Mirage : un framework offensif pour l'audit du Bluetooth Low Energy

Romain Cayre[1,2], Jonathan Roux[1,3], Eric Alata[1,3], Vincent Nicomette[1,3] et Guillaume Auriol[1,3]
prenom.nom@laas.fr

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] APSYS.LAB, APSYS
[3] Université de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Résumé. Nous assistons aujourd'hui à une mutation profonde de l'industrie informatique : des systèmes connectés d'un nouveau genre, désignés sous le terme d'objets connectés, apparaissent au sein des foyers et des entreprises, où leur usage se généralise progressivement. Dans ce contexte, la sécurisation de ces objets devient un enjeu majeur. Pourtant, l'audit d'un système connecté reste aujourd'hui une tâche complexe, nécessitant l'utilisation de nombreux outils hétérogènes, tant matériels que logiciels, souvent incompatibles entre eux et à la mise en œuvre parfois complexe. Nous présentons dans cet article le framework *Mirage* développé en Python et destiné à l'audit des technologies sans fil fréquemment utilisées par les objets connectés. Nous introduisons notamment les modules d'analyse et d'attaque développés dans le cadre de l'évaluation d'objets connectés utilisant la technologie sans fil *Bluetooth Low Energy*.

1 Introduction

1.1 Problématique

Les systèmes d'information sont actuellement en pleine mutation : de nombreux dispositifs du quotidien, désignés sous le terme générique d'*objets connectés*, sont dotés de nouvelles fonctionnalités « intelligentes » et s'inscrivent dans le cadre d'une interconnexion croissante. Cette évolution majeure, dont la vocation est d'étendre au monde physique le réseau Internet, est appelée *Internet des Objets*. 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.

Cette évolution se produit dans un contexte particulier : en effet, face au développement de ces nouveaux marchés, de nombreux protocoles de communication sans fil ont été propulsés sur le devant de la scène, sans que l'un d'entre eux ne se soit réellement démarqué et imposé. Ces

protocoles cherchent à répondre aux problématiques techniques liées aux objets connectés, comme l'économie d'énergie et la mobilité. *ANT+*, *Zigbee*, la variante « basse consommation » du Bluetooth (nommée *Bluetooth Low Energy*) se livrent ainsi une concurrence féroce, au fur et à mesure de leur déploiement au sein de l'*Internet des Objets*, et accumulent des fonctionnalités pour convaincre l'industrie, reléguant la sécurité au second plan. L'hétérogénéité de ces technologies et leur expansion rapide sont problématiques, car elles contribuent à augmenter la surface d'attaque des systèmes concernés, exposant par là-même les systèmes d'information environnants, tout en compliquant le travail de l'analyste.

Dans ce contexte d'expansion rapide, il devient indispensable de disposer d'outils fiables et efficaces et de méthodologies pertinentes afin d'auditer la sécurité de ces systèmes d'un nouveau genre. Nous nous concentrons ici sur l'audit des objets connectés utilisant le *Bluetooth Low Energy*, une technologie de communication sans fil dérivée du *Bluetooth*, particulièrement populaire pour le développement d'objets connectés en raison de sa faible consommation énergétique et de son large déploiement au sein des *smartphones* et tablettes.

De nombreux outils, logiciels comme matériels, ont été produits ces dernières années afin d'évaluer la sécurité des systèmes communiquants via *Bluetooth Low Energy*, et nous disposons aujourd'hui d'une vision assez complète des attaques et des risques inhérents à cette technologie. Cependant, il reste aujourd'hui particulièrement complexe d'auditer de façon satisfaisante ce type d'équipements, pour de nombreuses raisons.

Au niveau des composants matériels utilisés dans le cadre de la sécurité offensive, des systèmes et composants matériels *RF* variés sont utilisés, disposant chacun de leurs caractéristiques et de leurs APIs propres, impliquant de nombreux développements coûteux en temps et peu intéressants du point de vue de l'analyste. En outre, les problématiques liées à l'étude de la couche physique du *Bluetooth Low Energy*, dont le mécanisme de saut de fréquence (ou *channel hopping*) empêche l'utilisation de matériels génériques de type *Software Defined Radio* [11], ont amené au développement d'outils matériels spécifiques [4, 16], dont l'utilisation est parfois complexe.

De manière symétrique, l'hétérogénéité des solutions matérielles pose également de nombreux problèmes pour le développement logiciel des outils d'analyse de vulnérabilités. Devant cette diversité, les auditeurs utilisent des bibliothèques non développées dans une perspective offensive et souvent de haut niveau, donc peu adaptées aux enjeux de la sécurité offensive, ou développent des bibliothèques « maison » peu robustes

et peu modulaires. Un exemple frappant de cette problématique est le développement des deux outils de *Man In The Middle* pour le Bluetooth Low Energy : *GATTacker* [10] et *BTLEjuice* [3]. Ces derniers étant tous deux basés sur les bibliothèques nodeJS *noble* et *bleno* disponibles au moment du développement de ces outils, ils souffrent des mêmes limitations importantes du fait que ces bibliothèques ne permettent pas de gérer en parallèle sur le même système d'exploitation deux *dongles* Bluetooth, l'un en mode *Slave* et l'autre en mode *Master*. Pour contourner cette limitation, ils ont dû respectivement modifier les bibliothèques correspondantes ou mettre en place un système lourd de communication entre deux instances à l'aide de *WebSockets* afin que chaque *dongle* soit géré depuis un système d'exploitation différent, par exemple depuis une machine virtuelle.

Cette situation génère beaucoup de développements inutiles, et les codes, devant composer avec de nombreuses contraintes et limitations techniques, se complexifient à outrance. La **simplicité**, la **réutilisabilité** et la **modularité**, piliers du développement logiciel moderne, souffrent de cet état de fait.

1.2 Objectifs

Ces différents constats ont motivé le développement d'un framework d'audit, visant les technologies sans fil utilisées dans la conception d'objets connectés et notamment le *Bluetooth Low Energy*. L'objectif principal est de fournir un cadre de développement robuste et modulaire pour le test de vulnérabilités, qui soit capable de s'interfacer avec tout type d'outils matériels d'exploitation, tout en permettant à l'auditeur de se concentrer sur la logique du test (et non le fonctionnement ou les limitations de bibliothèques peu adaptées), et en lui offrant la possibilité d'intervenir au niveau des couches protocolaires basses.

Trois objectifs centraux ont motivé et guidé le développement de ce framework offensif d'attaque nommé *Mirage*.

Une approche unifiée. Lors de l'audit de sécurité des protocoles de communication utilisés par un objet connecté, il est courant de devoir utiliser plusieurs outils logiciels différents en parallèle. Cet état de fait pose de nombreux problèmes : les formats de fichiers ne sont pas forcément compatibles entre eux, chaque outil dispose bien souvent de sa propre API, de ses propres contraintes, etc.

Ces problèmes amènent l'auditeur à assimiler un grand nombre d'informations techniques, qui ne sont pas directement en lien avec la logique de l'attaque, ainsi qu'à installer de nombreux outils et bibliothèques.

La première contrainte qui a guidé la conception du framework fut donc celle de mettre en place une approche unifiée. Chaque outil proposé, s'il dispose de sa propre logique d'utilisation, est manipulable par une *API* similaire, et s'interface avec les autres modules de la même façon. Pour le développeur, il est également permis de manipuler les différentes technologies sans fil au travers d'API respectant les mêmes grands principes dans tout le framework. De plus, la structure même du framework impose de respecter le même type d'approche lors de la conception d'un module, tout en laissant évidemment une certaine liberté au développeur. Le respect de ce type de formalisme permet ainsi de développer des outils dont l'utilisation et l'interfaçage sont grandement facilités.

Une approche modulaire. Dans cette même logique, l'approche adoptée lors du développement du framework est celle de créer un système souple et modulaire. En effet, les attaques ciblant une technologie particulière peuvent contenir des éléments communs : si on prend l'exemple du *fuzzing* du serveur *GATT* d'un périphérique *Bluetooth Low Energy* et du clonage de ce même périphérique, un certain nombre d'éléments sont similaires. Il faudra scanner l'environnement pour détecter la présence de l'objet, se connecter et énumérer les différents services et caractéristiques présents sur l'objet, avant de mettre en place un comportement spécifique (exporter la connaissance acquise ou lancer une tentative de *fuzzing*).

Pour éviter la présence de codes redondants et faciliter la maintenance du code, le framework est conçu pour faciliter le découpage des attaques en modules logiciels fonctionnels. Ainsi, dans le cas précédemment évoqué, un module permet de se connecter à l'objet, un module permet le scan de l'environnement, un autre énumère les services disponibles, etc. Il est dès lors possible de combiner ces différents modules en une attaque complexe grâce à un mécanisme de chaînage (dont le fonctionnement est similaire à celui de l'opérateur «|» sous Unix), voire de les intégrer dans un module plus général si le comportement est trop complexe.

Afin de conserver une certaine souplesse dans l'utilisation, les modules sont paramétrables par l'intermédiaire d'arguments d'entrée. Chaque module disposant également de la possibilité de renvoyer une ou plusieurs valeurs de sortie, le rôle de l'opérateur de chaînage est d'exécuter successivement chaque module spécifié, en définissant les valeurs potentielles de sortie d'un module comme paramètres d'entrée du module suivant.

Une approche bas niveau. Comme souligné précédemment, les outils logiciels utilisés dans le développement des attaques sont souvent peu

adaptés car trop haut niveau, et leurs structures et leurs limitations peuvent constituer des freins au bon développement d'une attaque.

