes pour laquelle il est pensé.

Le fait de devoir l'intégrer dans un *back-end* suppose une *API* pour communiquer avec `Mirage` depuis `Flask`. Ne disposant pas nativement de cette *API* je l'ai créée en m'inspirant de celle du *CLI*: j'ai repris la méthode d'initialisation du framework mais est remplacé les arguments en ligne de commande par une instanciation et hydratation des modules manuelle.
Le *framework* est fait pour fonctionner le temps d'une attaque, après quoi il s'auto-termine, et compte sur le nettoyage par le système d'exploitation suite à la fin d'exécution de son processus pour fermer les *sockets* ou vider les files (*FIFO*) utilisées dans la communication avec le matériel par lien série. Dans cette philosophie j'ai été amené à combler ce manque car mon processus `python` est hôte de `Mirage`, il ne se ferme pas à la fin d'une attaque, retrouver un etat stable apres une attaque est donc primordial. Cela demande la suppression

des caches utilisés dans l'instance de `Mirage`, fermeture des *socket* et synchronisation des fils d'exécutions (*threads*) qui supervisent le matériel, pour ne pas remplir les files alors que l'attaque est terminée.

Si cela peut avoir du sense de faire une interface pour superviser une nouvelle attaque ajoutée à `Mirage`, reconduire le fonctionnement et interactions des attaques d'ores et déjà disponibles via le *CLI* vers un *GUI* est discutable. Le *MITM* et *hijacking* sont des attaques interactives: après usurpation d'un appareil dans le *MITM*, l'utilisateur peut modifier à la volée les paquets échangés ou communiquer via un terminal avec le *peripheral* lorsque qu'une connexion à été détournée avec du *hijacking*. Cette interaction n'est reproductible qu'en imittant un terminal sur l'interface, ce qui revient a recréer l'interpréteur déjà intégré à `Mirage` pour n'ajouter qu'un peu de commodité à l'utilisateur (qui n'a pas à devoir utiliser le *CLI* depuis le conteneur `Docker`).

`Mirage` utilise des délais d'attente (`wait`, `sleep`) dans le processus principal pour attendre une certaine trame ou qu'un appareil soit dans l'état voulu. L'interactivité est conservée par des mises à jour periodique de l'avancée de l'attaque soit via des tableaux contenant les informations trouvées soit des messages indiquant un événement précis. Pour garder cette même interactivité dans le *front-end* durant l'exécution d'une attaque est plus complexe car la librairie `Socket.IO` dans le *back-end* ne se base non pas sur des fils d'exécution mais une boucle d'événements (principe utilisé dans `NodeJS` et popularisé par le `javascript`). Les événments lents comme les interactions avec le réseau sont toujours différées car le fil d'exécution principal est lui meme interrompu par de multiples délais d'attentes. En conséquence `Socket.IO` ne transmet jamais les événements et le *front-end* reste figé. Pour remédier à cela il à d'abors fallut rendre `Mirage` compatible avec le système de boucle d'événements, solution fournie par `Socket.IO` (qui plus est sans modifier le code du *framework*) par le remplacement des méthodes standard de suspension du fil d'éxécution (`sleep`, `wait`) avec leurs équivalents en boucle d'événements. Ensuite, plutôt que de lancer l'instance de `Mirage` sur son propre fil d'exécution, on l'insert dans la boucle d'événements qui supervise tout le *back-end*. Les fils d'exécution crées par `Mirage` suspendent ainsi son événement dans la boucle et non le fil d'exécution de la boucle d'événement lors d'appels a `sleep` et `wait`.

5 Travail réalisé

5.1 Scan

Le *scan* des appareils et le *sniffing* des connexions BLE alentours se base sur un *sniffer* `BLE`: la carte `micro:bit` dans mon cas. Les fonctionnalités de *scan* sont nativement supportées par `Mirage` et intégrées dans le firmware adéquat au framework.
C'est l'une des deux attaques retrouvées dans le *front-end*: l'utilisateur peut commencer un *scan* qui notifiera le *front-end* lors de découvertes, puis l'arrêter quand bon lui semble. Les appareils et connexions répertoriées ainsi que leurs informations sont disponibles sur la colonne de droite (fig. 9).

