OASIS: un framework pour la détection d'intrusion embarquée dans les contrôleurs Bluetooth Low Energy

Romain Cayre, Clement Chaine, Guillaume Auriol, Vincent Nicomette, Geraldine Marconato

OASIS: un framework pour la détection d'intrusion embarquée dans les contrôleurs Bluetooth Low Energy

Romain Cayre[a,c], Clément Chaine[b], Guillaume Auriol[a,b], Vincent Nicomette[a,b], Géraldine Marconato[c]

[a]CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[b]Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
[c]APSYS.Lab, APSYS

Email: [a]prenom.nom@laas.fr, [b]prenom.nom@insa-toulouse.fr, [c]prenom.nom@airbus.com

Abstract

Les technologies de communication sans fil, telles que Zigbee ou Bluetooth Low Energy (BLE), sont aujourd'hui massivement utilisées par les objets connectés. Si ces nouveaux protocoles sont conçus pour résoudre des contraintes fonctionnelles (faible consommation d'énergie, mobilité, communications pair à pair ...), ils souffrent cependant de nombreux problèmes de sécurité. De nombreux défis techniques et scientifiques restent à résoudre afin de détecter efficacement les différentes attaques récemment publiées. Dans cet article, nous nous concentrons sur la sécurité de l'un des protocoles IoT les plus couramment utilisés, BLE, et plus particulièrement sur la détection des attaques ciblant les couches basses de ce protocole. Nous proposons une approche consistant à intégrer un système de détection d'intrusion dans les contrôleurs BLE. Cette approche permet de résoudre de multiples défis techniques liés à la conception du protocole et peut être facilement déployée sur de nombreux équipements existants supportant le protocole. Nous décrivons plusieurs heuristiques de détection ainsi intégrées dans des contrôleurs BLE, permettant de détecter avec succès 6 attaques majeures ciblant ce protocole. Nous présentons également OASIS, un *framework* générique permettant d'automatiser l'instrumentation des contrôleurs BLE, pour y inclure ces heuristiques de détection. Nous décrivons son architecture modulaire, comment nous l'avons implémenté avec succès sur cinq puces BLE largement utilisées intégrant des piles protocolaires hétérogènes, et comment nous l'avons utilisée pour détecter 6 attaques bas niveau critiques.

1 Introduction

De nombreux protocoles de communication sans fil [7, 1] sont aujourd'hui utilisés pour assurer la connectivité des objets connectés. Ces protocoles ont été conçus pour réduire la consommation énergétique, la complexité des piles protocolaires et pour faciliter les communications pair à pair, de façon à s'adapter aux ressources limitées de ces objets connectés. L'un des protocoles les plus utilisés aujourd'hui est le BLE, une variante du protocole Bluetooth intégré nativement dans un grand nombre d'équipements, allant des smartphones à des objets connectés aux ressources très limitées. Au cours des dernières années, ce protocole a été largement utilisé dans de nombreux cas d'application, en raison de son déploiement massif, de sa faible complexité et de sa polyvalence.

En conséquence, la sécurité de ce protocole est devenue une préoccupation majeure. De multiples vulnérabilités critiques [6, 17, 5, 4, 13] ont été découvertes récemment, illustrant l'intérêt croissant pour cette technologie et sa sécurité. Certaines de ces vulnérabilités sont liées à l'implémentation de la pile protocolaire et peuvent potentiellement être corrigées par les fabricants[5, 6, 17], tandis que d'autres [4, 13, 11, 19, 9] sont liées à la conception du protocole lui-même et ne peuvent pas être facilement corrigées sans modifier la spécification elle même.

Dans ce contexte, il devient fondamental de développer des mesures défensives, en particulier des systèmes de détection d'intrusion (IDS), permettant de détecter ce type d'attaques sans fil. Cependant, la conception d'un tel système reste un défi majeur. En effet, la conception du protocole BLE introduit de nombreuses contraintes techniques. Premièrement, le protocole est difficilement analysable par une sonde externe, principalement parce qu'il utilise un algorithme de saut de canal lors des connexions. Une connexion BLE saute régulièrement d'un canal à l'autre et peut utiliser n'importe lequel des 40 canaux de la bande ISM 2,4 GHz, forçant un IDS basé sur l'analyse du trafic réseau à surveiller en permanence une large bande de fréquences afin d'analyser exhaustivement les communications, augmentant ainsi le coût et la complexité d'un tel système. De plus, la nature sans fil du protocole introduit des problèmes liés à la différence potentielle de perception entre la sonde externe et les nœuds eux-mêmes. Contrairement à un protocole filaire, de nombreux facteurs peuvent avoir un impact sur l'exhaustivité et la représentativité du trafic surveillé, comme la position de la sonde dans l'environnement ou sa sensibilité. Le protocole étant également principalement utilisé pour établir des communications pair à pair, il n'est pas possible de surveiller facilement le trafic depuis un nœud central. Enfin, de nombreux équipements BLE sont dédiés à un usage mobile, générant un environnement dynamique dans lequel il est difficile d'identifier si la présence d'un nœud donné est légitime ou non.

Dans ce contexte, une approche complémentaire pertinente consiste à intégrer un IDS au sein des équipements directement, de façon à surveiller le trafic localement afin de détecter des attaques. Cependant, la conception d'un tel système est difficile pour différentes raisons. Tout d'abord, une pile protocolaire BLE est généralement divisée en deux parties: la partie *Hôte* (ou *Host*), qui gère les couches applicatives du protocole, et la partie *contrôleur* (ou *Controller*), qui est en charge des couches inférieures. Dans la plupart des cas, ces deux composants communiquent entre eux à l'aide d'une interface standardisée appelée *Host Controller Interface (HCI)*. Des études précédentes [3] ont déjà essayé d'instrumenter la partie *Hôte* de la pile afin de détecter des attaques. Cette approche est intéressante car l'instrumentation de la partie *Hôte* est généralement simple. Cependant, cette stratégie souffre de différentes limitations. Tout d'abord, la majeure partie du trafic de bas niveau est cachée à l'*Hôte* par conception. Cette situation est problématique car de nombreuses attaques existantes [11, 13] visent les couches inférieures de la pile et ne peuvent pas être détectées efficacement par un tel système. Un autre problème est lié à l'existence d'implémentations non standard de la pile, qui peuvent exposer une API propriétaire pour interagir avec le contrôleur au lieu de l'interface *HCI*: cette situation étant courante dans de nombreux équipements *IoT*, elle ne peut être ignorée lors de la conception d'un tel système.

En conséquence, instrumenter la partie *contrôleur* afin d'y insérer des mécanismes de sécurité constitue probablement le moyen le plus efficace pour détecter les attaques BLE. En effet, la partie *contrôleur* a accès au trafic bas niveau, permettant ainsi d'identifier les attaques ciblant les couches basses tout en permettant également la détection des attaques ciblant les couches applicatives. Cette stratégie permet l'extraction et l'exploitation de nombreuses caractéristiques pertinentes accessibles par le *contrôleur*, telles que le RSSI ou la validité du CRC, et est particulièrement adaptée pour mettre en place des mécanismes défensifs bas niveau permettant de détecter ou de prévenir une attaque. Cependant, l'instrumentation de la partie *contrôleur* n'est pas une tâche triviale. Premièrement, les implémentations des *contrôleur* sont généralement propriétaires et non documentées: la seule

façon de comprendre et d'instrumenter leurs composants internes est d'effectuer manuellement la rétro-ingénierie des *firmwares* correspondants, ce qui est un processus long, délicat et sujet aux erreurs. De plus, ces *contrôleur* sont implémentés sur de nombreuses puces différentes, basées sur des architectures hétérogènes. Leur instrumentation est également difficile, les fabricants ne proposant généralement aucun moyen simple de les patcher pour y inclure du code défensif. Enfin, les couches basses d'une pile protocolaire sont généralement soumises à de fortes contraintes temporelles, ce qui se traduit souvent par un code optimisé, potentiellement difficile à modifier et à améliorer.

Malgré ces difficultés, nous démontrons dans cet article la pertinence de cette approche, et proposons une suite d'outils logiciels facilitant sa mise en place. Les contributions de cet article sont les suivantes :

- Nous proposons une approche innovante pour la conception d'un système de détection d'attaques BLE, consistant à intégrer des heuristiques de détection d'intrusion directement au sein des contrôleurs. Celle ci permet l'identification de caractéristiques pertinentes accessibles par les contrôleurs, qui peuvent être utilisées pour caractériser l'occurrence d'attaques.

- Nous démontrons la pertinence de notre approche en la testant sur 6 attaques majeures, ciblant les couches basses du protocole. Nous avons ainsi conçu 6 modules permettant la détection de ces attaques et les avons intégrés au sein de différents contrôleurs BLE, nous permettant de détecter efficacement ces attaques avec de très bons taux de faux positifs et faux négatifs.

- Nous présentons un *framework* modulaire et générique, baptisé *OASIS*, dédié au développement de ces modules de détection d'intrusion. Nous l'avons validé sur différentes cartes de développement intégrant des contrôleurs d'architectures différentes, mais aussi des équipements commerciaux tels que smartphones ou des objets connectés.

- Nous avons réalisé la rétro-ingénierie de trois piles protocolaires très répandues, et nous avons développé un ensemble d'outils de rétro-ingénierie destiné à faciliter leur analyse. Ceux ci permettent d'identifier automatiquement les principales fonctions et zones mémoires nécessaires à l'instrumentation des contrôleurs et à l'implémentation des modules de détection.

Le plan de l'article est le suivant. La section 2 présente les travaux connexes à notre approche. La section 3 décrit brièvement le protocole BLE du point de vue de la sécurité, présente les six principales attaques de bas niveau sur lesquelles nous nous concentrons ainsi que les stratégies de détection correspondantes, et justifie le développement d'un *framework* intégré pour détecter les attaques de bas niveau. La section 4 décrit la conception globale du *framework OASIS*. La section 5 présente deux types d'architectures logicielles de contrôleurs que nous avons étudiés et instrumentés pour y intégrer les heuristiques de détection. La section 6 décrit les expérimentations que nous avons réalisées pour montrer la pertinence de notre *framework* et nos modules de détection dans un environnement réaliste. Enfin, la section 7 discute des limites de notre approche, tandis que la section 8 conclut l'article et propose de futures pistes de recherche.