Une volonté assumée lors de la conception du framework était de permettre de travailler sur les couches inférieures des protocoles sans fil, afin de pouvoir développer tout type d'attaque en limitant le moins possible le développeur par des contraintes logicielles. Ainsi, les modules matériels *RF* supportés par le framework ont été sélectionnés pour permettre de manipuler les couches basses des protocoles, et chaque protocole implémenté a été défini au niveau le plus bas accessible logiciellement. La pertinence de cette approche s'est manifestée à de nombreuses reprises lors du développement d'attaques pour le framework. Ainsi, dans le cas du *Bluetooth Low Energy*, travailler directement au niveau des paquets *HCI* et non avec une bibliothèque tierce a permis de développer des attaques comme le *Man In The Middle* sans les limitations imposées aux autres outils similaires, *BTLEJuice* [3] et *GATTacker* [10].

Dans la section 2, nous commençons par établir un bref panorama de la sécurité offensive pour le *Bluetooth Low Energy*. Nous présentons notamment les principales caractéristiques techniques de la technologie, puis nous dressons un état de l'art détaillé de la sécurité offensive pour ce type de systèmes. Ensuite, nous présentons en section 3 le fonctionnement général du framework *Mirage* ainsi que quelques éléments d'architecture, puis quelques modules d'attaque pertinents. La section 4 détaille l'audit d'une ampoule connectée par l'intermédiaire des modules précédemment décrits. Enfin, la section 5 conclut cet article.

2 Panorama de la sécurité offensive pour le Bluetooth Low Energy

2.1 Le Bluetooth Low Energy

Le *Bluetooth Low Energy* (aussi appelé *Bluetooth Smart*) est une évolution du protocole de transmission sans fil *Bluetooth* à destination des systèmes embarqués, introduit dans la norme 4.0 de la spécification du Bluetooth [15]. Développé en parallèle de la technologie Bluetooth, il introduit un certain nombre de fonctionnalités destinées à diminuer la consommation énergétique des systèmes l'utilisant, en faisant un choix particulièrement indiqué pour les objets connectés.

La couche physique. Le *Bluetooth Low Energy* (ou *BLE*) utilise une couche physique similaire à celle du *Bluetooth*, bien qu'incompatible en

raison de l'utilisation de modulations différentes. La modulation utilisée par le *BLE* est une modulation à déplacement de fréquence utilisant un filtre gaussien pour limiter la largeur spectrale (*Gaussian Frequency-Shift Keying*). Fonctionnant dans la bande de fréquences de 2.4GHz à 2.5GHz, le *BLE* peut utiliser jusqu'à 40 canaux de communication (bien qu'un sous ensemble de ces canaux puisse être utilisé), dont trois sont réservés à la diffusion de messages en *broadcast*, appelés *advertisements*. La répartition de ces canaux dans le spectre radio-fréquence est décrite par la figure 1.

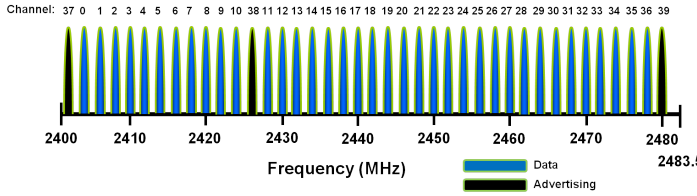

Fig. 1. Les canaux de communication du Bluetooth Low Energy.

La couche physique du *BLE* est basée sur un mécanisme de saut de fréquence (appelé *channel hopping*), destiné à éviter les interférences avec d'autres technologies sans fil utilisant des fréquences similaires. Ainsi, au cours d'une connexion, les systèmes changent fréquemment de canal selon un algorithme dépendant de trois paramètres négociés à la connexion :

- **le Channel Map**, définissant l'ensemble des canaux de communication utilisé par la connexion ;
- **le Hop Increment**, indiquant l'incrément pour le passage d'un canal à un autre ;
- **le Hop Interval**, indiquant la durée entre deux sauts de fréquence.

Le passage du canal *n* au canal *n+1* est défini par la formule suivante : $channel_{n+1} \equiv (channel_n + hopIncrement) \mod 37$

Le changement de canal est réalisé à intervalle régulier, cette durée étant paramétrée par le *Hop Interval* et pouvant être calculée à l'aide de la formule suivante : $\Delta t = 1.25ms \times \text{hopInterval}$

La synchronisation lors d'une communication est assurée grâce à un mécanisme d'échanges de paquets vides. Un algorithme de contrôle de redondance cyclique (abrégé *CRC*) permet de vérifier la validité des données reçues, et se paramètre à partir d'une valeur nommée *CRCInit*, elle aussi transmise lors de l'établissement de la connexion. Enfin, une adresse d'accès (ou *Access Address*) de 32 bits permet d'identifier la connexion courante.

Description protocolaire du Bluetooth Low Energy. Tout comme le *Bluetooth*, le *Bluetooth Low Energy* utilise une pile protocolaire (représentée en figure 2) séparée en deux parties : la partie *Host* et la partie *Controller*. Ces deux parties communiquent au travers d'une interface série appelée *Host Controller Interface* (généralement abrégée HCI).

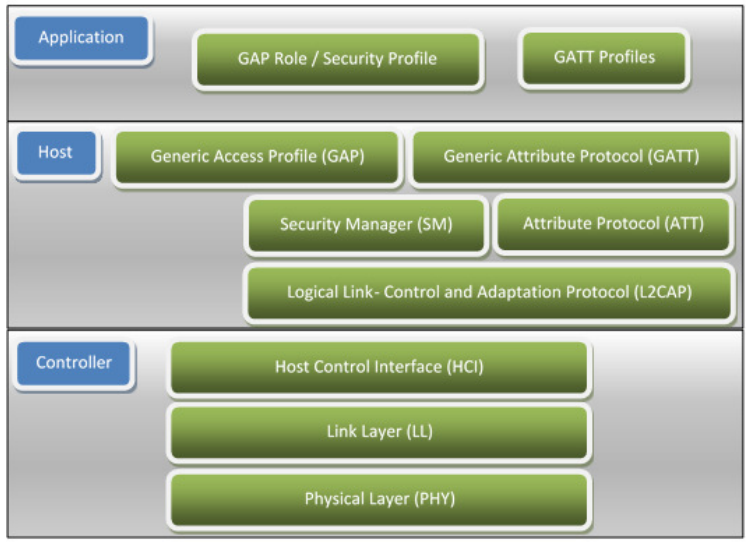

Fig. 2. Modèle en couches du Bluetooth Low Energy.

La partie *Controller* gère les couches protocolaires les plus basses, notamment la couche physique et la couche liaison (*Link Layer*). Cette couche liaison présente une architecture de type Maître / Esclave. Elle définit ainsi deux rôles :

— **le Master** est responsable de l'établissement et de la synchronisation de la connexion. C'est notamment lui qui est en charge de transmettre les paramètres nécessaires à l'algorithme de saut de fréquence et la valeur d'initialisation de l'algorithme de CRC ;

— **le Slave** est le système sur lequel un *Master* peut se connecter. En l'absence de connexion, il peut signaler sa présence par l'intermédiaire du mécanisme d'*advertisement*, et, lorsqu'une connexion est établie, il paramètre l'algorithme de saut de fréquence et de contrôle d'intégrité en fonction des informations transmises par le *Master*. Les objets connectés fonctionnent généralement en mode *Slave*.

La partie *Host*, quant à elle, gère les couches supérieures du protocole *BLE*. Contrairement au *Bluetooth* qui présente de très nombreuses couches en fonction des applications souhaitées, le *BLE* ne présente qu'un nombre limité de couches.

La couche dite *L2CAP* est une couche de transport des données. Les autres couches gérées par la partie *Host* utilisent toutes cette couche afin de communiquer avec le *Controller*.

La couche *Generic Access Profile* (ou *GAP*) définit quatre rôles possibles pour un périphérique *BLE* : *Broadcaster*, *Scanner*, *Peripheral* ou *Central*. En effet, le *Bluetooth Low Energy* permet deux types de mécanismes pour la diffusion de données [9].

Le mode d'Advertising est utilisé pour diffuser des données en *broadcast*, à tous les équipements de l'environnement. Les paquets sont alors émis sur les trois canaux d'*advertising* réservés à cet effet, et sont accessibles à tout équipement en écoute sur l'un de ces canaux. Ce type de communication peut être utilisé comme un mécanisme de diffusion de données à part entière, ou servir à signaler la présence d'un équipement *Slave* en attente d'une connexion. Dans ce dernier cas, les paquets contiennent généralement des informations utiles à l'identification du périphérique, comme son adresse BD (un identifiant unique sur 48 bits) ou son nom. Il est possible de découvrir la présence d'un équipement émettant des *advertisements* par l'intermédiaire d'un scan actif ou passif.

Le mode Connecté établit une communication bidirectionnelle entre un périphérique de type *Master* et un périphérique de type *Slave*. Dans ce mode, le *Master* émet une requête de connexion (appelée *CONNECT_REQ*) à destination de l'objet esclave sur l'un des trois canaux d'*advertising*, en signalant les paramètres de l'algorithme de saut de fréquence précédemment mentionnés. Ils vont alors établir une communication en suivant le même motif de saut de fréquence, permettant alors l'utilisation des couches applicatives *Security Manager* (en charge de la sécurisation de la communication et de l'échange des paramètres de chiffrement) et des couches *ATT* et *GATT*.