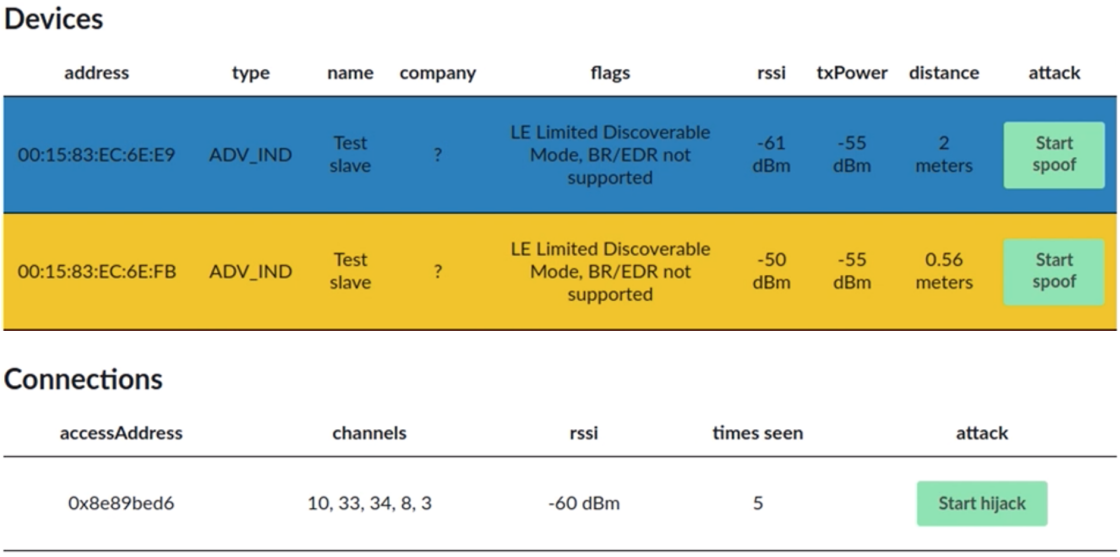

Figure 9: Appareils et connexions repertoriées à proximité

La carte `micro:bit` n'intégrant qu'une puce `nRF51`, une seule commande peut être réalisée à la fois. `Mirage` met cependant en place du balayage de canaux basé sur un changement rapide de commandes directement dans le *firmware* via les minuteurs disponibles sur la carte. Ce balayage permet la découverte d'appareils `BLE` sur les canaux d'annonces 37, 38 et 39 avec une seul carte `micro:bit`.
Le sniffing des connexions est peu fiable dû au changement imprédictible de canaux imposé par le *channel hopping*. Cette mitigation intégrée au protocole `BLE` rend incertain le temps pour identifier une ou plusieurs connexions `BLE`, la carte `micro:bit` changeant elle aussi de canaux pour maximiser ces chances de trouver des connexions les utilisants.

Cette attaque profite du caractère publique des canaux utilisés pour les communications, il est possible de mitiger son impact en rendant plus difficile l'identification des appareils par la réduction du nombre d'annonces émises et en choisissant le type d'annonce en fonction des besoins. Il n'est pas toujours nécessaire d'émettre des annonces indirecte contenant des données du *GAP*, les annonces directes contiennent par exemple seulement le *central* recherché, rendant plus complexe la tâche d'identification de l'appareil. Le *sniffing* des connexions peut également être durci en modifiant les paramètres de connexion émis plutôt que d'utiliser une carte des canaux par défaut se basant sur les 37 canaux de données.