2 Etat de l'art

Ces dernières années, plusieurs travaux ont contribué à l'amélioration de la sécurité du BLE. L'un des principaux défis techniques à résoudre pour surveiller efficacement les communications BLE est lié à l'utilisation d'algorithmes de saut de canal, compliquant considérablement l'écoute passive du protocole. Dans [26], M. Ryan a souligné les problématiques liés à ces algorithmes et a développé des heuristiques permettant d'inférer les paramètres de connexion en collectant et analysant des

événements spécifiques. Cette approche a permis le développement d'un sniffer adapté à la couche liaison, *l'Ubertooth One*, capable de déterminer les paramètres d'une connexion spécifique et de se synchroniser avec l'algorithme de saut de canal pour écouter passivement la communication. Cet algorithme a par la suite été amélioré par D. Cauquil dans [10] pour prendre en compte le mécanisme de *channel map*, destiné à permettre le *blacklisting* de certains canaux. Son algorithme ayant été implémenté sur une carte de développement spécifique, le *BBC Micro:Bit*, une implémentation similaire a été adaptée à *l'Ubertooth One* par S. Sarkar et al[2]. Dans [12], D. Cauquil a également développé une approche permettant de synchroniser un sniffer avec le deuxième algorithme de saut de canal, récemment introduit dans la spécification et basé sur un générateur de nombres pseudo-aléatoires. Dans [25], S. Qasim Khan a présenté *Sniffle*, une nouvelle implémentation de sniffer permettant de faciliter la synchronisation avec une connexion en suivant l'équipement ciblé pendant sa phase d'*advertising* .

Bien que ces travaux n'aient pas nécessairement été réalisés dans une perspective défensive, ils sont cependant pertinents dans le cadre de la détection d'intrusion BLE, la plupart des travaux défensifs existants s'appuyant sur des sniffers pour surveiller le trafic BLE. Malheureusement, ces approches passives souffrent de plusieurs limitations sérieuses qui ont un impact conséquent sur l'exhaustivité et la représentativité des communications surveillées. Premièrement, la plupart de ces sniffers ne peuvent surveiller qu'une seule connexion à la fois. G. del Arroyo et al ont tenté dans [18] de répondre à cette problématique en implémentant sur l'*Ubertooth one* un algorithme opportuniste basé sur un ordonnanceur et permettant de surveiller plusieurs connexions simultanément. Bien que ce travail semble prometteur, il est encore limité par le matériel utilisé et peut manquer certains paquets en fonction de l'environnement. Deuxièmement, la plupart des implémentations existantes sont instables, en partie à cause de l'utilisation de diverses heuristiques, qui ne sont pas adaptées à certains équipements (par exemple, certains smartphones mettent à jour fréquemment le *channel map* au cours d'une même connexion, compliquant considérablement l'inférence de ce paramètre par un sniffer). A notre connaissance, surveiller de manière exhaustive le trafic BLE au niveau de la couche liaison à partir d'une sonde externe, notamment en mode connecté, reste aujourd'hui un défi conséquent.

Cette situation a un impact important sur les travaux de recherche visant à développer des systèmes de détection d'intrusion pour ce protocole: la plupart d'entre eux étant basés sur les sniffers mentionné précédemment, ils s'appuient généralement sur la seule surveillance des *advertisements*, plus simple à surveiller mais limitant leur portée à des attaques d'usurpation d'identité ciblant le mode *advertising*. Dans [30], J. Wu et al. ont présenté *BlueShield*, une approche permettant de détecter les attaques d'usurpation d'identité en profilant les équipements surveillés à l'aide de multiples caractéristiques extraites des paquets d'*advertisements*. Dans [29], Y. Sung et al. ont exploré l'utilisation de l'intensité du signal reçu (RSSI) pour détecter les intrus. Dans [31], M. Yaseen et al. ont présenté *MARC*, un *framework* visant à détecter les attaques Man-in-the-Middle, exploitant quatre fonctionnalités déduites des paquets d'*advertisements* telles que l'intervalle d'*advertising* ou les niveaux RSSI.

D'autres travaux de recherche se sont également consacrés à l'analyse du trafic en mode connecté. Dans [24], A. Newaz et al. combinent une approche basée sur les n-grammes avec diverses techniques d'apprentissage automatique pour effectuer la détection d'intrusion par l'analyse de modèles de flux de trafic anormaux sur les dispositifs médicaux personnels. De même, dans [22], A. Lahmadi et al. explorent l'utilisation de techniques d'apprentissage automatique pour identifier les attaques Man-in-the-Middle en créant un modèle de comportements légitimes basé sur des caractéristiques telles que le RSSI, les numéros de canaux ou la distance. Bien que ces travaux fournissent des résultats intéressants concernant l'analyse du trafic, ils se concentrent sur une détection hors ligne appliquée à des jeux de données collectés en amont, et sont difficiles à déployer en pratique.

Certains travaux se sont également concentrés sur la construction d'IDS pour les réseaux BLE de type Mesh. Dans [21], A. Lacava et al. proposent un IDS distribué basé sur le déploiement de nœuds de surveillance au sein du réseau, capables de collecter le trafic local et de détecter les attaques en se basant sur une prise de décision coopérative. De plus, dans [20], M. Krzyszton et al. ont effectué des simulations pour choisir les placements optimaux des noeuds de surveillance dans ce type d'IDS coopératif. L'approche distribuée adoptée par ces travaux de recherche est particulièrement pertinente pour les réseaux Mesh, mais semble difficilement applicable aux équipements BLE existants, qui n'utilisent généralement pas de mécanismes de routage. Cependant, nos travaux pourraient compléter efficacement ce type de stratégie en permettant son déploiement sur des équipements réels tout en s'adaptant aux topologies existantes.

Notre approche étant basée sur une détection locale réalisée sur les nœuds eux-mêmes, elle s'affranchit de beaucoup des problèmes techniques qui limitent les travaux de recherche existants. Tout d'abord, elle ne repose pas sur une stratégie d'écoute passive et n'est donc pas soumise aux contraintes techniques liées au *sniffing* BLE, les nœuds collectant et analysant les caractéristiques localement, que ce soit en mode *advertising* ou en mode *connecté*. Notre approche évite également certains problèmes inhérents à l'utilisation d'une sonde externe tels que le placement de la sonde. Nous démontrons également la pertinence d'embarquer des heuristiques de détection légères et bas niveau pour détecter les attaques BLE existantes, ce travail étant le premier à fournir une détection pratique d'un certain nombre d'attaques bas niveau ciblant le mode *connecté*, telles que InjectaBLE[13], BTLEJack[11] ou KNOB[4]. Notre approche fournit également une solution flexible, qui peut être déployée en pratique sur un seul équipement ou servir de base à un IDS distribué. Nous considérons qu'elle peut compléter efficacement les approches existantes, à la fois en facilitant l'instrumentation d'équipements réels et en fournissant un *framework* facile d'utilisation, modulaire et extensible pour construire des heuristiques de détection sur des équipements hétérogènes.

3 Détection des attaques bas niveau BLE

Dans cette section, nous introduisons d'abord brièvement quelques pré-requis concernant le protocole BLE. Ensuite, nous décrivons six attaques bas niveau critiques liées à la conception du protocole BLE et montrons comment ces attaques peuvent être détectées au moyen de caractéristiques appropriées disponibles au sein des contrôleurs BLE. Enfin, nous donnons la liste des ressources nécessaires pour implémenter ces heuristiques de détection et comment elles motivent le développement de notre *framework* de détection embarqué, *OASIS*.

3.1 Présentation du Bluetooth Low Energy

Le protocole BLE est une variante du protocole Bluetooth, visant essentiellement à réduire sa consommation électrique et la complexité de sa pile protocolaire. Dans cette sous-section, nous introduisons les pré-requis techniques nécessaires pour comprendre à la fois les attaques de bas niveau liées à la conception de ce protocole et notre approche de détection embarquée.

La couche physique du BLE utilise une modulation dite *Gaussian Frequency Shift Keying* (GFSK) avec un débit de données de 1 Mbps ou 2 Mbps. Elle fonctionne dans la bande ISM 2,4 GHz, qui est divisée en 40 canaux de communication, chacun utilisant une bande passante de 2 MHz. Trois de ces canaux, nommés canaux *d'advertising* et numérotés de 37 à 39, sont dédiés au mode *advertising*, permettant aux équipements de diffuser des paquets dits *advertisements*, visant à notifier leur présence aux autres équipements de l'environnement. Les canaux restants, nommés canaux de données, sont utilisés en mode *connecté*, un autre mode de fonctionnement basé sur une topologie

Maître/Esclave, permettant à deux équipements d'établir une connexion pour échanger des données. Chaque nœud est lié à un rôle (nommé rôle*GAP*) en fonction de ses capacités : a) un *Broadcaster (ou Advertiser)* est uniquement capable de transmettre des *advertisements*, b) un *Scanner (ou Observer)* est uniquement capable de scanner les *advertisements*, c) un *Peripheral* est capable d'annoncer sa présence à l'aide d'*advertisements* et peut accepter les connexions entrantes et d) un *Central* est capable de scanner les *advertisements* et d'initier une connexion avec un *Peripheral*.

En mode *advertising*, un équipement émet régulièrement des trames d'*advertisements* sur les 3 canaux d'*advertisements*. Le temps entre chaque *advertising event* (qui représente un cycle complet de sauts sur les trois canaux) est défini par l'*advInterval*, un entier choisi par l'équipement et multiple de 0,625 ms, et par un délai aléatoire nommé *advDelay* entre 0 et 10ms, qui est généré automatiquement par la couche liaison pour chaque *advertisement event*. Après chaque transmission, l'équipement qui émet les *advertisements* écoute brièvement le canal, à la recherche de potentielles *Scan Request* ou *Connection Request* transmises par un autre équipement.