Un périphérique uniquement capable d'émettre des *advertisements* correspond au rôle *Broadcaster*, tandis qu'un périphérique uniquement capable d'écouter ces canaux est dit *Scanner*. Le rôle *Peripheral* correspond quant à lui à un objet capable d'utiliser le mécanisme d'*advertisement* et d'accepter des connexions d'un *Master*, tandis que le rôle *Central* peut établir des connexions.

Les couches ATT et GATT. Après établissement d'une connexion, les couches *Attribute Protocol* (ou *ATT*) et *Generic Attribute Protocol* (ou *GATT*) offrent un mécanisme générique d'échange de données applicatives [9]. Contrairement au *Bluetooth*, où chaque type d'application dispose d'une couche applicative dédiée, le *BLE* présente une couche applicative unique, au fonctionnement générique.

Un périphérique capable de supporter le rôle *Peripheral* fonctionne en tant que serveur *ATT* : il se comporte alors comme une base de données d'*attributs*. Chacun de ces *attributs* dispose d'un identifiant unique (un entier non signé de 16 bits compris entre 0x0000 et 0xFFFF, appelé *handle*), d'un *type* (identifié par un *UUID*[4] sur 16 ou 128 bits) et d'une valeur (de taille variable).

Un périphérique correspondant au rôle *Central*, après avoir établi une connexion avec un *Peripheral*, se comporte comme un client *ATT*. Il dispose alors d'un certain nombre de méthodes pour manipuler les attributs du serveur. Ces méthodes permettent :

- **des accès en écriture** : la modification d'un attribut est réalisée par l'intermédiaire des méthodes *Write Command* (n'attendant pas de réponse de la part du serveur) et *Write Request* (nécessitant un acquittement de la part du serveur) ;
- **des accès en lecture** : la lecture de la valeur d'un attribut identifié par son *handle* est effectuée par l'intermédiaire de la méthode *Read Request*, qui génère une *Read Response* de la part du serveur, tandis que des méthodes complémentaires permettent de lire la valeur d'un attribut en fonction de son type (*Read By Type Request* et *Read By Group Type Request*) ou de récupérer ses caractéristiques de type et son *handle* (*Find Information Request*).

Le serveur est, quant à lui, capable de communiquer avec le client soit par l'intermédiaire de réponses à ses requêtes, soit grâce à un mécanisme de notification (*Handle Value Notification*).

La couche *GATT*, quant à elle, est une spécialisation de la couche *ATT*. Basée sur cette dernière, elle permet de hiérarchiser les attributs en un ensemble de *services* (primaires ou secondaires), eux-mêmes composés de caractéristiques (*characteristics*). Enfin, des informations supplémentaires (comme une description ou un rôle) peuvent être apportées aux caractéristiques par l'intermédiaire de descripteurs (*descriptors*). Cette structure permet de mettre en place des profils harmonisés entre différents objets au comportement similaire : ainsi, deux périphériques capables de se comporter comme des moniteurs de rythme cardiaque présenteront un

4. *Universal Unique Identifier*, soit un « Identifiant Universel Unique »

service commun *Heart Rate Service* (illustré en figure 3), contenant des caractéristiques communes (comme *Heart Rate Measurement* ou *Body Sensor Location*).

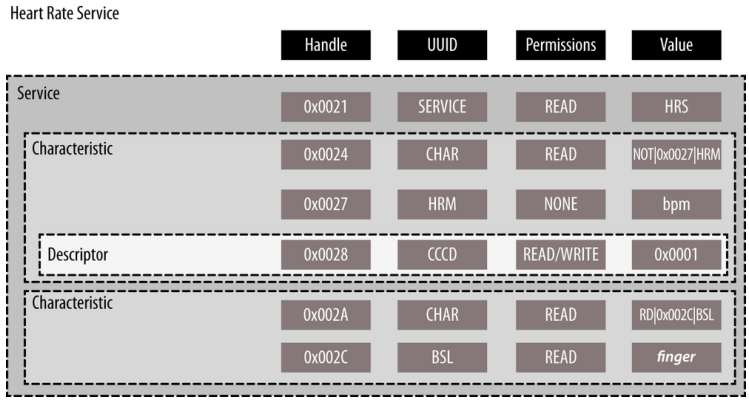

Fig. 3. Exemple d'un service GATT standardisé.

Ces définitions de services et de caractéristiques sont nommées *Profiles*, et l'organisme Bluetooth SIG propose ainsi des profils standards pour de nombreuses applications différentes [9, 15]. Cependant, cela n'empêche pas la création de comportements spécialisés, et l'usage de ces profils standards n'est évidemment pas obligatoire.

La découverte des données *GATT* est implémentée au travers d'algorithmes simples, utilisant les méthodes de la couche *ATT*.

2.2 Etat de l'art

Composants matériels. Plusieurs composants matériels ont été développés et utilisés afin de permettre l'analyse et la génération de trafic *BLE* dans un cadre de sécurité offensive [4, 16]. Les composants les plus couramment utilisés sont les *dongles* Bluetooth standard implémentant (au minimum) la spécification Bluetooth 4.0 [15], permettant notamment de jouer le rôle d'un composant *Bluetooth Low Energy* légitime (de type *Master* ou de type *Slave*), contrôlable via une interface série de type *HCI*. Un certain nombre de ces *dongles* ont été dotés par leurs fabricants de fonctionnalités supplémentaires intéressantes du point de vue offensif, notamment la possibilité de modifier l'adresse BD du *dongle*, ce qui permet d'usurper l'adresse d'équipements légitimes (c'est notamment le cas des *dongles Cambridge Silicon Radio*).

Au niveau de l'analyse du trafic réseau, et notamment dans une optique de *sniffing*, il a été nécessaire de développer des composants matériels spécifiques : en effet, les spécificités de l'algorithme de saut de fréquences du *BLE* imposent l'utilisation de composants radios capables de suivre les déplacements de fréquences de la communication. Cette contrainte exclut de fait toute approche basée sur la technologie de radio logicielle, appelée *Software Defined Radio* (en tout cas celle disponible pour un coût inférieur à 300 euros), en raison des délais imposés par la communication via USB et de l'implémentation logicielle des protocoles. Le projet *Ubertooth* [16] a été conçu dans cette optique, et a abouti au développement de deux itérations successives d'un matériel de sniffing *Bluetooth* et *Bluetooth Low Energy* (*l'Ubertooth Zero* et *l'Ubertooth One*).

De même, plusieurs *sniffers* ont été développés à base de microcontrôleur Nordic Semiconductor de la série nRF51 : la compagnie Adafruit a proposé un sniffer [17] (le *Bluefruit LE Sniffer*) capable de suivre des communications BLE depuis leur initiation, tandis que Damien Cauquil a implémenté sur un *Micro:bit* [4] (un matériel développé par la BBC à des fins pédagogiques, embarquant un microcontrôleur nRF51) le firmware de *BTLEJack* [5], un outil capable de sniffer des communications depuis leur initiation ou de retrouver les paramètres d'une communication existante. Il est à noter que cet outil (tout comme l'*Ubertooth* [16]) implémente un certain nombre de mécanismes de déni de services (notamment le *Jamming*, ou brouillage, de connexions) et permet l'*usurpation* de la communication.

Approches passives. Les principaux outils logiciels d'attaques passives du *Bluetooth Low Energy* se sont concentrés sur le *sniffing* de communications BLE. Il s'agit d'un problème complexe, dont la principale difficulté résulte du mécanisme de saut de fréquence. En effet, celui-ci étant particulièrement rapide, et la bande de fréquence concernée particulièrement large, il semble exclu de surveiller l'ensemble des 39 canaux de communication possibles, sauf à utiliser du matériel particulièrement onéreux.

La première stratégie possible consiste à capturer la communication dès son établissement, et notamment le paquet *CONNECT_REQ*, qui contient les différents paramètres déterminés dans le saut de fréquence et dans le calcul du *CRC*.

Dès lors, il suffit d'utiliser le même algorithme pour suivre la communication en même temps que ses acteurs légitimes. Cependant, ce paquet peut être émis sur n'importe lequel des trois canaux d'*advertising* : ceux-ci étant éloignés les uns des autres dans le spectre radio, il est possible de

n'être pas en écoute sur le bon canal au moment de l'établissement de la connexion (la probabilité de succès étant alors de 1/3).

Une seconde approche, proposée par Mike Ryan dans le *whitepaper* de la conférence *With Low Energy Comes Low Security* [13], consiste à déduire les paramètres de connexion d'une connexion déjà établie. Supposant que l'ensemble des 37 canaux de données étaient systématiquement utilisés, il propose l'algorithme suivant pour retrouver ces paramètres :

Initialiser : Mettre le sniffer en écoute sur un canal de donnée arbitraire.

Récupérer l'access Address : Identifier l'access Address de la connexion (32 bits) en cherchant des paquets vides. Ceux-ci ayant un format prédictible et étant fréquemment transmis (en effet, les paquets vides sont échangés lorsqu'il n'y a pas de transmission entre le master et le slave), ils permettent de déduire des adresses d'accès candidates, qui seront ensuite validées ou infirmées selon leur fréquence d'apparition.

Identifier la valeur de CRCInit : En effet, la suite de l'algorithme de récupération des paramètres ne permettant pas de travailler avec des faux positifs, il faut être capable de déterminer si un paquet reçu est valide ou non. Pour retrouver cette valeur, Mike Ryan propose d'utiliser le fait que ces valeurs de CRCInit sont calculées à l'aide d'un registre à décalage linéaire gauche, et sont donc réversibles. Il déduit d'un registre à décalage inversé des valeurs de CRCInit candidates, ce qui lui permet ensuite de réaliser des calculs et de sélectionner la bonne valeur de CRCInit.