5.2 Localisation

Il existe plusieurs moyens de localiser des appareils (fig. 10), la localisation intérieur est d'ailleurs un champ de recherche complexe et très actif allant de l'inventaire d'entrepots jusqu'au profilage publicitaire.
Pour obtenir une estimation de la distance d'un appareil on peut se basé sur le temps que met l'onde à nous parvenir (appelé *Time Of Arrival* ou *TOA*) pour en déduire la distance à partir de sa vitesse. Cependant cela requiert une information fournie par l'émetteur: l'heure d'émission. En me placant en tant qu'attaquant je ne controle pas les appareils ciblés et ne peut pas garantir la présence de cette information car peu utilisée dans les appareils particuliers.
Une autre méthode beaucoup plus populaire et accessible se base sur le *RSSI* (*Received Signal Strength Indicator*). C'est un indicateur de la puissance du signal reçu en `dBm` duquel peut être déduit la distance de l'émetteur. Cependant, le `BLE` pouvant émettre sur une plage de puissances, il est primoridial de connaître ou trouver la puissance d'émission utilisée de la part de l'émetteur. Heureusement un standard à été développé pour les balises, nommé *iBeacon* et intégré dans le *GAP* et le *GATT* en tant que *Tx Power* (puissance de transmission). Il fourni une valeur de calibrage qui représentative de la puissance mesurée par le constructeur à 1 mètre. Même si sa présence n'est pas garantie, le standard est très répandu dans les appareils domestique et de bureautique.
Les entrepots et centres commerciaux utilisent le *fingerprinting*, c'est à dire le positionnement par rapport a des appareils proches identifiés. Chaque appareil est répertorié avec sa position et son calibrage, l'objet à localiser applique ensuite une *trilatération* (fig. 10) à partir de la position de 3 appareils à proximité. Cette solution n'est adaptée à mon besoin car elle demande la liste des appareils identifiés, information indisponible en tant qu'attaquant.

Ensuite viens une seconde problématique, le *TOA* et *RSSI* ne fournissent pas d'information sur la direction de l'appareil, seulement une distance. Il faut alors croiser plusieurs relevés avec de la *trilatération*, procedé au coeur du système GPS visant a trouver une intersection commune entre minimum 3 cercles (voir fig. 10), ou utiliser une matrice d'antennes pour calculer la direction du signal reçu.
Le `BLE` intègre depuis la version 5.1 un mécanisme d'angles de départ et d'arrivée (*AOA*/*AOD*) permettant de trouver la direction à l'aide d'une matrice d'antennes en plus d'augmenter la précision de l'ordre du mètre au centimètre. Chaque antenne reçois le signal avec un décalage par rapport à ses voisines, ce décalage temporel est utilisé pour approximer l'angle d'émission. En utilisant cette technique, et à partir de plusieurs émetteurs, on peut determiner une position sans utiliser le *RSSI* ou *TOA* mais en utilisant la *triangulation* (fig. 10).

Pour ma part, je travail sur la version 4.0 du protocole `BLE`, qui n'intègre pas le mécanisme *AOA*/*AOD*. Même si il reste possible de mettre en place cette methode sans la version 5.1 du protocole, cela requiert une matrice d'antennes, matériel indisponible au vu des conditions exceptionnelles.
J'ai opté pour le *RSSI* au vu de la popularité et facilité de mise en place de la méthode. Les relevés sont fortement impactés par l'environnement, l'étude de celui-ci et la mise en place de modèles apadtés étant impossible dans mon cas, j'ai fixé le facteur environnmental en tant qu'espace dégagé. Je laisse tout de même la possibilité à l'utilisateur de modifier ce facteur si besoin. Le second facteur est la sensibilité de réception, la puce `nRF51` garantie une valeur à plus ou moins 6dBm avec un seuil de -30 a -90dBm, mon but est donc de réduire l'impact des ces écarts.