Si une *Connection Request* est reçue, les deux équipements entrent en mode *connecté*, dans lequel les équipements impliqués appliquent un algorithme de saut de canal et sautent sur des canaux de données. L'équipement qui initie la connexion agit comme un *Master* tandis que l'équipement acceptant la connexion entrante agit comme un *Slave*. Plusieurs paramètres inclus dans la *Connection Request* sont fournis en entrées de l'algorithme de saut de canal sélectionné. Les équipements communiquent pendant des intervalles de temps nommés *connection events* : d'abord le *Master* transmet une trame au *Slave*, puis le *Slave* attend 150us et transmet sa propre trame au *Master*. Si les équipements n'ont pas de données à échanger, ils transmettent des trames avec une charge utile de longueur nulle pour faciliter la synchronisation. La durée d'un *connection event* est définie par un paramètre nommé *Hop Interval*, qui est un multiple entier de 1,250 ms inclus dans la charge utile du *Connection Request*. Lorsqu'un *connection event* est terminé, les équipements sautent sur le canal suivant en fonction de l'algorithme de saut de canal sélectionné avec les paramètres fournis. Cette conception est efficace pour éviter les interférences potentielles, qui sont courantes dans la bande ISM 2,4 GHz du fait de sa sur-utilisation. Notons qu'un mécanisme complémentaire nommé *Channel Map* permet de mettre en liste noire ou blanche les canaux de données.

Chaque trame de la couche liaison commence par un préambule d'1 octet suivi d'un champ de 4 octets nommé *Access Address* utilisé pour la synchronisation. Chaque trame comprend également un en-tête, une charge utile et un contrôle de redondance cyclique (CRC) de 3 octets, permettant de vérifier l'intégrité de la trame. L'algorithme de calcul du CRC est configuré à l'aide d'une valeur d'initialisation, qui est une constante prédéfinie en mode *advertising* (0x555555) et une valeur aléatoire choisie par l'initiateur et transmise dans la charge utile de la *Connection Request* en mode *connecté*. Une trame corrompue peut être détectée en comparant le CRC calculé avec le CRC inclus à la fin de la trame: elle sera alors ignorée par la couche liaison du récepteur.

La spécification BLE introduit plusieurs mécanismes de sécurité permettant de protéger efficacement la confidentialité et l'intégrité des communications. Par exemple, la connexion peut être chiffrée à l'aide d'une clé dérivée de la *Long Term Key*, qui est négociée lors de la phase de *bonding*. La spécification introduit plusieurs méthodes permettant aux dispositifs communicants de négocier des paramètres de sécurité, tels que l'entropie de la clé ou les capacités d'entrée/sortie. Cependant, certaines de ces méthodes sont connues pour être vulnérables ou faibles[4, 27], et il est malheureusement courant de rencontrer des équipements commerciaux qui n'ont mis en œuvre aucune mesure de sécurité.

3.2 Détection des attaques bas niveau

Dans cette sous-section, nous présentons brièvement six attaques bas niveau critiques ciblant le protocole BLE, et présentons notre stratégie de détection pour chacune. Nous nous sommes concentrés sur les attaques liées à la conception du protocole lui-même et qui ne peuvent pas être facilement corrigées sans modifier la spécification.

3.2.1 GATTacker et BTLEJuice: attaques de l'homme du milieu

Présentation de l'attaque : Deux stratégies principales ont été développées afin de réaliser une attaque Man-in-the-Middle ciblant une connexion BLE. Elles sont toutes deux basées sur une stratégie de *spoofing* ciblant les *advertisements* transmis par un *Peripheral* avant l'initiation de la connexion, même si elles adoptent des approches différentes pour effectuer cette opération. *GATTacker* [19] exploite le fait qu'un *Central* essayant d'initier une connexion avec un *Peripheral* transmet sa *Connection Request* juste après la réception d'un paquet d'annonce transmis par le *Peripheral*. En conséquence, l'approche *GATTacker* transmet des paquets d'*advertisement* usurpés plus rapidement que le *Peripheral* légitime pour maximiser la probabilité de recevoir la *Connection Request* à sa place. Une fois que le *Central* est connecté au faux *Peripheral* de l'attaquant, ce dernier initie une connexion à l'aide d'un deuxième dongle BLE avec le *Peripheral* légitime pour établir l'attaque Man-in-the-Middle. L'approche *BTLEJuice* [9], quant à elle, est basée sur le fait qu'un *Peripheral* cesse de transmettre des *advertisements* lorsqu'il est impliqué dans une connexion. L'attaquant utilise un premier dongle pour établir une connexion avec le *Peripheral* cible, le forçant à cesser de diffuser ses *advertisements*. Ensuite, l'attaquant utilise un deuxième dongle pour exposer un *Peripheral* usurpé, attendant qu'un *Central* initie une connexion.

Stratégies de détection : Notre stratégie pour détecter *GATTacker* est basée sur l'observation qu'un *Peripheral* transmettant des *advertisements* doit suivre un algorithme de saut de canal spécifique, qui dépend de deux paramètres (l'*advertising delay* et l'*advertising interval*, comme nous l'avons mentionné dans la sous-section 3.1). Si un attaquant transmet des *advertisements* simultanément, un nœud surveillant les canaux d'*advertisement* en tant que *Scanner* ou *Central* devrait recevoir à la fois les *advertisements* légitimes et usurpés et être capable de détecter que les paquets reçus ne sont pas conformes à la spécification du protocole, indiquant par là même la présence d'un nœud malveillant.

Afin de détecter cette situation, nous estimons d'abord l'*advertising interval* pour chaque équipement transmettant des *advertisements*. Cette estimation est basée sur une fenêtre glissante durant laquelle est calculée la durée entre deux *advertisements* consécutifs d'un même équipement reçus sur le même canal : la valeur minimale de ces durées dans la fenêtre est notre estimation de l'*advertising interval* (utiliser cette valeur minimale permet de minimiser l'impact de l'*adverstising delay* qui est une valeur aléatoire). Ensuite, nous fixons un seuil de détection à une valeur correspondant à l'*adversiting interval* moins la valeur maximale d'*avdertising delay*, ce qui représente le pire cas légitime. Chaque fois qu'un nouveau paquet est reçu, une nouvelle estimation est calculée avec cette méthode, et si la valeur calculée est inférieure au seuil de détection, une alerte est levée indiquant la présence d'un nœud malveillant.

L'attaque *BTLEJuice* est plus difficile à détecter car un nœud surveillant un canal d'*advertising* n'a aucune garantie d'observer la *Connection Request* transmise par l'attaquant. En conséquence, nous choisissons d'adopter une autre stratégie, permettant au *Peripheral* cible de détecter sa propre usurpation par un attaquant. Lorsqu'une connexion est établie, le *Peripheral* maintient la connexion et analyse simultanément les *advertisements*. Au cours de cette opération de scan, le *Peripheral* vérifie si sa propre adresse est incluse dans les paquets d'*advertisements* observés. Si une telle

situation est détectée, cela signifie qu'un attaquant tente d'effectuer l'attaque *BTLEJuice* et une alerte est déclenchée.

Même si ces stratégies permettent une détection efficace, elles présentent toutefois certaines limites qu'il convient de souligner. La détection de *GATTacker* doit pouvoir estimer correctement l'*advertising interval* légitime avant de pouvoir détecter un nœud attaquant: par conséquent, la détection nécessite que le dispositif de surveillance ait pu parcourir sa fenêtre glissante pour estimer l'intervalle avant le début de l'attaque. Cette phase d'apprentissage pourrait probablement être réduite ou supprimée dans un environnement contrôlé, où les équipements réalisant la détection pourraient utiliser des *advertising intervals* préc-configurés pour chaque *Peripheral* surveillé. De même, la détection de *BTLEJuice* nécessite que le *Peripheral* cible soit capable de maintenir simultanément une connexion et de scanner les *advertisements*. La plupart des contrôleurs devraient être capables d'effectuer ces opérations simultanément, mais cela peut être problématique pour certains contrôleurs spécifiques qui n'implémentent qu'un sous-ensemble des rôles *BLE* (par exemple, certaines piles protocolaires de Nordic SemiConductors implémentent uniquement le rôle *Peripheral* et ne peuvent donc pas effectuer l'opération de scan simultanément).

3.2.2 Attaques de brouillage continu

Présentation de l'attaque : Un problème de sécurité courant observé lors de l'utilisation d'un protocole de communication sans fil est lié au fait que le support est ouvert par conception, ce qui permet à un attaquant de porter atteinte à la disponibilité des communications en attaquant le lien lui-même. L'une des stratégies les plus simples pour effectuer une telle attaque de déni de service consiste à transmettre un signal de brouillage, qui interfère avec le trafic légitime et génère des CRC invalides, forçant les nœuds légitimes à ignorer les trames corrompues.

Plusieurs stratégies de brouillage existent [8, 28], mais nous nous concentrons ici sur une attaque de brouillage simple visant à attaquer les canaux d'*advertising* en transmettant un signal de forte puissance sur les fréquences correspondantes. Les canaux d'*advertising* sont en effet une cible intéressante pour un attaquant, car ils sont à la fois utilisés pour indiquer la présence d'équipements et pour initier des connexions. De ce fait, un brouilleur continu ciblant ces canaux peut avoir un impact conséquent sur les nœuds présents dans l'environnement, en interrompant toute tentative d'initiation de connexion ou tentative de scan. Cette approche offensive est également intéressante du point de vue des coûts, car elle ne nécessite pas de suivre l'algorithme de saut de canal d'une connexion et cible des canaux prédéfinis, réduisant considérablement le coût et la complexité du brouilleur.