Identifier le Hop Interval : en constatant l'écart temporel entre deux paquets reçus sur le même canal, on peut déduire que le temps écoulé est celui d'un cycle complet du *Channel Map*. Toujours en supposant que les 37 canaux de données sont utilisés, on peut donc déduire le *Hop Interval* en appliquant la formule suivante :

$$\text{hopInterval} = \frac{\Delta t}{37 \times 1.25ms}$$

Identifier le Hop Increment : finalement, en mesurant le temps entre un premier paquet arrivé sur un canal de communication arbitraire *0* et le paquet suivant sur le canal consécutif *1*, on peut retrouver le nombre de sauts réalisés entre le premier et le second paquet :

$$\text{channelsHopped} = \frac{\Delta t}{\text{hopInterval} \times 1.25ms}$$

Il est alors possible de retrouver le *Hop Increment* en résolvant l'équivalence suivante :

$$0 + \text{hopIncrement} \times \text{channelsHopped} \equiv 1 \mod 37$$

Soit :

$$\text{hopIncrement} \equiv \text{channelsHopped}^{-1} \mod 37$$

On peut résoudre cette équivalence grâce à l'application du petit théorème de Fermat, en utilisant une *Look Up Table*.

À l'issue de cet algorithme, on dispose alors de tous les paramètres nécessaires pour se synchroniser avec l'algorithme de saut de fréquence. Mike Ryan a implémenté son algorithme sur un des sniffers matériels que nous avons présentés auparavant, nommé *Ubertooth One* [16].

Cependant, une hypothèse de Mike Ryan, pourtant valide en pratique lorsqu'il a défini son algorithme, est apparue comme une limitation conséquente à cette approche : en effet, le *Channel Map* n'est pas forcément composé de l'ensemble des 37 canaux, rendant l'algorithme de récupération de paramètres inefficace.

De plus, certains canaux peuvent être réutilisés pour conserver un nombre de 37 canaux au total. Damien Cauquil a proposé une amélioration de l'algorithme initialement proposé par Mike Ryan lors de la 25^{e} édition de la conférence *DEFCON*, qu'il a implémenté avec succès sur un *Micro:bit* détourné de son utilisation pédagogique initiale [4].

Il propose de tout d'abord déterminer le *Channel Map* en écoutant successivement sur les 37 canaux possibles à la recherche de paquets valides : cette étape intervient donc logiquement après la récupération de l'*Access Address* et du *CRCInit*. Cette phase peut durer jusqu'à 4×37 secondes, rendant cette stratégie particulièrement lente. Le *Hop Interval* est retrouvé grâce à la même approche que Mike Ryan, mais le *Hop Increment* est, quant à lui, déduit selon la même procédure en prenant soin de sélectionner des canaux de communication n'apparaissant qu'une seule fois dans le *Channel Map* précedemment déterminé.

Si les approches de Mike Ryan [13] et Damien Cauquil [4] sont intéressantes et fonctionnelles, on constate cependant qu'elles restent difficiles à mettre en œuvre et présentent des inconvénients conséquents. Les délais importants impliqués par l'approche exhaustive de Damien Cauquil pour retrouver le *Channel Map* la rendent difficilement applicable pour la supervision de connexions courtes, pourtant assez fréquentes avec cette technologie. Cela implique également un certain nombre de paquets qui ne

seront pas sniffés lors de la récupération des paramètres de l'algorithme de saut de fréquence.

La question du chiffrement de ces communications se pose également : en effet, la spécification du *Bluetooth Low Energy* [15] propose des mécanismes de chiffrement des données. Si Mike Ryan a proposé un outil hors ligne, nommé *crackle* [14], capable dans certaines conditions de récupérer la clé de chiffrement et de déchiffrer le trafic, il est à noter que celui-ci se base sur l'étude des paquets émis lors de la négociation de la couche *Security Manager*, initiée en début de connexion, et est donc conditionné à la présence de ces paquets : seule une connexion sniffée depuis son initiation permet d'assurer le succès de cette attaque.

Approches actives. De nombreux travaux ont permis de manipuler le *Bluetooth Low Energy* en vue de mettre en place des attaques actives. Une série d'outils, fournie par la bibliothèque *BlueZ*, permet de manipuler les *dongles* HCI relativement rapidement et peut être utile dans une optique d'audit de sécurité : les utilitaires *hcitool*, *hciconfig* et *gatttool* permettent ainsi de manipuler les advertisements ainsi que de se connecter en tant que *Master* à un serveur GATT. Cependant, ces outils n'ont pas été conçus dans une optique de sécurité, et sont peu flexibles pour un usage offensif.

Un outil de collecte d'informations sur les serveurs ATT / GATT, nommé *bleah*, a également été proposé par *evilsocket* [7]. Il permet de scanner exhaustivement les différents services et caractéristiques d'un serveur GATT, et de communiquer avec certains d'entre eux. Cependant, cet outil manque de mécanismes d'interfaçage, qui limitent son usage à de la collecte d'informations active et empêchent son utilisation pour le développement d'outils connexes. Un outil plus complet, nommé *nRFConnect* et initialement développé pour le développement d'objets connectés, est disponible, et permet notamment de gérer le scan, la connexion et la collecte d'information sur un périphérique BLE en mode *Slave*. Il s'agit cependant d'une solution propriétaire, limitant de fait son utilisation au sein d'outils offensifs, et qui ne prévoit pas de mécanismes d'interfaçage avec d'autres outils.

Une pile protocolaire *BLE* partielle, développée en Python par Mike Ryan et nommée PyBT [12] a également été conçue, dans une optique de sécurité offensive, afin de pouvoir simuler le comportement d'un équipement de type *Slave* ou de type *Master*. Celle-ci est cependant difficile à manier et reste partielle, ne proposant notamment pas les dissecteurs permettant de travailler à plus haut niveau. Celle-ci constitue cependant incontestablement un bon point de départ au développement d'outils

offensifs pour le *Bluetooth Low Energy*, et a largement été réutilisée au sein du framework *Mirage*.

Deux stratégies de *Man-in-the-Middle* ont également été proposées et implémentées dans les outils *GATTacker* [10] et *BTLEJuice* [3]. Celles-ci consistent à « cloner » le serveur *GATT* de l'objet *Slave* à attaquer et simuler un périphérique identique : pour rendre l'attaque plus efficace, il est possible d'usurper l'adresse BD de l'objet attaqué. Dès lors, si un *Master* se connecte sur l'objet simulé, l'attaquant peut se connecter lui-même sur l'objet légitime, et rediriger le trafic d'une connexion à l'autre : il est alors en situation de *Man In The Middle*, et peut non seulement observer le trafic, mais également arbitrairement modifier certains paquets, ne pas les rediriger pour provoquer un déni de service ou injecter du trafic lui-même, à destination du *Slave* ou du *Master*.

Cette attaque présente l'avantage de ne nécessiter que du matériel standard : deux adaptateurs *Bluetooth* supportant le *BLE* sont suffisants pour mener ce type d'attaque. De plus, l'attaquant étant en mesure d'établir lui-même les connexions avec chacune des parties, cette stratégie évite la problématique du trafic chiffré, car les paramètres sont négociés par l'attaquant lui-même.

Les deux outils précédemment mentionnés diffèrent par la stratégie mise en œuvre pour forcer le *Master* à se connecter sur l'objet cloné et non sur l'objet légitime. *GATTacker* [10] exploite le fait que, si deux équipements disposant de la même adresse BD envoient des *advertisements* en même temps, le *Master* a une plus forte probabilité de se connecter sur celui dont la cadence d'émission d'*advertisements* est la plus élevée (l'émission du paquet de connexion, nommé *CONNECT_REQ*, faisant en effet suite à la réception par le *Master* d'un paquet d'*advertising* légitime). Les objets connectés ayant souvent des problématiques d'économie d'énergie, il est généralement possible d'émettre des *advertisements* plus rapidement pour l'attaquant qui n'est pas forcément soumis à cette contrainte.

BTLEJuice [3], quant à lui, exploite la caractéristique des objets fonctionnant en mode *Slave* de n'accepter qu'une seule connexion à la fois. La connexion avec le *Slave* est alors initiée dès le début de l'attaque : l'objet cesse alors d'émettre des *advertisements* et l'attaquant peut mettre en place l'objet cloné et simuler la phase d'*advertising* sans risque d'une connexion du *Master* sur l'objet légitime, celui-ci n'étant plus visible.

Ces deux outils souffrent cependant des limitations des bibliothèques nodeJS *noble* (fonctionnement en mode *Central*) et *bleno* (fonctionnement en mode *Peripheral*) qu'ils utilisent : en effet, celles-ci sont haut niveau et forcent l'attaquant à mettre en place un faux serveur *GATT* et à maintenir

sa cohérence avec celui de l'objet attaqué tout au long de la connexion. Plus problématique, elles ne permettent pas d'utiliser deux adaptateurs Bluetooth sur le même système d'exploitation, l'un fonctionnant en mode *Master* et l'autre en mode *Slave* (ce qui est évidemment indispensable à l'attaque).