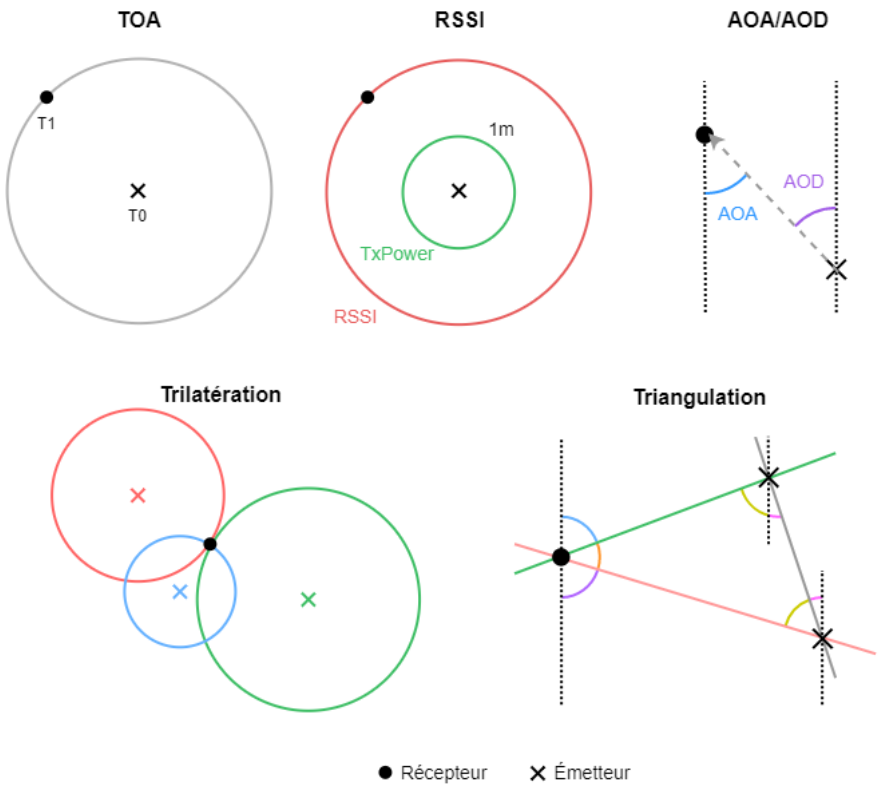

Figure 10: Différentes méthodes pour la localisation intérieur

Pour les appareils en mouvement on peut ajouter de la précision avec le *dead-reckoning*, permettant de faire des prévisions de position à partir de celle actuelle et des capteurs intégrés à l'appareil (gyroscope, accéléromètre). On retrouve ce mécanisme pour les appareils ou applications *GPS*, tirant avantage des capteurs intégrés dans nos *smartphones*. Des modèles mathématiques comme le filtre de *Kalman* permettent également d'approximé les prochaines valeurs.

Le fait de se placer en tant qu'attaquant donc de ne pas controller les appareils rend le *dead-reckoning* inutilisable. J'ai choisit le modèle de pertes le plus fréquement utilisé pour le *RSSI* puisque un modèle de pertes personnalisé pour un environnement donné n'est pas envisageable car le projet est fait pour de la sensibilisation et est donc amené à en changer frequement.
Dans le but de réduire les écarts, j'ai commencé par un lissage des valeurs sur une fenêtre modifiable. Cela me permet de confirmer une tendance, minimisant l'impact des fluctuations du *RSSI*. Le filtre de *Kalman* semble être une amélioration intéressante et pourrait être une prochaine étape.

Intégration

Dans `Mirage`, la localisation se base sur le travail d'identification précédement réalisé par la phase de *scan*. Le *firmware* `Mirage` intègre des informations relevées depuis la puce `nRF51` sur la transmission reçue comme la puissance du signal (*Received Signal Strength Indicator* ou *RSSI*). À partir de cette information ainsi qu'un calibrage (`TxPower`) il est possible d'approximer la distance avec le modèle de pertes suivant:

$$distance = 10^{(TxPower-RSSI)/(10*n)}$$

Ou `n` est le facteur environnemental variant de 2 (espace dégagé) à 4 (zone urbaine).