Stratégie de détection : Une solution évidente pour détecter une telle attaque serait d'analyser la couche physique pour détecter le signal de brouillage. Cependant, cette solution ne peut pas être mise en place facilement par un système de détection d'intrusion intégré dans des nœuds légitimes car la plupart des contrôleurs BLE existants ne permettent pas un accès direct à la couche physique. Cela nécessiterait également une analyse complexe car l'attaquant a très peu de contraintes concernant la conception du brouilleur, et n'est pas obligé de se conformer à la spécification du protocole. Une autre façon de détecter cette attaque repose sur l'idée de surveiller ses conséquences au niveau de la couche liaison. En effet, une attaque de brouillage réussie provoque des corruptions de paquets, entraînant des CRC invalides. Comme tout équipements conforme à la spécification est capable de vérifier la validité du CRC d'un paquet, notre stratégie de détection est basée sur cette vérification : chaque seconde, les nœuds implémentant les rôles *Scanner* ou *Central* (c'est-à-dire qui sont capables de scanner des *advertisements*) calculent le nombre de paquets reçus sans corruption d'intégrité par seconde sur un canal donné, les trames avec un CRC invalide étant ignorées. Si cette valeur est égale

à zéro pendant plus d'un certain nombre de mesures (fixé à 5 dans nos expérimentations) pour un canal donné, on considère que le canal est considéré comme brouillé et une alerte est levée.

Notons que cette stratégie détecte un environnement sans aucun trafic comme un faux positif : même si cette situation se produit rarement, elle doit être prise en compte dans une optique défensive. Une façon de faire la distinction entre cette situation légitime et une attaque serait d'estimer à la fois le nombre de paquets corrompus et non corrompus par seconde, et de ne déclencher l'alerte que si le nombre de paquets non corrompus par seconde est égal à zéro alors que le nombre de paquets corrompus par seconde est différent de zéro. Cependant, cette variante pourrait conduire à des faux négatifs si l'attaquant brouille le préambule des paquets, n'entraînant aucune réception valide par l'IDS embarqué, et donc aucun déclenchement d'alerte. Si l'environnement peut être maîtrisé, l'insertion d'un équipement de type *Advertiser* non connectable pourrait constituer un bon compromis, permettant d'appliquer la première stratégie sans risque de faux positifs.

3.2.3 L'attaque BTLEJack

Présentation de l'attaque: *BTLEJack* [11] est une autre attaque qui peut avoir un impact conséquent sur la disponibilité. Cette attaque, présentée par D. Cauquil, consiste à brouiller une connexion établie ou à usurper le rôle de *Master* dans certaines circonstances. L'attaquant se synchronise d'abord avec une connexion établie, puis transmet un signal de brouillage lorsque le *Slave* envoie une réponse au *Master* à chaque *connection event*. L'attaque exploite un mécanisme de compteur visant à détecter une perte de lien en incrémentant la valeur de ce compteur à chaque paquet manqué ou invalide. Lorsque ce compteur atteint un seuil prédéfini, le *Master* considère la connexion comme perdue et quitte la connexion, ce qui permet ainsi à l'attaquant de l'interrompre ou, dans le pire des cas, d'usurper le rôle de *Master* si le *Slave* ne quitte pas la connexion immédiatement après la déconnexion du *Master*.

Stratégie de détection: Du point de vue d'un nœud *Central*, la détection de cette attaque peut être effectuée facilement : contrairement à une perte de connexion normale, le *Central* reçoit des trames comprenant un CRC invalide sur plusieurs événements de connexion consécutifs lors d'une attaque, alors que dans un scénario légitime, il ne devrait recevoir aucun paquet. Cette situation a une très faible probabilité de se produire dans une situation légitime, l'algorithme de saut de canal assurant l'utilisation de plusieurs canaux répartis sur l'ensemble de la bande ISM 2,4GHz. Le stratégie de détection consiste donc à déclencher une alerte lorsque le compteur de trames consécutives reçues avec corruption d'intégrité atteint la valeur du compteur de connexions moins un (juste avant le succès de l'attaque).

3.2.4 L'attaque KNOB

Présentation de l'attaque : L'attaque *KNOB*, présentée par D. Antonioli et al.[4], permet à un attaquant en situation d'homme au milieu d'injecter une faible valeur d'entropie lors du processus d'appairage. En effet, le processus d'appairage inclut un protocole dédié à la négociation d'entropie, permettant à chaque équipement impliqué d'indiquer combien d'octets d'entropie peuvent être utilisés lors de la génération de la clé. Par conséquent, un attaquant peut effectuer une attaque visant à diminuer l'entropie, en fixant ce nombre d'octets à 7 octets (valeur minimum autorisée) au lieu de 16 qui est la valeur standard utilisée en BLE. En conséquence, la clé peut facilement être découverte par une attaque de force brute, ce qui compromet la sécurité des futures communications entre les équipements concernés.

Stratégie de détection: Cette attaque peut être détectée à la fois par un *Central* ou un *Peripheral* en utilisant une simple stratégie passive. Lorsqu'une *Paring Request* est reçue (le paquet utilisé pour négocier la valeur d'entropie), la valeur de cette entropie est extraite de la charge utile du paquet et une alerte est déclenchée si la valeur est inférieure à 10 octets. Même si le protocole permet techniquement d'utiliser légitimement une valeur aussi faible, considérer qu'un équipement ne devrait pas être autorisé à utiliser une valeur d'entropie suffisamment faible pour permettre une attaque par force brute est une contrainte qui nous semble totalement justifiée du point de vue de la sécurité.

3.2.5 L'attaque InjectaBLE

Présentation de l'attaque: Cette attaque est une attaque récente par injection ciblant les communications BLE baptisée InjectaBLE[13]. L'attaque détourne une fonctionnalité permettant à deux équipements de communiquer même en cas d'une éventuelle dérive d'horloge: lorsqu'un *Peripheral* passe en mode réception pour recevoir un paquet du *Central* lors d'un *connection event*, il écoute pendant une courte fenêtre (nommée *window widening*) après et avant l'instant théorique, permettant à un attaquant d'exploiter une *race condition* et d'injecter un paquet malveillant avant le *Central* légitime. Cette attaque est critique, surtout si la connexion n'est pas chiffrée, car elle permet d'usurper n'importe quel rôle de la communication ou d'effectuer un Man-in-the-Middle par l'injection de trames judicieusement choisies.

Stratégie de détection: Cette attaque peut être détectée par le *Peripheral* ciblé lui-même en calculant en permanence l'intervalle entre deux paquets reçus consécutifs. On est ainsi capable de détecter si un paquet a été injecté en comparant le dernier intervalle à l'intervalle de connexion légitime : si l'intervalle est inférieur à l'intervalle théorique moins un seuil estimé empiriquement, on considère la trame comme malveillante, provoquant le déclenchement d'une alerte. Notons que toutefois, cette stratégie peut conduire à des faux positifs si les équipements utilisent des horloges avec une dérive importante, même si nos expérimentations ont conduit à de très bons résultats avec cette heuristique.

3.3 Données nécessaires à la détection

Les stratégies de détection que nous avons décrites ci-dessus nous donnent un bon aperçu des données (appelées *caractéristiques* dans la suite de cet article) dont il faut disposer pour implémenter les modules de détection. Les parties suivantes de la pile BLE doivent être instrumentés:

- **mécanismes de traitement des paquets :** la majorité de nos stratégies a besoin d'accéder aux paquets au niveau de la couche liaison, en particulier les paquets reçus. Les *advertisements* et les paquets de données doivent être collectés, avec certaines meta-données pertinentes comme la validité du CRC ou le RSSI.
- **mécanismes de gestion du temps :** nous avons besoin de collecter les timestamps avec la meilleure précision possible de façon à estimer les intervalles entre les paquets, nécessaires à l'implémentation des modules de détection des attaques *GATTacker* ou *InjectaBLE*. Nous avons aussi besoin d'exécuter du code régulièrement et indépendamment des instants de réception des paquets, par exemple pour calculer le nombre de paquets valides par seconde (pour notamment la détection du brouillage continu).

- **mécanismes de gestion des connexions et de l'équipement :** certaines stratégies de détection utilisent des données qui sont gérées par le contrôleur et qui sont liées aux connexions (notamment le *connection interval* pour la détection de l'attaque *InjectaBLE*) ou à l'équipement lui-même (notamment l'adresse BD pour la détection de l'attaque *BTLEJuice*). Comme certaines de nos stratégies de détection sont limitées à certains rôles spécifiques, nous avons aussi besoin de connaître en temps réel la valeur du rôle courant de l'équipement instrumenté, et déclencher l'exécution de code lors de l'occurence d'un événement (comme l'initiation d'une connexion).

- **opérations de haut niveau :** nous avons besoin d'instrumenter des opérations de haut niveau du contrôleur, par exemple pour déclencher le mode *scan* lorsqu'une connexion est établie pour la détection de l'attaque *BTLEJuice*.

Notons que la mise en œuvre de ces mécanismes peut être très hétérogène selon la pile protocolaire utilisée. Pour éviter le développement de multiples modules de détection selon la pile, il est nécessaire de disposer d'un *framework* générique fournissanr des *wrappers* destinés à faciliter l'instrumentation des piles et exposant une API homogène. La suite de cet article est consacré à la conception et à l'évaluation de ce *framework*.

4 Conception du Framework

Dans cette section, nous décrivons la conception d'*OASIS*, un *framework* générique et modulaire permettant de patcher des contrôleurs BLE pour y intégrer des heuristiques de détection, telles que celles présentées dans la section précédente. Nous présentons tout d'abord les principales lignes directrices qui ont guidé son élaboration. Ensuite, nous décrivons son architecture globale et la structure du code de détection généré. Enfin, nous décrivons brièvement l'implémentation de ses principaux composants et un cas typique d'utilisation.

4.1 Principes fondamentaux de conception

Dans la sous-section 3.3, nous avons mis en évidence les exigences minimales nécessaires pour mettre en œuvre les stratégies de détection mentionnées précédemment. Cependant, de nombreuses implémentations de contrôleurs sont propriétaires et non documentées. La conséquence directe de cette situation est que nous ne pouvons pas instrumenter directement le code source, ce qui nous oblige à trouver un moyen d'interagir avec la pile en patchant le binaire du firmware lors de l'exécution tout en executant notre propre code sans perturber le comportement légitime de la pile.