Pour contourner ces problématiques, Damien Cauquil a implémenté pour *BTLEJuice* [3] l'utilisation de deux instances fonctionnant sur deux systèmes d'exploitation différents, et communiquant par l'intermédiaire de *Web Sockets*, tandis que *GATTacker* [10] a nécessité de modifier en profondeur le fonctionnement de ces bibliothèques, les rendant de fait non standard. Les solutions apportées sont ainsi fonctionnelles mais limitent considérablement la flexibilité et compliquent le déploiement de ces outils.

Finalement, plusieurs travaux concernant le *jamming* (ou brouillage) des connexions et des *advertisements BLE* ont été réalisés ces dernières années. Comme toutes les technologies sans fil, le *Bluetooth Low Energy* est sensible au *jamming*. Celui-ci permet d'interrompre une connexion entre un *Master* et un *Slave* en saturant le spectre radio des fréquences utilisées par le *BLE*. Des travaux intéressants sur le *jamming sélectif* des *advertisements* [2] ont notamment été proposés, permettant de masquer le trafic d'*advertising* d'un objet en particulier : ils ont été implémentés au sein d'un firmware destiné au nRF51, mais l'outil n'est pas disponible publiquement. Cette attaque est cependant particulièrement intéressante dans le cadre du développement des technologies de « Balises » telles qu'*iBeacon* ou *Eddystone*, qui se basent sur le mécanisme d'*advertising* pour émettre des données applicatives en broadcast. L'Ubertooth [16] et l'implémentation de *BTLEJack* [5] sur Micro:bitde Damien Cauquil proposent des mécanismes permettant de brouiller une connexion en cours, et, dans le cas de *BTLEJack*, d'*hijacker* une connexion établie. Cette nouvelle attaque, présentée à la DEFCON 26, utilise les différences de *timeout* entre le *Master* et le *Slave* pour forcer le *Master*, via l'utilisation de *jamming* sur l'utilisation des paquets du *Slave*, à se déconnecter et prendre ainsi sa place au sein de la communication. L'outil d'attaque est disponible publiquement et propose de nombreuses fonctionnalités intéressantes, il ne dispose cependant pas de mécanismes d'interfaçage directs avec des outils connexes.

Dans la prochaine partie, nous présenterons quelques éléments d'architecture du framework *Mirage* ainsi que son utilisation. Nous détaillerons le fonctionnement de quelques modules d'attaques implémentés au sein de celui-ci, ainsi que leur utilisation dans le cadre d'un audit d'objets connectés.

3 Présentation du framework Mirage

3.1 Présentation générale

Le framework *Mirage* a été développé en Python 3 et utilise un écosystème de bibliothèques open-source standard (telles que Scapy [1] ou pySerial). Il est organisé autour de quatre composants logiciels principaux :

- **Le cœur (`core`)** : il s'agit des mécanismes principaux du framework, gérant le point d'entrée, le lancement et le paramétrage des modules, le *scripting*, le système de tâches de fond (implémenté grâce au *multiprocessing*) et la gestion des signaux ;
- **Les bibliothèques (`libs`)** : il s'agit du composant contenant l'implémentation des différentes technologies sans fil supportées par le framework, mais également les mécanismes d'affichage / journalisation et les fonctions « utilitaires » (manipulation des tâches et des modules, gestion du temps, etc.) ;
- **Les modules (`modules`)** : il s'agit de plusieurs composants indépendants, implémentant les différentes attaques et outils du framework. Ce sont les éléments principaux à exécuter lors de l'utilisation du framework ;
- **Les scénarios (`scenarios`)** : il s'agit de *bindings* destinés à enrichir le fonctionnement de certains modules complexes. Il s'agit d'une collection de méthodes permettant de réagir à certains évènements (comme l'arrivée d'un paquet ou la pression sur une touche).

La première des tâches du composant dit *core* est de gérer le point d'entrée du module et les deux modes associés. En effet, le framework dispose d'un point d'entrée unique (le fichier *mirage.py*) qui permet de le lancer dans un mode dit *interactif* ou dans le mode *ligne de commande*. En mode *interactif*, l'utilisateur accède à une interface de type « interpréteur de commande », lui permettant de charger, paramétrer et exécuter les différents modules, mais également de gérer les tâches de fond. Il est également possible de paramétrer et lancer les modules directement depuis l'environnement `bash`, en utilisant le mode « ligne de commande ».

Ce découpage en modules permet ainsi d'implémenter des stratégies d'attaques génériques, paramétrables par l'intermédiaire des arguments fournis en entrée et capables de retourner un certain nombre d'informations en sortie. Le chargement et l'exécution des modules sont gérés au travers d'un module de chargement, dit *core.Loader*. Le rôle de ce chargeur est d'explorer et d'indexer le contenu du dossier *modules*, puis de gérer le chargement et l'exécution des classes correspondantes. Ce système prend en charge un mécanisme d'**exécution chaînée**, permettant de lancer

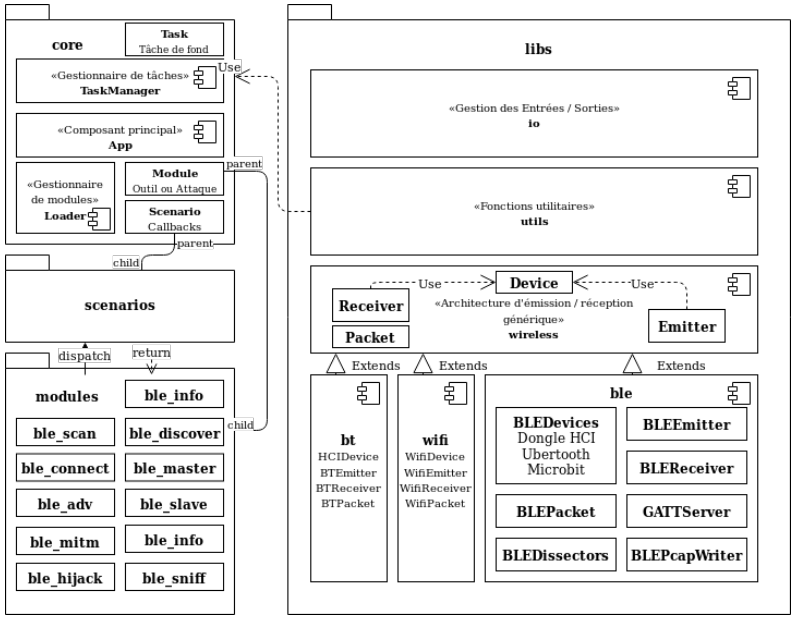

Fig. 4. Vision d'ensemble de l'architecture du framework Mirage.

successivement un premier module, puis un second, etc. en connectant chaque sortie du module *n* sur les paramètres d'entrée du module *n+1*. Ce fonctionnement permet de développer des petits modules effectuant chacun une tâche simple, puis de les exécuter successivement afin de mettre en place un comportement complexe. Les paramètres fournis en entrée et non utilisés sont également propagés par l'intermédiaire du mécanisme de chaînage. Cette logique étant similaire à celle des utilitaires bash sous *UNIX*, elle en réutilise l'opérateur «|» pour chaîner les modules.

Un exemple d'une telle exécution pourrait être le suivant :

```
$ ./mirage.py
--> load ble_connect|ble_discover
[INFO] Module ble_connect loaded !
[INFO] Module ble_discover loaded !
<< ble_connect|ble_discover >>--> set ble_connect1.INTERFACE hci1
<< ble_connect|ble_discover >>--> set ble_discover2.ATT_FILE out.ini
<< ble_connect|ble_discover >>--> run
```

ou en mode « ligne de commande » sous la forme :

```
$ ./mirage.py "ble_connect|ble_discover"
  ble_connect1.INTERFACE=hci1 ble_discover2.ATT_FILE=out.ini
```

Cette exécution, représentée en figure 5, exécutera donc (1) le module **ble_connect** avec les paramètres par défaut et l'interface *hci1* (fournie par l'utilisateur), puis propagera le paramètre *INTERFACE* au module **ble_discover** et exécutera ce dernier (2) avec le paramètre *ATT_FILE* fixé à "out.ini" et les autres paramètres fixés à leurs valeurs par défaut.

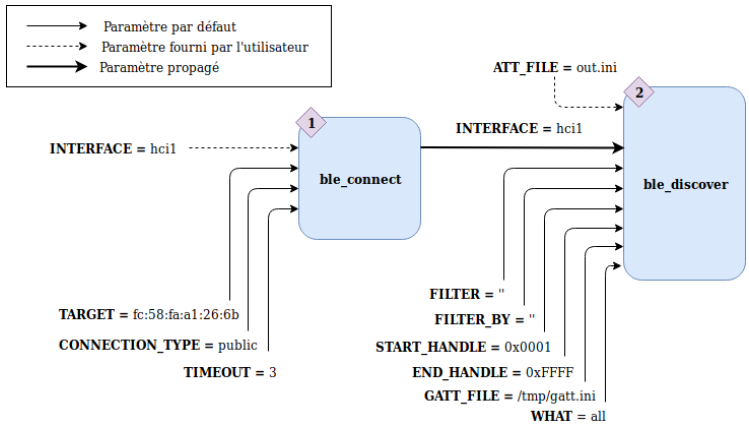

Fig. 5. Exemple d'exécution chaînée de deux modules.

La philosophie du framework consiste ainsi à implémenter les attaques et outils de façon modulaire et générique, et non directement des vulnérabilités propres à un type de système particulier, tout en permettant de personnaliser leur comportement de façon avancée pour l'adapter aux conditions d'audit par l'intermédiaire d'une série de *callbacks*, regroupés sous le terme de *scénarios*.