Les fonctionnalités de localisation ont été intégrées en tirant profit des possibilités d'extensiblité de `Mirage` avec l'ajout d'un nouveau module: `ble_locate`. Le module partage beaucoup de fonctionnalités avec `ble_sniff`, permetttant le *scan* des appareils et connexions à promixité mais modifie l'*API* de `BTLEJack` au sein de `Mirage` pour faciliter le *sniff* de connexions. Mon but était de faire avec ce qui était intégré au *firmware* `Mirage` car sa recompilation demande la mise en place d'un environnement précis[36]. Le module se conforme aux codes de `Mirage` et pourrait être fusionné au sein du *framework* comme fonctionnalité supplémentaire à l'avenir.
Concernant l'interface, une carte permet de se faire une idée de la distance des appareils localisés. La carte a une échelle relative à l'appareil le plus éloigné car le `BLE` à une portée d'émission d'environ 10 mètres, ce qui renderait indistinctibles les appareils proches si la carte couvrait toute la zone d'émission. Les autres appareils sont mis à l'échelle relativement par rapport au plus éloigné pour garder une représentation réaliste.

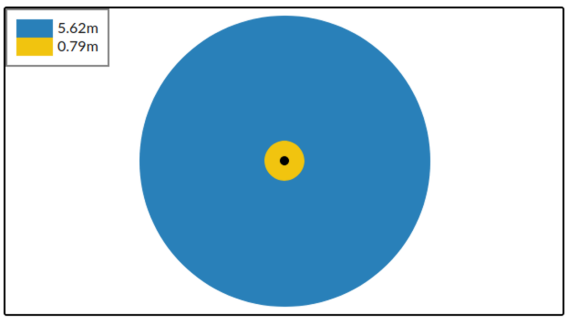

Figure 11: Carte des appareils localisés

Cette attaque concerne cependant seulement les appareils qui implémentent le standard *iBeacon*. D'une façon plus générale l'attaque est facilement mitigable en ne transmettant pas d'indication de distance dans le *GAP*, l'exposant uniquement dans le *GATT* une fois le *central* identifié. Même si les balises sont forcées d'inclure ses indications dans le *GAP*, ils ne sont pas concernés par la plupart des attaques car non connectables.

5.3 MITM

C'est une des attaques fournies par `Mirage`, implémentée dans le module `ble_mitm`. Cette attaque ne se base pas sur un *sniffer* pour intercepter du traffic et extraire des informations mais deux dongles `BLE CSR8510` pour occuper et usurper le *peripheral* ciblé puis profiter du manque d'authentification et un chiffrement peu robuste dans la méthode d'appairage `JustWorks`.
L'attaque requiert que le *peripheral* ciblé ne soit pas connecté pour pouvoir usurper son identité et s'annoncer à sa place. Même si il est auparavant possible de forcer une déconnexion des appareils via du brouillage (principe utilisé dans le *hijacking*), l'attaque échoue si une session (*LTK*) à été mise en place car celle-ci à besoin des paramètres échangés au moment de l'appairage pour casser le chiffrement.
L'attaque requiert également qu'un *central* se connecte au *peripheral* usurpé, après quoi il est en position de *man-in-the-middle*, redirigeant le traffic d'un appareil à l'autre en

utilisant les deux dongles `CSR8510`.
Cette attaque fonctionne autant sur des appareils qui connectés comme appairés avec la méthode `JustWorks` car le chiffrement peut être facilement brisé avec l'outil *Crackle*[37], implémenté par le module `ble_crack` dans `Mirage`, qui permet de trouver la clef de chiffrement à partir des information échangées lors de l'appairage.

Les attaques *MITM* sont une problématique répandue dans les communication donc nombres de contre-mesures sont disponibles dans les protocoles réseau. Le `BLE` fournie d'autres méthodes d'appairage permettant d'authentifier la connexion et certains appareils restreignent les droits ou refusent les connexions qui ne se conforment pas à leurs exigences.

5.4 Hijack

Le *hijack* peut être vu comme une version plus versatile du *MITM* puisque l'attaque peut fonctionner sur une connexion pré-établie. L'attaque à cependant une vocation uniquement offensive, elle ne permet pas la rétro-ingénierie par l'étude des communication échangées. Elle peut être vue comme un *MITM* ou l'on ne relaye pas les paquets entre *peripheral* et *central* mais forgeons nos propres paquets pour discuter avec le *central*.