Cette situation a motivé le développement d'un *framework* générant du code embarqué qui doit pouvoir fonctionner indépendamment du contrôleur, sans altérer son comportement normal. Cela implique de bien choisir les fonctions instrumentées pour éviter d'introduire des retards dans les composants sensibles au temps, mais aussi de trouver un moyen d'injecter notre code et nos données en mémoire sans impacter l'exécution du contrôleur. Notre *framework* doit également permettre à un développeur d'implémenter facilement un nouveau module de détection en lui fournissant un environnement facile d'utilisation pour allouer de la mémoire, collecter les caractéristiques utiles pour les modules de détection ou réagir à un événement spécifique.

Les contrôleurs sont également hétérogènes et ne peuvent être instrumentés sans écrire de code spécifique pour chacun. Cependant, un module de détection décrit une logique indépendante de l'implémentation sous-jacente du contrôleur, et le code correspondant ne doit être écrit qu'une seule fois. Par conséquent, chaque wrapper spécifique à une cible doit exposer une API homogène,

facilitant le développement des composants indépendants de la cible. Par conséquent, l'un des principes clés qui a guidé la conception de notre *framework* est la **généricité**.

De plus, certains contrôleurs n'implémentent qu'un sous-ensemble de la spécification BLE. Par exemple, certains contrôleurs orientés IoT n'implémentent que le rôle *Peripheral*. Comme certaines de nos stratégies de détection ne fonctionnent que si l'équipement utilise un rôle spécifique, il n'est pas pertinent d'intégrer systématiquement l'ensemble du *framework*. Il est également important de prendre en compte les fortes contraintes en termes de temps et de mémoire liées à l'approche embarquée. Aussi la **modularité** doit être une ligne directrice fondamentale de la conception de notre *framework*. Enfin, étendre le *framework* pour ajouter une nouvelle cible ou un nouveau module de détection doit être aussi simple que possible.

4.2 Logiciel de détection embarqué

Le *framework OASIS* permet de générer un logiciel embarqué et de l'injecter dans la mémoire de la puce. Ce logiciel embarqué instrumente le contrôleur cible en patchant des fonctions spécifiques afin d'extraire des caractéristiques utiles, puis transmet ces caractéristiques aux modules de détection sélectionnés, qui les analysent et éventuellement déclenchent une alerte si une attaque est détectée. Même si le code interagit avec la pile BLE, il est conçu pour s'exécuter sans interférer avec le comportement légitime : par conséquent, le code utilise et gère son propre espace mémoire, indépendamment de l'application principale.

Le code embarqué comprend trois composants principaux, comme illustré dans la figure 1 : un **wrapper** spécifique à la cible, un **core** et un **ensemble de modules de détection**. Ils sont décrits dans les sections suivantes.

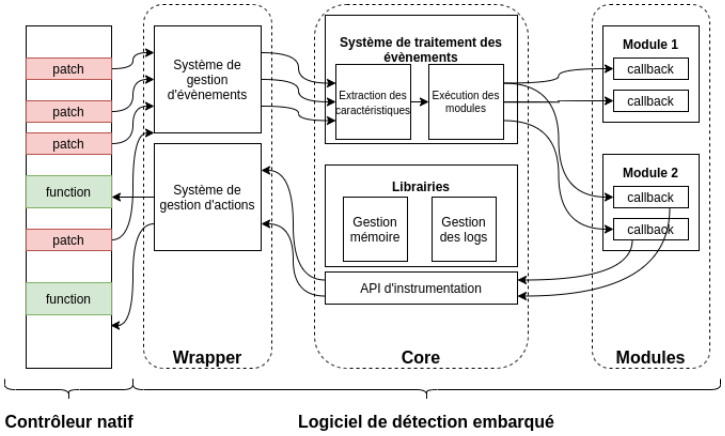

Figure 1: Vue d'ensemble du logiciel embarqué

4.2.1 Wrapper

Le *wrapper* est spécifique à chaque cible et permet d'interagir avec le contrôleur. Il est composé de deux principaux éléments : 1) un *système de gestion d'événements* permettant de réagir à des événements spécifiques (par exemple, réception de paquets, transmission de paquets, initiation de connexion, ...) et d'extraire toutes les caractéristiques de bas niveau disponibles dans le contrôleur,

et 2) un *système de gestion d'actions*, permettant de déclencher des actions spécifiques dans le contrôleur (par exemple, envoyer un événement à l'hôte, entrer dans un état spécifique,...).

Le *système de gestion d'événements* est composé d'un ensemble de fonctions *wrappers* correspondant aux événements surveillés. Il instrumente certaines instructions spécifiques de la pile BLE pour rediriger le flux d'exécution vers une fonction trampoline, qui sauvegarde le contexte et appelle le *wrapper* correspondant. Une fois le *wrapper* exécuté, la fonction trampoline restaure le contexte, exécute l'instruction modifiée par le patch et redirige le flux d'exécution vers l'instruction suivante de la pile protocolaire. Ce mécanisme permet d'appeler le *wrapper* correspondant lorsqu'un événement spécifique se produit. Le *wrapper* extrait alors toutes les caractéristiques disponibles et les propage au *système de traitement d'événements* implémenté dans le composant *Core*.

Le *système de gestion d'actions* est composé d'un ensemble de fonctions permettant de déclencher une action spécifique dans le contrôleur instrumenté. Selon la pile instrumentée, il peut effectuer un appel de fonction, simuler une commande HCI transmise par l'hôte ou modifier une variable dans la mémoire du contrôleur. Ce composant est le seul qui est spécifique à chaque cible : par conséquent, chaque *wrapper* implémenté expose une API similaire, permettant aux composants indépendants de la cible d'interagir avec le contrôleur de manière standardisée et unifiée.

4.2.2 Core

Le *Core* est le composant central impliqué dans le logiciel de détection. Il est composé d'un *système de traitement d'événements*, d'un *ensemble de bibliothèques* et d'un *système d'instrumentation*.

Le *système de traitement des événements* gère les différents événements surveillés par le logiciel de détection. Lorsque le *wrapper* génère un événement spécifique, le *Core* collecte les caractéristiques extraites par le *wrapper* et en déduit éventuellement d'autres complémentaires (par exemple, le *Core* peut déduire le *advInterval* utilisé par un *Advertiser* ou un *Peripheral* à partir des *timestamps* des *advertisements* reçus de cet objet). Ensuite, le *système de traitement d'événements* propage l'événement, ainsi qu'une structure contenant les caractéristiques collectées, aux modules de détection intégrés au sein du logiciel embarqué en appelant les *callbacks* correspondants.

Le *Core* expose également un *système d'instrumentation*, qui peut être utilisé par les modules de détection pour interagir avec le contrôleur. Ce système d'instrumentation propage les appels de fonction au *wrapper*, permettant au contrôleur d'entrer dans un état spécifique ou de déclencher une action de manière générique. Il fournit également diverses *bibliothèques* facilitant le développement des modules. Le *Core* expose sa propre librairie de gestion de mémoire, permettant d'allouer et de libérer dynamiquement de la mémoire sans interférer avec la gestion native de la mémoire (le logiciel de détection gère son propre tas indépendant), une implémentation de *hashmap* et un système de journalisation, permettant d'envoyer des alertes de détection à l'hôte.

4.2.3 Modules de détection

Les modules de détection mettent en œuvre les stratégies de détection : ils sont généralement en charge d'analyser les caractéristiques fournies par le composant *Core* pour détecter les attaques. Ils peuvent également déclarer des *callbacks* qui sont exécutés lorsqu'un événement spécifique se produit, par exemple lorsqu'un paquet est reçu ou qu'une connexion est établie. Ils ont également accès aux caractéristiques collectées et déduites à l'aide d'une structure spécifique, et peuvent déclencher divers comportements via l'API d'instrumentation.

Chaque module est indépendant, et peut être considéré comme un logiciel de détection embarqué à part entière. Grâce à cette conception, l'utilisateur peut choisir les modules qu'il souhaite inclure dans le logiciel de détection embarqué, et facilement intégrer de nouvelles heuristiques de détection en développant ses propres modules. Un système de dépendances permet également de compiler et

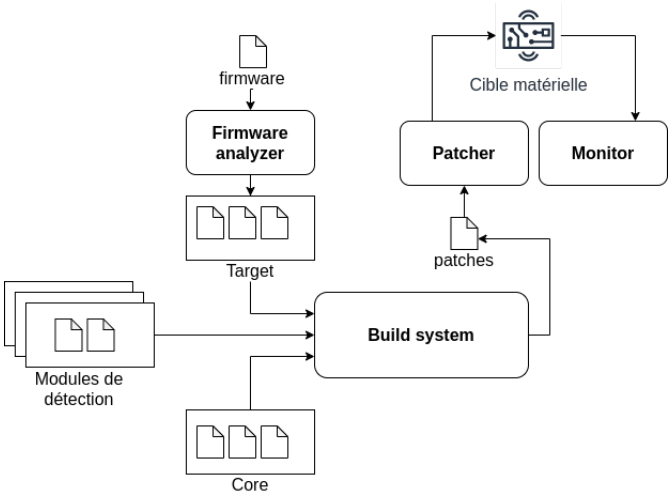

Figure 2: Architecture du framework

de charger en mémoire uniquement un sous-ensemble des fonctionnalités du *framework*, en fonction des besoins des modules sélectionnés. Cette fonctionnalité est particulièrement importante compte tenu des contraintes de temps et de mémoire inhérentes à une approche embarquée.

4.3 Architecture du framework

Le *framework OASIS* permet de générer le logiciel de détection présenté dans la section précédente et de l'injecter en mémoire. Le *framework* comprend 4 composants principaux, comme illustré dans la Figure 2: le **Firmware analyzer**, le **Build system**, le **Patcher** et le **Monitor**. Nous les décrivons dans les sections suivantes.