Les *scénarios* sont de simples classes, héritant de la classe *core.Scenario*, dont les méthodes réagissent à un signal particulier généré par un module. Il est ainsi possible de réagir à la réception d'un paquet particulier, au démarrage ou à la terminaison du module, ainsi que de fournir une interface utilisateur simple en réagissant aux pressions sur une touche. Chaque méthode d'un scénario peut retourner un booléen, dont la valeur indiquera si le comportement prévu par défaut par le module doit être exécuté ou non. Ce mécanisme assure ainsi une grande flexibilité et adaptabilité du framework aux conditions d'audit, et permet rapidement de modifier ou d'implémenter des comportements complexes. Finalement, le composant *core* permet également la gestion de *tâches de fond*, facilitant l'implémentation du multiprocessing. Il est également en charge de la gestion des fichiers de configuration.

Un autre composant indispensable au fonctionnement du framework est le composant *libs*. Il implémente un certain nombre de mécanismes nécessaires au fonctionnement des modules, notamment la gestion du *logging* et de l'affichage, l'architecture de manipulation des protocoles sans fil, la gestion des spécificités de ces protocoles, les parseurs et dissecteurs associés et quelques fonctionnalités complémentaires.

Une fonctionnalité essentielle implémentée par le composant *libs* est la gestion des protocoles sans fil. En effet, il était nécessaire de trouver une architecture logicielle à la fois simple d'utilisation et puissante, tout en permettant de s'interfacer avec des composants matériels très divers.

Afin de relever ce défi, le mécanisme d'émission et de réception des paquets a été implémenté sous la forme de trois composants principaux : **Device**, **Receiver** et **Emitter**. Le composant **Device** est chargé de s'interfacer avec un matériel donné. Il est possible que pour une même technologie, plusieurs *Device* soit définis, permettant ainsi de s'interfacer avec des composants matériels différents : c'est notamment le cas du *Bluetooth Low Energy*, pour lequel sont implémentés trois *Devices* représentant respectivement un *dongle* HCI, l'Ubertooth One ainsi que le firmware de *BTLEJack*. Un *Device* implémente *a minima* la méthode *send* (chargée d'émettre des données fournies sous la forme d'un tableau d'octets) et la méthode *recv* (chargée de recevoir un paquet et de le renvoyer sous la forme d'un tableau d'octets). Une méthode *isUp* implémente la vérification de la présence du matériel, et la méthode *init* est en charge de l'initialisation de celui-ci. Il est possible de définir des méthodes supplémentaires permettant d'implémenter des capacités complémentaires du matériel.

Le **Receiver** est l'interface de réception du protocole, avec laquelle les modules vont travailler. Celle-ci communique avec l'objet *Device* qui lui est associé afin de récupérer les données reçues. Elle implémente une méthode *_convert* (privée), permettant de transformer ces données en un objet *Packet*, représentant une abstraction de celles-ci et fournissant un accès rapide aux champs importants du paquet correspondant. Une méthode *next* permet de récupérer le paquet courant, tandis que la méthode *receive* permet de récupérer ceux-ci sous la forme d'un *générateur*[5]. Enfin, une méthode *onEvent* permet d'enregistrer une fonction *callback*, qui sera exécutée lors d'un événement particulier (telle que la réception d'un paquet particulier). Les fonctionnalités supplémentaires du *Device* sont accessibles directement, l'objet *Receiver* se comportant comme un *proxy* [8] pour ces méthodes.

5. Un *générateur* est un objet Python permettant de créer et de manipuler facilement les itérateurs, en ajoutant une couche d'abstraction supplémentaire.

L'**Emitter** est l'interface d'émission du protocole, avec laquelle les modules vont travailler. Celle-ci communique avec l'objet *Device* qui lui est associé afin d'envoyer les données spécifiées. Elle implémente elle aussi une méthode *_convert* (privée), permettant de transformer un objet *Packet* en données binaires. Les méthodes *send* et *sendp* (équivalentes) permettent d'émettre un ou plusieurs paquets sous la forme d'un objet de type *Packet*. Les fonctionnalités supplémentaires du *Device* sont accessibles directement, l'objet *Emitter* se comportant comme un *proxy* pour ces méthodes.

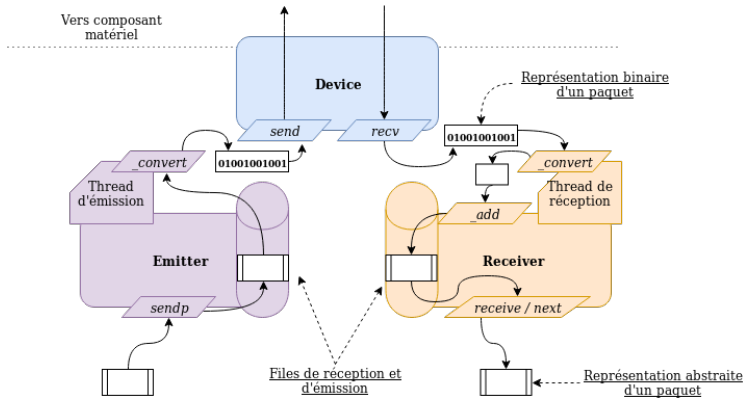

Fig. 6. Présentation de l'architecture générique d'émission / réception.

Les *Emitters* et *Receivers* communiquent avec l'objet *Device* qui leur est associé par l'intermédiaire de files (des structures de données de type *First In, First Out*). Le traitement de la sortie de chacune des files est assuré par un *thread* géré de façon transparente, c'est notamment lui qui réalise les éventuelles conversions données binaires / abstraction *Packet*. En revanche, les utilisateurs manipulent les paquets, au sein des modules, sous la forme d'abstraction héritant de la classe *Packet*. Ainsi, cela permet de ne pas manipuler directement des données brutes, mais d'offrir la possibilité d'implémenter des mécanismes de dissection, de conversion et de construction. Cette structure est particulièrement intéressante, car elle offre ainsi à l'utilisateur la possibilité d'implémenter des comportements complexes, ou de lui présenter une abstraction facile à manipuler, sans pour autant lui interdire de manipuler les paquets sous forme de données binaires. Cette structure a ainsi permis, dans le cas du protocole *BLE*, d'harmoniser les différents formats de paquets utilisés par chaque *Device* (HCI, BTLEJack ou Ubertooth) en une seule abstraction, disposant des informations couramment manipulées pour le paquet en question.

De nombreuses bibliothèques liées à l'utilisation, l'attaque et l'analyse des communications *Bluetooth Low Energy* ont également été implémentées au sein de *Mirage*. À l'heure actuelle, le framework est en effet capable de gérer les composants matériels suivants :

1. Un dongle HCI implémentant au minimum la spécification Bluetooth 4.0 [15] (permettant ou non de modifier l'adresse BD par l'intermédiaire de paquets dits *vendor-specific*) ;
2. Un Ubertooth One [16] ;
3. Un Micro:bitéquipé de la version 1.3 du firmware de *BTLEJack*, ou équipé de la version 3.14 (version modifiée pour le framework *Mirage*) du firmware de *BTLEJack* [5].

Une pile protocolaire *BLE* a été implémentée au sein du framework afin de contourner les limitations dues à l'utilisation de bibliothèques très haut niveau telles que Bleno et Noble. Elle permet notamment d'utiliser le dongle en tant que *Master* (capable de se connecter sur plusieurs *Slaves* simultanément) et en tant que *Slave* et permet l'usurpation d'adresse BD de manière transparente si le matériel le permet.

De nombreux dissecteurs facilitent l'analyse des données des couches ATT, GATT et Security Manager. Un serveur ATT / GATT basique a également été implémenté, afin de permettre de simuler de façon réaliste le comportement d'un objet connecté implémentant le rôle *Slave*, ainsi que les fonctions cryptographiques utilisées par la couche Security Manager.

Les API de l'*Ubertooth One* et de *BTLEJack* ont aussi été implémentées au sein du framework : elles sont développées en pur Python et ne font pas appel à des composants ou bibliothèques externes au framework pour fonctionner. Elles implémentent les principaux mécanismes de *sniffing*, *jamming* et *hijacking* proposés par ces différents composants matériels, et harmonisent leur comportement afin de proposer l'utilisation la plus unifiée possible. Le matériel est ainsi abstrait par le framework ; l'utilisateur peut donc se concentrer sur la logique de l'attaque ou de l'audit à implémenter.

Des outils plus bas niveau sont également proposés dans le cadre de l'analyse et de l'attaque du *BLE*. Un mécanisme d'export de fichier PCAP a été mis en place, permettant notamment d'exporter les communications sniffées. Des fonctions utilitaires permettent la conversion numéro de channel / fréquence, le calcul de CRC ainsi que la vérification de validité d'adresse d'accès (ou *Access Address*).

Finalement, un firmware *BTLEJack* personnalisé pour le framework a également été développé : numéroté 3.14, il permet notamment de profiter de fonctionnalités d'écoute et de *jamming* réactif des *advertisements* non présentes par défaut au sein du firmware proposé par Damien Cauquil.

3.2 Présentation des modules d'audit

Nous allons maintenant présenter les différents modules développés dans le cadre de l'audit des systèmes capables de communiquer via *Bluetooth Low Energy*. Ces modules font partie intégrante du framework *Mirage*, et permettent d'effectuer les tâches les plus courantes dans ce type d'audit sans impliquer de développement supplémentaire de la part de l'utilisateur. Cependant, il est possible de personnaliser le fonctionnement de certains d'entre eux par l'utilisation de *scenarios*, permettant ainsi de simuler précisément le fonctionnement d'un objet ou d'effectuer des actions particulières lors d'une attaque.