Contrairement au *MITM* qui relaye les communications logiciellement au sein de `Mirage` entre deux `CSR8510`, cette attaque utilise un sniffer `micro:bit` pour se synchroniser puis détourner une communication.
La première phase de synchronisation à deux modes de fonctionnement suivant les besoins: le premier cible des appareils ayant l'intention de se connecter, `Mirage` intercepte les requêtes de connexion émises sur les canaux d'annonces pour se synchroniser avec la connexion fraichement créée. Le second essaye de retrouver les paramètres de connexions d'une connexion établie afin de s'y synchroniser.
Dans les deux cas il est nécessaire de connaître les paramètres échangés à la connexion pour se synchroniser (carte des canaux ainsi que le nombre et temps entre chaque saut), seule la méthode d'obtention diffère.

Je m'intéresse à la seconde méthode car je propose cette attaque sur des connexion établies, auparavant répertoriées via *sniffing*.
La récupération des paramètres de connexion est peu reliable due au saut de fréquence utilisé par le protocole pour éviter les interférences. La `micro:bit` change de canal fréquement pour augmenter ces chances d'en trouver un utilisé, mais il en va de même pour les connexions. C'est le jeu du chat et de la souris et ce n'est qu'une question de temps et de chance avant que la carte découvre tout les canaux utilisés. Le second point noir de cette attaque est qu'elle est inutile si la connexion est chiffrée: le chiffrement est dèjà établi et incassable vu que l'on se synchronise bien après la phase d'appairage. Il reste possible de détourner la connexion pour du déni de service, même si on sera incapable de communiquer avec l'autre appareil.

L'attaque est disponible nativement dans `Mirage` avec le module `ble_hijack`, fortement lié au module `ble_sniff` pour se synchroniser à une connexion établie. Ne disposant pas d'assez d'éléments pour casser le chiffrement d'une connexion établie, le détournement cible les appareils n'en utilisant pas comme des capteurs/actionneurs industriels ou domotique.

5.5 Tests et validation

Dû aux conditions exceptionnelles imposées par le confinement, je n'avait pas de produits `BLE` à disposition pour réaliser mes attaques. J'ai donc mis en place un réseau de test

prédictible et factice entre deux `CSR8510` à l'aide des modules imittant un *peripheral* (`ble_slave`) et son *central* (`ble_master`). Grâce aux scénarios `Mirage` j'ai pu modifier leurs fonctionnement pour mettre en place un scénario de test reproductible qui m'a grandement aidé pour identifier et corriger les *bugs* lors des développements. Il semble cependant complexe d'automatiser le test de toutes les attaques implementées puisque incertaines. Les tests unitaires de code donnent un résultat attendu et identique en un temps donné, maintenant tester le réseau est beaucoup plus incertain car instable. Les attaques peuvent ne jamais se déclencher ou des interférences peuvent interrompre le déroulement.
Des tests sur le *scan*, la localisation, l'usurpation (*MITM*) et le détournement (*hijack*) sont tout de même réalisables manuellement via le *CLI* `Mirage`.

Le *scan* requiert un dongle `CSR8510` émettant des annonces et minimum une connexion établie. Le scénario `MockSlave` (modification du module `ble_slave` à des fins de test) s'annonce jusqu'à ce qu'une connexion soit faite, permettant de tester le *scan* d'appareils alentours. Quant aux connexions établies, cela requiert une pair de `MockSlave` et `MockMaster` avec leurs `CSR8510` respectifs: les scénarios de test émettent des requêtes périodiquement une fois connectés pour générer un traffic.
Comme évoqué precedemment, le *scan* des connexions établie est incertain et peut durer indéfiniement. On pourrait mitiger ce problème avec la parrallélisation de `micro:bit`, chacune scannant une partie des 37 canaux de données. Il faudrait 37 `micro:bit` dans l'idéal pour les canaux de données et 3 pour ceux d'annonces. `Mirage` dispose d'ailleurs du balayage pour pallier à ce probleme de parrallélisation lors du scan d'appareils.