4.3.1 Firmware analyzer

Pour instrumenter un contrôleur spécifique, le *framework* s'appuie sur un ensemble de code source et de fichiers de configuration propres à une cible. Ces fichiers contiennent toutes les informations nécessaires au *framework* pour patcher le firmware du contrôleur, injecter le code du logiciel de détection dans la mémoire et interagir avec le contrôleur. Une cible typique est décrite par les fichiers suivants :

- **target.conf :** un fichier de configuration contenant des informations requises par le *framework* telles que l'architecture matérielle de la puce ou le *backend* à utiliser pour interagir avec la cible.
- **patch.conf :** un fichier de configuration fournissant une liste d'instructions du firmware qui peuvent être patchées pour capturer des événements spécifiques (réception de paquets, connexions, ...).
- **wrapper.c :** le code source du wrapper qui permet de fournir une API d'instrumentation unifiée au *Core*.

- **functions.ld :** une liste de quelques fonctions inclues dans le firmware du contrôleur qui sont utilisées par le *wrapper*. Ce fichier est fourni en entrée du *linker*.

- **linker.ld :** un script d'édition des liens qui décrit les zones mémoire disponibles pour injecter le logiciel de détection.

L'obtention des informations nécessaires à la génération de ces fichiers nécessite généralement d'effectuer une rétro-ingénierie du firmware du contrôleur, la plupart d'entre eux étant propriétaires et non documentés. Ce processus étant délicat et sujet aux erreurs lorsqu'il est effectué manuellement, le rôle du *firmware analyzer* est d'automatiser cette tâche de rétro-ingénierie et la génération de fichiers spécifiques à la cible correspondante.

Le processus est divisé en deux étapes principales. Le premier est dédié à la rétro-ingénierie du *firmware* fourni tandis que le second utilise les informations collectées pour générer les fichiers source et de configuration propres à la cible. L'étape de rétro-ingénierie est principalement basée sur une analyse statique du binaire du *firmware* qui tente d'identifier les fonctions, variables et structures pertinentes au moyen d'expressions régulières. Même si cette analyse dépend de l'architecture du contrôleur, nous avons observé que, pour un type de contrôleur donné, différents *firmwares* peuvent présenter de nombreuses similitudes, principalement liées à la réutilisation de code, ce qui nous permet d'automatiser l'analyse de plusieurs *firmware* partageant la même architecture de contrôleur.

De même, diverses stratégies d'analyse permettent d'identifier des zones mémoire utilisables pour injecter le code du logiciel de détection sans perturber l'exécution du contrôleur. Notons que nous avons travaillé sur trois architectures différentes de contrôleurs, dont deux sont décrites dans la section 5, mais le *framework* pourrait facilement être étendu pour ajouter le support de nouvelles architectures de contrôleurs.

Une fois le firmware analysé, les fichiers de configuration et de code source de la cible nécessaires à son instrumentation sont générés. Les fonctions liées à certains événements spécifiques, comme la réception de paquets, sont partiellement désassemblées pour identifier une instruction qui peut être patchée afin de rediriger le flux d'exécution vers le *wrapper* lorsque l'événement se produit : les instructions identifiées sont alors utilisées pour générer le fichier **patch.conf**. Les autres fonctions et données qui ne sont pas liées à un événement spécifique sont également utilisées pour générer les fichiers **wrapper.c** et **functions.ld**. Enfin, les zones mémoire identifiées sont utilisées pour générer à la fois les fichiers **linker.ld** et **patch.conf**.

4.3.2 Build system

Une fois générée, la cible est fournie en entrée du *build system*, avec les composants logiciels indépendants de la cible (par exemple, le *core* et les modules de détection sélectionnés). Le *build system* est composé d'un ensemble de scripts permettant de générer la liste finale des *patchs* et blocs binaires (composés de données ou de code) qui seront injectés dans la mémoire, en utilisant des outils standards tels que le compilateur et l'assembleur GNU gcc.

Le *build system* exécute les étapes suivantes:

- **compilation des modules de détection :** chaque module sélectionné est compilé de façon séparée, de façon à générer le bloc binaire correspondant.

- **gérération des callbacks des modules :** pour chaque module sélectionné, *le build system* dresse la liste des *callbacks* déclarés par le module. Du code C est ensuite généré, contenant des pointeurs de fonctions pour les *callbacks* associés à chaque événement, permettant ainsi au *Core* de rediriger l'exécution au *callback* approprié lors de l'occurrence d'un événement.

- **génération des fonctions trampoline :** pour chaque *patch* nécessaire à l'instrumentation de la cible, le *build system* genère une fonction trampoline qui sauvegarde le contexte d'exécution, le restaure et exécute l'instruction qui a été ignorée.
- **Compilation et édition des liens :** l'ensemble du logiciel de détection (le *core*, le *wrapper*, les modules de détection, le code C généré pour les *callbacks* et les fonctions trampolines) est compilé et l'édition des liens est réalisée . Un mécanisme de dépendances permet de ne compiler que les composants nécessaires si les modules sélectionnés n'utilisent pas certains composants.
- **Extraction des symboles :** chaque symbole contenu dans le binaire généré est extrait et stocké dans un fichier temporaire contenant le nom du symbole, son adresse et sa valeur.
- **génération des blocs binaires et patchs :** la liste final des blocs binaires et *patchs* est générée, en utilisant les symboles extraits du binaire (ou blocs binaires) et les *patchs* sélectionnées qui doivent être appliqués au *firmware* du contrôleur.

4.3.3 Patcher et monitor

Lorsque la liste des *patchs* et blocs binaires est générée, le *framework* peut les injecter en mémoire grâce au *patcher*. Selon le type de contrôleur utilisé, il peut utiliser un *backend* différent pour exécuter le processus de *patching*: par exemple, les piles Broadcom et Cypress sont instrumentées à l'aide d'*InternalBlue*[23], tandis que les piles Zephyr et Nordic Semiconductors sont instrumentées à l'aide d'*openOCD*.

De même, le *framework* réutilise ces *backends* pour faciliter le débogage et collecter les journaux et les alertes de détection. En utilisant la liste des *patchs* et blocs binaires, le *monitor* est capable de mapper un symbole donné à son adresse en mémoire, facilitant ainsi le débogage du logiciel de détection.

4.4 Utilisation du framework

Le *framework* peut facilement être utilisé ou étendu grâce aux différents composants présentés précédemment. Un *workflow* typique d'utilisation comprend les étapes suivantes :

- **Génération des fichiers propres à la cible (optionnel) :** si les fichiers propres à la cible n'existent pas (le *framework* est fourni avec un ensemble de fichiers propres à différentes cibles), l'utilisateur peut récupérer le *firmware* et utiliser le *firmware analyzer* pour réaliser automatiquement sa rétro-ingénierie et générer les fichiers correspondant à la cible.
- **Sélection des modules de détection :** l'utilisateur peut facilement sélectionner les modules de détection qu'ils souhaite inclure dans le logiciel final de détection, ou écrire ses propres modules en utilisant du C standard. Les autres composants logiciels ne nécessitent aucune modification si les caractéristiques collectées sont suffisantes pour réaliser la détection.
- **Génération du logiciel de détection :** l'utilisateur peut alors exécuter le *build system* pour générer le logiciel de détection, et il peut l'injecter en memoire à l'aide du *patcher*.
- **Monitorer le logiciel de détection :** l'utilisateur peut deboguer le logiciel de détection ou collecter les journaux générés et les alertes en utilisant le *monitor*.

5 Instrumentation des contrôleurs

Notre travail s'est concentré sur trois pile protocolaires BLE hétérogènes et communément utilisées: la pile développée par *Broadcom/Cypress*, embarquée dans de nombreuses puces *Bluetooth* développées par ces constructeurs, le *SoftDevice* de *Nordic SemiConductors*, principalement embarqué dans leur gamme de puces destinées à l'IoT *nRF51* et *nRF52*, ainsi que la pile protocolaire BLE inclue dans le système d'exploitation embarqué et open-source *Zephyr*.

Dans cette section, nous présentons brièvement le fonctionnement des piles protocolaires propriétaires, ces dernières ayant nécessité un effort conséquent de rétro-ingénierie, ainsi que la stratégie appliquée pour les instrumenter. Une illustration globale de ces stratégies est présentée en figure 3. Pour chaque pile protocolaire analysée, nous avons procédé à une rétro-ingénierie partielle d'un ensemble représentatif de *firmwares* implémentant la pile concernée. Cela nous a permis d'identifier l'architecture logicielle, l'implémentation des caractéristiques nécessaires à la détection et détaillées en section3.3, ainsi que l'agencement mémoire.