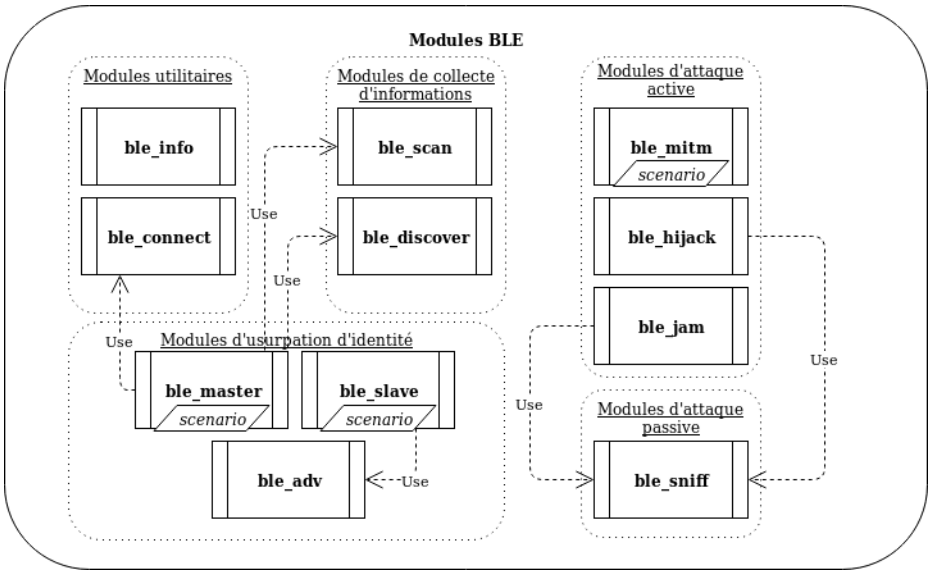

Fig. 7. Représentation des modules BLE implémentés dans le framework Mirage.

À l'heure actuelle, le framework propose cinq types de modules :

— Les modules **utilitaires** (*ble_info, ble_connect*) : il s'agit de modules permettant d'effectuer une action donnée (par exemple, se connecter à un objet) ou permettant de lister les informations associées à une interface donnée.

— Les modules de **collecte d'informations** (*ble_scan, ble_discover*) : il s'agit de modules destinés à la collecte d'informations, permettant par exemple de scanner l'environnement à la recherche d'objets en phase d'*advertisements* ou

d'identifier les attributs, services et caractéristiques d'un serveur ATT/GATT.
— Les modules d'**usurpation d'identité** (*ble_adv, ble_slave, ble_master*) : il s'agit de modules permettant de simuler le comportement d'un *Advertiser*, d'un *Peripheral* ou d'un *Central*, et potentiellement d'usurper l'identité d'un objet existant.
— Les modules d'**attaques passives** (*ble_sniff*) : il s'agit de modules d'écoute passive, destinés à l'observation passive d'*advertisements* et de connexions (nouvellement établies ou existantes). Le comportement des différents composants matériels est harmonisé.
— Les modules d'**attaques actives** (*ble_mitm, ble_hijack, ble_jam*) : il s'agit de modules destinés à la mise en place d'attaques actives telles que des *Man-in-the-Middle*, l'*hijacking* ou le *jamming* de communications *Bluetooth Low Energy*. Le module Man-in-the-Middle implémente des mécanismes permettant de récupérer la clé de chiffrement s'il observe une phase d'appairage, et autorise la manipulation de données chiffrées si l'attaquant dispose de la clé.

Une description plus détaillée du fonctionnement et de l'usage des différents modules est disponible sur l'annexe en ligne [6].

4 Expérimentations : audit d'une ampoule connectée

4.1 Présentation générale

Les outils développés dans le cadre du framework nous offrent une base solide pour réaliser une étude du protocole de communication d'une ampoule connectée. L'objectif de cet audit sera de procéder à la rétro-ingénierie du protocole de communication de l'ampoule connectée afin d'évaluer la surface d'attaque de cet objet : selon les résultats obtenus, nous pourrons ainsi envisager d'explorer un ou plusieurs chemins d'attaque, tels que la prise de contrôle des fonctionnalités légitimes de l'ampoule ou l'injection de code malveillant au sein du micro-logiciel.

Dans notre cas, il est à noter que l'ampoule peut se connecter via le *Bluetooth Low Energy* à un smartphone exécutant une application mobile permettant de manipuler la couleur, la luminosité, la température, l'état « allumé / éteint » et de mettre à jour le firmware contenu dans l'ampoule.

Dans un premier temps, il a été nécessaire d'ajouter l'ampoule à auditer dans l'application Android, et d'utiliser les éléments d'interface pour découvrir les fonctionnalités légitimes de l'ampoule connectée. L'objectif de cet audit était d'utiliser les outils développés dans le framework pour comprendre et documenter les comportements précédemment énumérés.

4.2 Collecte d'informations active

Dans un second temps, il a paru intéressant de lister les attributs du serveur *ATT* stockés dans l'ampoule, ainsi que leurs abstractions *GATT* sous la forme de services primaires, secondaires et de caractéristiques. En effet, disposer à la fois des données *ATT* et des données *GATT* permet de se faire une représentation précise des hiérarchies entre les attributs, tout en conservant une vision bas niveau de la structure de l'objet.

Pour cela, il était nécessaire de *dumper* l'ensemble de la base de données du serveur *ATT* et du serveur *GATT*. Il a donc été nécessaire d'utiliser les modules suivants :

- **ble_scan** : pour scanner l'environnement à la recherche des *advertisements* des objets connectés ;
- **ble_connect** : pour se connecter à l'objet audité ;
- **ble_discover** : pour énumérer les services, caractéristiques et attributs associés aux couches *ATT*/*GATT* de l'objet.

La première étape consiste donc à lancer le module **ble_scan**, dont voici la sortie (tronquée pour des raisons de place) :

```
$ ./mirage.py ble_scan
[INFO] Module ble_scan loaded !
[SUCCESS] HCI Device (hci0) successfully instanciated !
+Devices found------+-------+------------+-------+-----------+
| BD Address        | Name  | Company    | Flags | Data      |
+-------------------+-------+------------+-------+-----------+
| C4:BE:84:39:8E:07 | Salon | TI         | [...] | 1009[...] |
| 3E:00:22:CF:2F:AD |       | Microsoft  | [...] | 1eff[...] |
+-------------------+-------+------------+-------+-----------+
```

Ce scan nous permet d'obtenir trois informations pertinentes : l'adresse BD, le fabriquant du système ainsi que le nom de l'objet, directement extraits des informations diffusées par le mécanisme d'*advertising*. Ici, l'objet qui nous intéresse correspond au nom `Salon`, possède l'adresse BD `C4:BE:84:39:8E:07`.

À l'aide de cette information, il est ensuite possible de lancer une phase de collecte d'informations composée d'une connexion sur l'objet, puis d'une procédure de découverte de l'intégralité de ses services et caractéristiques (au niveau GATT). Cette opération nous permet ainsi d'obtenir la structure des couches protocolaires hautes, notamment la couche GATT. On exportera également ces données sous la forme d'un fichier .cfg en renseignant le paramètre *ATT_FILE* du module **ble_discover**.

```
$ ./mirage.py "ble_connect|ble_discover"
ble_connect1.TARGET=C4:BE:84:39:8E:07
ble_discover2.ATT_FILE=/tmp/gatt.cfg
```

La sortie du module est disponible dans les annexes en ligne [6]. On constate donc la présence de trois services communs sur la plupart des objets connectés :

— **Generic Access** (handles `0x0001` à `0x000b`) ;
— **Generic Attribute** (handles `0x000c` à `0x000f`) ;
— **Device Information** (handles `0x0010` à `0x001e`).

Ainsi que trois services non identifiés, propres à l'objet (handles `0x001f` à `0xffff`).

On peut souligner la présence de deux caractéristiques potentiellement intéressantes, contenues dans le premier de ces trois services : **DataTransmit** (de *handle* `0x0020`) et **DataReceive** (de *handle* `0x0023`).

4.3 Rétro-ingénierie du protocole de communication

Pour déterminer plus en détail le fonctionnement de l'objet, on réalise une attaque *Man In The Middle* tout en activant les différentes fonctionnalités de la lampe depuis l'application pour analyser le trafic correspondant. Aucun scénario n'étant chargé avec le module de *Man In The Middle*, le comportement par défaut (redirection et *logging* des paquets) est appliqué :

```
$ ./mirage.py ble_mitm TARGET=C4:BE:84:39:8E:07
SHOW_SCANNING=no ADVERTISING_STRATEGY=preconnect
```

La trace de l'attaque, annotée en fonction des actions réalisées sur le téléphone, est disponible dans les annexes en ligne.

Cette attaque nous permet de déterminer la forme générale des messages de commande. Ils sont déclenchés par l'intermédiaire d'une **Write Request**, à destination du *handle* de valeur `0x0021` (il s'agit de la caractéristique **DataTransmit** identifiée à l'étape précédente).