Les `MockSlave` sont programmés pour émettre `TxPower` dans leurs annonces pour permettre la localisation (calibrage précédemment relevé à 1 mètre du `CSR8510` puis intégré au *GAP*). On peut ainsi mesurer l'acuité des distances en les comparants à la realité. Après avoir fait plusieurs tests à des distances variants de moins d'un mètre jusqu'à 10 mètres, le *RSSI* est fortement affecté par la distance et les objets entre l'émetteur et le récepteur. À une distance de moins d'un mètre on trouve un résultat avec une précision de l'ordre de 30 centimètres et si l'on se place hors de la ligne de vue de l'émetteur, la distance calculée augmente car le *RSSI* diminue dû aux pertes. Un autre problème est le seuil de sensibilité, annoncé de -30 a -90dBm par le constructeur, de la puce `nRF51`: celui-ci varie entre -45dBm au plus proche jusqu'à -70dBm à la distance maximale hors de la ligne de vue (après quoi le signal est considéré perdu). La précision des résultats aux bornes de ces valeurs est faussée car un appareil se trouvant à 10 comme à 45 centimètres aura un *RSSI* de -45dBm, idem pour un appareil proche de la distance maximale de réception.
Une amélioration intéressante serait l'ajout de points de relevés pour permettre la triangulation à partir d'un seul récepteur. Il suffirait dès lors de se déplacer entre chaque relevé pour avoir une idée des appareils localisés dans l'espace, et plus seulement une distance. Bien sur il est possible d'ajouter 2 autres `RaspberryPi` avec leurs `micro:bit` pour permettre une triangulation à partir d'un seul relevé, en admettant qu'elles soient disposées correctement.

L'usurpation demande 2 `CSR8510` pour mener l'attaque et 2 pour la tester ainsi qu'un ordre précis dans l'exécution. Un dongle `CSR8510` écoute les annonces dans l'attente du `MockSlave`. Une fois trouvé, il s'y connecte, le clone et maintient la connexion pour stopper l'émission d'annonces. Un second `CSR8510` usurpe le `MockSlave` précédemment cloné et s'annonce en tant que tel dans l'attente du `MockMaster`. Dès lors que le `MockMaster` se connecte à l'usurpateur, un scénario modifie les paquets échangés entre `MockSlave` et `MockMaster`, rendant vérifiable le fonctionnement de l'attaque depuis leurs *CLI* respectifs.

Le détournement de connexion est testé contre une connexion factice sans appairage entre un `MockSlave` et un `MockMaster`. Avec une seule `micro:bit`, le *sniffing* de la connexion était déjà très long pour espérer tomber sur les canaux utilisés par une seule connexion. La récuperation des paramètres de connexion n'a jamais réussie, même avec un long délai. `BTLEJack` et `Mirage` mettent d'ailleurs en garde sur la reliabilité de cette attaque et est plus considérée comme expérimentale qu'utile pour le *pentest*. Il semble obligatoire de parralléliser l'analyse des canaux de données afin d'augmenter leur couverture pour espérer conduire l'attaque voir réduire le temps nécessaire à cette phase.

6 Conclusion

Les mécanismes du `BLE` évoluent pour rester viable dans son marché d'objets intelligents facilitant l'interaction homme-machine. Il dispose des mécanismes nécessaires pour mettre en place un niveau de sécurité acceptable pour les standards contemporains. La recherche d'interoperabilité engendre encore des vulnérabilités, mais avec la prochain dépréciation du protocole 4.0, et donc des connexions *legacy*, l'écoute passive ne devrait plus être une menace car la version 4.2 met en place les connexion sécurisées basées sur *Diffie-Hellman*. La plupart des vulnérabilités restantes seront de l'ordre utilisateur plutôt que d'un défaut de conception du protocole. Il n'empêche que beaucoup des constructeurs n'utilisent toujours pas les mécanismes fournis par le `BLE` mais leur solution personnalisée.