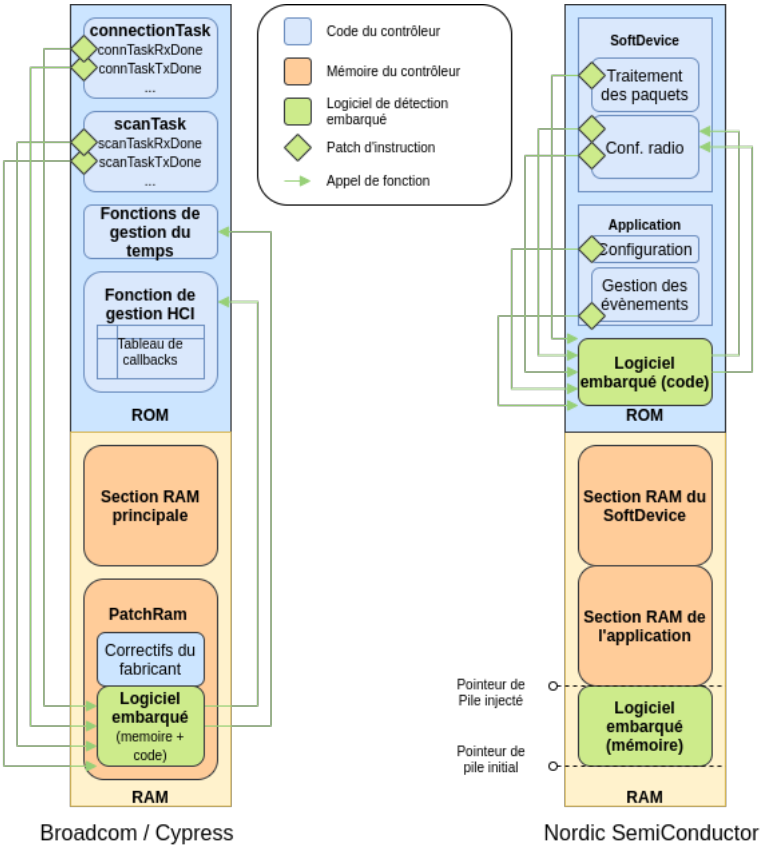

Figure 3: Intégration du logiciel embarqué dans les piles protocolaires propriétaires

5.1 Contrôleurs Bluetooth de Broadcom et Cypress

Les puces *Bluetooth* produites par *Broadcom* et *Cypress* reposent sur l'utilisation d'une pile protocolaire propriétaire basée sur l'OS temps réel *ThreadX*. Les puces concernées, basées sur un processeur *ARM Cortex M3*, sont particulièrement répandues et sont embarquées dans de très nombreux équipements, tels que des smartphones (Nexus 5, Samsung Galaxy S10/S20, ...), des ordinateurs (Raspberry Pi, ...) ou des objets connectés (FitBit Charge, ...). Bien que ces puces soit peu documentées, plusieurs travaux importants dans la communauté sécurité[15, 23, 16] ont partiellement documenté leur fonctionnement. Nous nous concentrons ici uniquement sur l'implémentation du protocole BLE, ces puces supportant également le Bluetooth BR/EDR.

Les fonctionnalités BLE sont implémentées comme des tâches, représentant un état spécifique de l'équipement (tel que *connexion*, *scan*, etc). Une tâche est composée d'un ensemble de fonctions liées à un évènement spécifique, tel que l'initialisation de la tâche, la réception ou la transmission d'un paquet, listées dans un tableau de *callbacks*. Les tâches sont gérées par un composant logiciel nommé *Bluetooth Core Scheduler*, permettant de démarrer, stopper et ordonnancer les différentes tâches en cours. Nous avons instrumenté les fonctions liées à l'initialisation et au traitement des paquets de chaque tâche BLE, nous permettant d'analyser les paquets transmis et reçus en temps réel tout en nous permettant d'identifier le rôle GAP utilisé. Nous avons également extrait de certaines fonctions destinées à la configuration du composant radio les adresses des structures utilisées pour stocker des caractéristiques pertinentes, telles que les paramètres de connexion ou l'adresse BD de l'équipement.

Un *thread* dédié gère les opérations de haut niveau, et notamment la gestion de l'interface *HCI*. Chaque commande *HCI* est traitée par le *thread* et entraîne l'exécution d'une fonction spécifique, stockée dans un tableau de pointeurs de fonctions indexé par l'*opcode* de la commande. Les évènements *HCI* sont générés par l'intermédiaire d'une fonction permettant l'allocation et l'initialisation du *buffer* d'évènement, tandis qu'une autre fonction permet sa transmission à l'hôte. Nous avons instrumenté le *thread* de traitement des commandes, nous permettant d'injecter des commandes arbitraires afin de déclencher des opérations haut niveau, ainsi que les fonctions de gestion des évènements, que nous avons détournées afin de construire notre système de journalisation destiné à transmettre les alertes à l'hôte.

Le *firmware* est stocké dans la ROM, mais les fabriquants ont intégré un mécanisme nommé *PatchRam* destiné à l'application de correctifs: une zone mémoire spécifique stockée en RAM peut être utilisée pour appliquer un nombre limité de modifications du firmware en ROM. Les correctifs du fabriquant sont écrits dans une zone dédiée de la RAM, puis une instruction spécifique du *firmware* original est modifiée pour rediriger le flot d'exécution vers la fonction mise à jour en RAM. Ces mécanismes peuvent être déclenchés à l'aide de commandes HCI non standards (dites *vendor-specific*), nous permettant de facilement détourner le processus de mise à jour pour patcher le *firmware* existant et intégrer notre propre code en mémoire. Le code et les données du logiciel de détection embarqué sont injectés dans la zone de RAM dédiée aux *patchs* constructeurs, tandis que le mécanisme *PatchRam* est utilisé pour altérer certaines instructions du *firmware* dans la ROM pour mettre en place nos *hooks*. L'outil *InternalBlue* [23] facilite considérablement ce processus et est utilisé comme *backend* par notre *framework* pour patcher et monitorer ces puces.

5.2 SoftDevice de Nordic SemiConductors

Nordic SemiConductors a conçu un contrôleur propriétaire spécifique pour ses puces BLE (notamment les familles *nRF51* et *nRF52*, basées sur des processeurs *ARM*), nommé *SoftDevice*. Ces puces sont communément utilisées pour le développement d'objets connectés, et de multiples versions du *SoftDevice* ont été développées et sont utilisées au sein de l'écosystème *IoT*.

Le *SoftDevice* est fourni par le constructeur sous la forme d'un binaire, chargé dans les parties basses de la ROM. L'application utilisateur, quant à elle, est chargée dans les parties supérieures de la ROM, et communique avec le *SoftDevice* à l'aide d'une API propriétaire non standard basée sur des appels superviseurs (ou *Supervisor calls*). Une application typique initialise le *SoftDevice*, configure les fonctionnalités *BLE* souhaitées et surveille les évènements générés par le *SoftDevice* en appelant une fonction spécifique au sein d'une boucle infinie. Le *SoftDevice* est en charge des opérations bas niveau: une fonction de traitement des paquets est appelée à chaque interruption radio lors de la réception ou de la transmission d'un paquet, identifiant l'état courant de la radio et le rôle GAP par l'intermédiaire de variables et structures internes. Nous avons également identifié un ensemble de fonctions de configurations destinées à stocker des caractéristiques telles que les paramètres de connexion au sein de structures internes du *SoftDevice*.

Nous avons instrumenté la fonction de traitement des paquets ainsi que les fonctions de configuration radio au sein du *SoftDevice*, et extrait diverses caractéristiques des structures internes précédemment identifiées. La fonction utilisée par l'application pour collecter les évènements générés par le *SoftDevice* a également été instrumentée, nous permettant de générer le bon appel superviseur pour intéragir avec le *SoftDevice* quand nous devons déclencher une action de haut niveau. Nous avons également instrumenté le point d'entrée de l'application, nous permettant d'exécuter notre routine d'initialisation destinée à initialiser la mémoire et à configurer un *timer* pour faciliter les opérations de gestion du temps.

La stratégie permettant de patcher le *firmware* et d'injecter notre code et nos données en mémoire est basée sur la modification du binaire du *firmware*. Les instructions du *firmware* à patcher sont modifiées dans le binaire lui même, puis le code et la mémoire de notre logiciel de détection embarqué sont insérés à la suite du firmware initial. Nous altérons également le vecteur d'interruption afin d'introduire une valeur d'initialisation du pointeur de pile plus basse, nous permettant de réserver une zone dédiée de la RAM afin d'éviter de potentiels conflits entre la mémoire utilisée par le logiciel de détection embarqué et celle utilisée par le *SoftDevice* et l'application. Le *firmware* modifié est chargé dans la ROM de la puce par l'intermédiaire d'*openOCD*, puis la zone correspondant aux données du logiciel de détection embarqué est copiée de la ROM à la zone de RAM réservée par la routine d'initialisation.

6 Expérimentations

Nous avons effectué différentes expériences pour évaluer notre approche de détection. Pour chaque attaque, nous avons implanté le module de détection correspondant sur plusieurs puces et généré du trafic légitime et malveillant dans un environnement réaliste pour estimer les performances de détection. Chaque expérience a été réalisée dans des conditions similaires, toutes les cartes embarquant le logiciel de détection étant connectées à une passerelle centrale collectant les résultats de la détection tout en générant régulièrement les attaques et le trafic légitime.

6.1 Conditions expérimentales

Nos expériences ont été menées sur cinq cibles différentes : une carte Raspberry Pi 3+ (équipée d'un contrôleur BCM4345C0), un smartphone Nexus 5 (équipé d'un contrôleur BCM4339), un porte-clés intelligent Gablys (équipé d'un contrôleur nRF51822), une carte de développement IoT de Cypress (équipée d'un contrôleur CYW20735), une carte de développement nRF de Nordic Semiconductor (équipée d'un contrôleur nRF51422) intégrant divers exemples issus du SDK (par exemple, *Scanner* et *Peripheral*). Ces cibles sont respectivement appelées Ra, Ne, GA, D1, D2 dans le tableau 6.1.

	Cibles				
	Ra	Ne	Ga	D1	D2
GATTacker	✓	✓		✓	✓
BTLEJuice			✓	✓	✓
Jamming	✓	✓		✓	✓
KNOB			✓	✓	✓
InjectaBLE	✓			✓	✓
BTLEJack		✓		✓	

Table 1: Cibles utilisées par expérience

Pour chaque expérience, les cibles ont été sélectionnées en fonction de leur prise en charge des rôles requis par nos modules de détection.

6.1.1 Expérience 1 - Gattacker

Les attaques ont été menées à l'aide de deux dongles *HCI* et du *framework* offensif Mirage [14] (module *ble_mitm*). Les attaques ciblent une ampoule connectée, située à deux mètres des cartes utilisées pour la détection. Nous avons réalisé 250 attaques, d'une durée aléatoire comprise entre 10 et 30 secondes. Chaque attaque était suivie d'une période de 30 secondes sans attaque, ce qui correspond donc à 250 périodes de trafic légitime. La détection étant basée sur le rôle *Scanner*, chaque carte de détection a été configurée pour effectuer une opération de *scan* pendant toute l'expérience.