Les messages ont tous la forme suivante :

`55`	identifiant (1 octet)	paramètre (taille variable)	`0d 0a`

Les messages d'allumage / extinction possèdent l'identifiant `0x10`, et un paramètre sur un octet valant : `0x01` (*On*) ou `0x00` (*Off*)

Allumage de la lumière	`55 10 01 0d 0a`
Extinction de la lumière	`55 10 00 0d 0a`

Les messages de manipulation de la couleur possèdent l'identifiant `0x13`, suivi de trois octets composant le code *RGB* de la couleur souhaitée :

Modification de la couleur (Rouge)	`55 13 ff 00 00 0d 0a`
Modification de la couleur (Vert)	`55 13 00 ff 00 0d 0a`
Modification de la couleur (Bleu)	`55 13 00 00 ff 0d 0a`

Les messages de modification de la luminosité, de la température et de changement de mode ont également été identifiés, le détail du format de ces trames est disponible dans l'annexe en ligne.

Pour confirmer nos hypothèses, on peut se connecter à la lampe à l'aide du module **ble_master** afin de vérifier qu'on observe bien le comportement prévu :

```
$ ./mirage.py ble_master
```

La trace de l'attaque est disponible dans les annexes en ligne. Un scénario pour l'attaque Man In The Middle a été développé, permettant de modifier le comportement de l'ampoule lors des actions sur l'interface (les coordonnées Red et Green sont inversées, l'appui sur On provoque un clignotement de l'ampoule). Il est présenté au sein des annexes en ligne.

4.4 Récupération du firmware légitime

La dernière étape de notre audit va consister à analyser la procédure de mise à jour du firmware de l'ampoule : en effet, lors de la connexion de l'application, une boîte de dialogue apparaît sur l'application afin de proposer la mise à jour du micro-logiciel via le *Bluetooth Low Energy*. Pour étudier cette phase de mise à jour, nous allons utiliser le module **ble_sniff** pour sniffer la communication depuis son initiation :

```
$ ./mirage.py ble_sniff CHANNEL=37 SNIFFING_MODE=newConnections
  INTERFACE=microbit0
```

Cette capture (disponible dans les annexes en ligne et annotée en fonction des actions) nous permet d'établir le lancement de la procédure de mise à jour : plusieurs requêtes sont ainsi dédiées à l'identification du modèle de l'ampoule et de la version du firmware, et amorcent le déclenchement de la procédure de mise à jour *Over-the-air*. Une fois ces échanges réalisés, le Master se met à écrire par l'intermédiaire de **Write Commands** dans le *handle* `0x0040` des valeurs composées d'un compteur sur les deux premiers octets, suivie de ce qui semble correspondre aux données du *firmware* par blocs de 16 octets. On peut alors implémenter le scenario **slave_lightbulb**, conçu pour instrumenter le module **ble_slave** selon l'échange précédemment présenté, qui permet de récupérer le *firmware* dans son intégralité dans un fichier `firmware.bin`. Le code de ce scénario est présenté dans les annexes en ligne.

La création d'un clone identique à l'ampoule en terme d'adresse BD, d'*advertisements* et de couches GATT présente de multiples intérêts. Tout

d'abord, cela peut permettre facilement de simuler le comportement de l'objet à auditer. Cependant, on peut aussi concevoir cette stratégie de clonage comme une attaque de type « Déni de Service » : une telle approche sur une serrure connectée par exemple pourrait permettre de récupérer le code de déverouillage de la serrure, en laissant l'objet cloné à faible distance de l'objet légitime. Ce type de stratégie pourra être instancié en une seule commande par l'exécution chaînée suivante :

- **ble_scan** : Récupération des données d'*advertisements* ;
- **ble_connect** : Connexion sur l'ampoule ;
- **ble_discover** : Découverte des données GATT (services et caractéristiques) et export dans un fichier .cfg ;
- **ble_adv** : Mise en place des *advertisements* ;
- **ble_slave** : Mise en place du comportement d'un *Slave* utilisant les mêmes données GATT et implémentant le scénario *slave_lightbulb*.

```
$ ./mirage.py "ble_scan|ble_connect|ble_discover|ble_adv|ble_slave"
  ble_scan1.TARGET=C4:BE:84:39:8E:07
  ble_discover3.GATT_FILE=/tmp/gatt.cfg
  ble_adv4.INTERFACE=hci1
  ble_slave5.SCENARIO=slave_lightbulb
```

Après l'exécution des modules de collecte d'informations, le slave est instancié et exécute le scénario de récupération du *firmware*. À l'issue de cette exécution, on dispose du *firmware* dans le fichier `/tmp/firmware.bin`. Un désassemblage de l'application Android nous a permis de valider l'intégrité du fichier ainsi récupéré vis-à-vis de celui récupéré sur le serveur Web du fabricant de l'ampoule par l'application.

Cet audit nous a permis de montrer qu'aucun mécanisme de sécurité ne semble implémenté sur cet objet. On note ainsi que la transmission des commandes ainsi que la procédure de mise à jour *Over The Air* semblent transmettre les données en clair, menant à poser la question du chiffrement de telles données. La rétro-ingénierie de ce firmware sort du cadre de cette présentation, cependant celle-ci pourrait ouvrir des pistes intéressantes, telles que l'injection d'un firmware malveillant au sein de l'ampoule connectée.

5 Conclusion

Dans cet article, nous avons présenté les problématiques liées à la sécurité des objets connectés, l'état de l'art de la sécurité offensive pour la technologie *Bluetooth Low Energy* ainsi qu'une contribution à celle-ci sous la forme d'un framework destiné à l'audit de ce type de systèmes.

Les différents modules présentés dans cet article constituent une tentative d'approche unifiée et modulaire de ce type d'outils, et offrent une base solide pour le développement ultérieur de nouveaux outils. L'exemple d'audit proposé à la fin de l'article illustre bien la philosophie de l'outil : disposer d'une approche fonctionnelle et modulaire, abstraire les éléments techniques et unifier les API afin de pouvoir concevoir attaques et outils sans devoir écrire des centaines de lignes de code techniques mais sans grand intérêt vis-à-vis de l'attaque développée. Dans un contexte d'expansion rapide de ce type de systèmes, il semble indispensable de produire un éco-système d'outils destinés à l'audit de sécurité qui soit à la fois flexible et puissant, mais également stable, cohérent et en adéquation avec les bonnes pratiques du développement logiciel traditionnel.

Nous avons fait le choix dans cet article de présenter uniquement les modules et développements liés à l'analyse du *Bluetooth Low Energy*. Cependant le framework a été conçu dans une optique de généricité et de nombreuses autres technologies sans fil sont intégrées (telles que les protocoles ShockBurst et Enhanced ShockBurst, le Wi-Fi, le Zigbee, l'infrarouge...) au sein du framework. L'architecture logicielle d'émission / réception a notamment été construite afin d'être facilement adaptable à de nombreux composants matériels, et les choix techniques réalisés ont été avant tout pensés dans une logique d'adaptabilité, avec le développement d'une API unifiée, facile d'utilisation et puissante afin d'harmoniser le développement de ce type d'outils. Dans cette optique, le code du framework et sa documentation sont mis à disposition sous une license libre, afin de faciliter les contributions extérieures.

Références

1. Documentation officielle de Scapy. `https://scapy.readthedocs.io`.
2. Sebastian Bräuer, Mehran Roshandel, Sven Zehl, et Soroush Mashhadi Sohi. On Practical Selective Jamming of Bluetooth Low Energy Advertising. 2016.
3. Damien Cauquil. BtleJuice, un framework d'interception pour le Bluetooth Low Energy. `https://www.slideshare.net/NetSecureDay/nsd16-btle-juice-un-framework-dinterception-pour-le-bluetooth-low-energy-damien-cauquil`, 2017.
4. Damien Cauquil. Weaponizing the BBC Micro:bit. `https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEFCON-25-Damien-Cauquil-Weaponizing-the-BBC-MicroBit.pdf`, 2017.
5. Damien Cauquil. You'd better secure your BLE Devices or we'll kick your butts! `https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/Damien%20Cauquil%20-%20Updated/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf`, 2018.

6. Romain Cayre, Jonathan Roux, Eric Alata, Vincent Nicomette et Guillaume Auriol, Mirage : un framework offensif pour l'audit du Bluetooth Low Energy – Annexes. `http://homepages.laas.fr/rcayre/SSTIC2019/`, 2019.
7. "evilsocket". Dépôt Github de Bleah, un outil de collecte d'informations actif pour le Bluetooth Low Energy. `https://github.com/evilsocket/bleah`, 2009.
8. Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
9. Naresh Gupta. *Inside Bluetooth Low Energy, Second Edition*. Artech House, 2016.
10. Sławomir Jasek. Gattacking Bluetooth Smart Devices. 2017.
11. Michael Ossmann. HackRF : low cost Software Radio platform. `https://github.com/mossmann/hackrf`, 2018.
12. Mike Ryan. Dépôt Github de PyBT, l'implémentation logicielle d'une pile Bluetooth Low Energy. `https://github.com/mikeryan/PyBT/tree/master/PyBT`, 2009.
13. Mike Ryan. Bluetooth : With Low Energy comes Low Security. 2013.
14. Mike Ryan. How Smart is Bluetooth Smart ? 2013.
15. Bluetooth SIG. *Bluetooth : Core Specifications, v. 5.0*, 2016.
16. Dominic Spill. Ubertooth. `http://ubertooth.sourceforge.net/`, 2012.
17. Kevin Townsend. Introducing the Bluefruit LE Sniffer. `https://learn.adafruit.com/introducing-the-adafruit-bluefruit-le-sniffer`, 2012.

L'annexe en ligne, ainsi que les sources des figures, sont disponible à l'adresse suivante : `http://homepages.laas.fr/rcayre/SSTIC2019/`