Le projet s'est plus rapproché d'une étude de recherche et développement plutot que d'un projet de stage que j'ai eu l'occasion de retrouvé en entreprise par le passé. Le sujet à demandé un effort de documentation et de compréhension (etude du protocole BLE, des attaques existantes, du materiel et des outils disponibles pour l'analyser) plus important que le développement (*front* & *back-end* puis intégration du framework `Mirage` avec ajout d'un nouveau module).
Ce n'est cependant pas pour me déplaire puisque j'éprouve un certain intérêt pour la compréhension et l'étude des systèmes dans les moindres détails, je suis également toujours plus à l'aise en travaillant sur un système compris et maîtrisé, ce qui me permet de voir l'étendue des possibilités, plutôt que de me jeter directement dans le développement.

Ce projet m'a permit de me plonger totalement dans le réseau, d'en analyser la structure/-couches et jouer avec les outils de dissection (`wireshark`, `scapy`), comblant un certain manque dans les années d'étude précédentes.
Le seul bémol reste les conditions exceptionnelles dans lequel il a été réalisé obligeant de travailler avec une connexion factice plutôt que des appareils commercialisés, ce qui aurait permit une consolidation des observations avancées.

[1]. https://zigbeealliance.org/.

[2]. https://www.z-wave.com/.

[3]. https://www.thisisant.com/.

[4]. https://www.threadgroup.org/.

[5]. https://www.accton.com/Technology-Brief/ble-beacons-and-location-based-services/.

[6]. https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/.

[7]. https://www.bluetooth.com/specifications/gatt/services/.

[8]. https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2017/02/10/bluetooth_advertisin-hGsf.

[9]. https://www.bluetooth.com/blog/bluetooth-pairing-part-1-pairing-feature-exchange/.

[10]. https://www.bluetooth.com/blog/bluetooth-pairing-part-2-key-generation-methods/.

[11]. https://www.bluetooth.com/specifications/gatt/services/.

[12]. https://fr.mathworks.com/help/comm/examples/modeling-of-ble-devices-with-heart-rate-profile.html.

[13]. https://www.blackhat.com/.

[14]. https://www.defcon.org/.

[15]. https://www.sstic.org/.

[16]. https://www.wireshark.org/.

[17]. https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF51-Dongle.

[18]. http://www.ti.com/tool/CC2540EMK-USB.

[19]. https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Sniffer-for-Bluetooth-LE.

[20]. http://www.ti.com/tool/PACKET-SNIFFER.

[21]. https://www.adafruit.com/product/2269.

[22]. https://github.com/virtualabs/btlejack.

[23]. https://microbit.org/.

[24]. https://homepages.laas.fr/rcayre/mirage-documentation/index.html.

[25]. https://homepages.laas.fr/rcayre/mirage-documentation/modules.html.

[26]. https://github.com/noble/noble.

[27]. https://github.com/noble/bleno.

[28]. Cayre, R.; Roux, J.; Alata, E.; Nicomette, V. and Auriol, G.: Mirage : Un framework offensif pour l'audit du bluetooth low energy (2019), https://hal.archives-ouvertes.fr/hal-02268774.

[29]. https://github.com/mikeryan/PyBT.

[30]. https://homepages.laas.fr/rcayre/mirage-documentation/devices.html.

[31]. https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/.

[32]. https://www.qualcomm.com/products/csr8510.

[33]. https://homepages.laas.fr/rcayre/mirage-documentation/devices.html#btlejack-device.

[34]. https://www.waveshare.com/ble400.htm.

[35]. https://www.waveshare.com/nrf51822-eval-kit.htm.

[36]. http://docs.yottabuild.org/.

[37]. https://github.com/mikeryan/crackle.

[38]. SIG, B.: Bluetooth spécifications version 4.2 (2014), https://www.bluetooth.com/specifications/archived-specifications/.

[39]. Jasek, S.: Gattacking bluetooth smart devices, http://gattack.io/whitepaper.pdf.

[40]. Ryan, M.: Bluetooth: With low energy comes low security, https://www.usenix.org/system/files/conference/woot13/woot13-ryan.pdf.