6.1.2 Expérience 2 - BTLEJuice

Nous avons effectué les attaques à l'aide de deux dongles *HCI* et du *framework* offensif Mirage [14] (module *ble_mitm*) visant les cibles elles-mêmes. De même, nous avons généré des connexions légitimes représentant le trafic légitime à l'aide du module *ble_master* de Mirage. Chaque attaque dure un temps aléatoire entre 10 et 30 secondes, tandis que chaque connexion légitime est effectuée pendant 5 secondes. La détection étant basée sur l'utilisation d'un rôle *Peripheral* pouvant simultanément maintenir la connexion et scanner les canaux d'*advertising*, nous avons sélectionné des cibles supportant ces contraintes.

6.1.3 Expérience 3 - Jamming

L'attaque est menée à l'aide d'un *HackRF one* transmettant des données aléatoires sur la fréquence utilisée par l'un des trois canaux d'*advertising* (utilitaire *hackrf_transfer*). Nous avons effectué 250 attaques, ciblant un canal d'*advertising* choisi au hasard pendant une durée aléatoire comprise entre 10 et 30 secondes. Chaque attaque est suivie d'une période de 30 secondes sans attaque, correspondant aux phases de trafic légitime. Une ampoule connectée était présente pendant toute l'expérience dans l'environnement. La stratégie de détection étant basée sur le rôle *Scanner*, chaque cible a été configurée pour effectuer une opération de *scan* pendant toute l'expérience.

6.1.4 Expérience 4 - KNOB

À notre connaissance, il n'existe aucune implémentation de cette attaque *over the air*, la preuve de concept présentée dans l'article original étant implémentée sous la forme d'un patch *InternalBlue* destiné à imiter le comportement de l'attaque. Nous avons développé notre propre implémentation

over-the-air en modifiant le *framework* Mirage pour permettre la transmission d'une *Pairing Request* incluant une valeur arbitraire du champ *MaxKeySize*. Chaque cible simulait un rôle *Peripheral*.

6.1.5 Expérience 5 - InjectaBLE

L'attaque nécessite de sniffer une connexion, ce qui est une tâche non triviale [26, 12], et peut parfois échouer en raison d'une désynchronisation du sniffer. Par conséquent, la réalisation d'une expérience entièrement automatisée pourrait conduire à des résultats invalides (l'échec de l'attaque étant considéré comme un faux négatif, par exemple). Nous avons donc choisi de surveiller manuellement l'expérience: cela nous a permis de contrôler le succès de l'injection mais a impacté le nombre d'attaques qui pouvaient être réalisées dans un délai raisonnable. Nous avons effectué 100 attaques (soit 100 injections réussies lors d'une connexion) et simulé 100 comportements légitimes (c'est-à-dire 100 transmissions de paquets légitimes lors d'une connexion, avec différents types et longueurs de paquets) par cible.

6.1.6 Expérience 6 - BTLEJack

Comme pour *InjectaBLE*, l'attaque *BTLEJack* nécessite de sniffer une connexion et s'appuie sur une stratégie de *jamming*, ce qui entraîne un risque important de désynchronisation ou d'échec de l'attaque. En conséquence, nous avons également choisi de surveiller manuellement l'expérience pour contrôler le succès de l'attaque. Nous avons effectué 100 attaques pour chaque cible, une attaque étant définie comme une connexion qui a été interrompue avec succès par BTLEJack. Nous avons également effectué 100 connexions légitimes par cible (c'est-à-dire une connexion sans attaque). Chaque cible simulait un rôle *Central*, se connectant à répétition à l'ampoule connectée.

6.2 Résultats des expériences

Pour chaque expérience réalisée, nous avons calculé le nombre de vrais positifs (i.e. alerte de détection levée lors d'une séquence d'attaque, notée TP), de faux positifs (i.e. alerte de détection levée lors d'une séquence légitime, notée FP), de vrais négatifs (i.e. pas d'alerte de détection lors d'une séquence légitime, notée TN) et faux négatifs (i.e. pas d'alerte de détection lors d'une séquence d'attaque, notée FN) par cible. Nous avons également calculé le rappel (ou *Recall*) et la précision à l'aide des formules suivantes :

$$Recall = \frac{TP}{TP + FN} \qquad Precision = \frac{TP}{TP + FP}$$

Les résultats de chaque expérience sont répertoriés dans le tableau 2. Différentes observations peuvent être faites à partir de ces résultats. Tout d'abord, nous pouvons souligner que nos stratégies de détection sont pertinentes pour détecter avec succès les attaques, comme l'illustrent les très bonnes valeurs de rappel que nous avons obtenues (comprises entre 0,9 et 1,0). Nous soulignons que ces expériences ayant été menées en conditions réalistes, les résultats associés peuvent être considérés comme représentatifs d'un véritable attaquant utilisant des outils standards.

De même, les bonnes valeurs de précision, toutes comprises entre 0,87 et 1,0, prouvent que nos stratégies de détection ne génèrent qu'une très faible quantité de faux positifs. De plus, quatre de nos six expériences présentent une valeur de précision égale à 1,0 pour chaque cible testée. Les stratégies de détection qui ne reposent que sur la surveillance passive des *advertising* (ex. GATTacker et Jamming) génèrent un peu plus de faux positifs : cette situation s'explique par le fait qu'elles doivent calculer des estimations qui peuvent être impactées par certains changements d'environnement inhérents à ces canaux intensivement utilisés.

Expérience	Cible	TP	FP	TN	FN	Recall	Precision
GATTacker	Ra	250	0	250	0	1.0	1.0
	Ne	250	0	250	0	1.0	1.0
	D1	250	0	250	0	1.0	1.0
	D2	250	19	231	0	1.0	0.93
BTLEJuice	Ga	245	0	250	5	0.98	1.0
	D1	239	0	250	11	0.96	1.0
	D2	250	0	250	0	1.0	1.0
Jamming	Ra	238	9	241	12	0.95	0.96
	Ne	250	13	237	0	1.0	0.95
	D1	247	13	237	3	0.99	0.95
	D2	250	39	211	0	1.0	0.87
KNOB	Ga	225	0	250	25	0.9	1.0
	D1	245	0	250	5	0.98	1.0
	D2	246	0	250	4	0.98	1.0
InjectaBLE	Ra	99	0	100	1	0.99	1.0
	D1	100	0	100	0	1.0	1.0
	D2	94	0	100	6	0.94	1.0
BTLEJack	Ne	95	0	100	5	0.95	1.0
	D1	98	0	100	2	0.98	1.0

Table 2: Résultats des expériences

Enfin, nous pouvons souligner que les résultats d'une expérience donnée sont globalement homogènes pour chaque cible testée. Cela montre que nos modules de détection sont, comme prévu, indépendants des implémentations sous-jacentes du *wrapper*. Même si certaines de nos stratégies ne peuvent pas être implémentées systématiquement sur toutes les cibles en raison des exigences du rôle, ces expérimentations démontrent également que ces stratégies de détections peuvent être implémentées sur différents types d'équipements, dont un smartphone, un *Raspberry Pi* et un objet connecté du commerce avec des ressources limitées.

7 Discussions

Dans cet article, nous avons concentré notre travail sur les attaques bas niveau, qui sont d'une part difficiles à détecter sur les équipements, et pour lesquelles il est difficile d'intégrer des mécanismes de protection d'autre part, même en les prévoyant dès leur conception. Cependant, notre approche peut être facilement appliquée à tout type d'attaques actives ciblant le protocole Bluetooth Low Energy (BLE). En effet, implémenter des heuristiques de détection au niveau le plus bas accessible par logiciel permet à la fois de détecter les attaques de bas niveau mais est également pertinent pour détecter les attaques visant les couches supérieures ou étant liées à une implémentation vulnérable spécifique, cette approche donnant accès à tout le trafic reçu et transmis par le nœud.

Plus important encore, nous considérons également que notre approche est suffisamment générique pour être étendue à d'autres protocoles de communication sans fil couramment utilisés par les équipements *IoT*, tels que Zigbee ou ShockBurst. En effet, les contraintes liées à ce type de protocoles, telles que la dynamicité de l'environnement et l'absence de nœud central, sont effectivement résolues par une détection embarquée effectuée directement par les nœuds eux-mêmes. De même, l'instrumentation des couches les plus basses permet l'accès à un grand nombre de caractéristiques, permettant de construire des modules de détection efficaces pour différents types d'attaques. D'autre part, le fait que nous ayons réussi à implémenter une telle approche pour le protocole BLE, qui fournit de nombreuses fonctionnalités et utilise des mécanismes complexes tels que le saut de canal, est encourageant pour implémenter une telle stratégie sur un protocole sans fil plus simple. Ainsi, nous sommes convaincus que la méthodologie appliquée pour construire nos modules de détection, basée sur l'analyse de l'impact de l'attaque sur les fonctionnalités de bas niveau, peut également

être généralisée à d'autres technologies sans fil.

Certaines limites et défis liés à cette approche doivent également être soulignés. Tout d'abord, la mise en place de la détection sur des nœuds locaux complique la collecte des alertes, surtout si ces alertes doivent remonter vers un SOC unique. Cependant, ce problème peut être résolu en établissant un canal de communication sécurisé dédié à la signalisation des alertes entre un nœud de surveillance central et les nœuds locaux détectant le trafic malveillant. Un tel canal pourrait également être utilisé pour permettre aux nœuds de partager des connaissances sur les menaces détectées ou coordonner des algorithmes de détection plus complexes impliquant plusieurs équipements. Dans une perspective de généralisation de cette conception de détection à d'autres protocoles sans fil, les techniques de *Cross Technology Communications* serait une solution prometteuse pour établir un canal de communication sécurisé entre des équipements locaux intégrant des protocoles sans fil hétérogènes.

Une autre limitation est liée à la nécessité d'écrire du code spécifique pour chaque cible embarquant des piles protocolaires hétérogènes. Lorsque l'implémentation est propriétaire, ce qui est une situation courante, elle nécessite également d'effectuer une rétro-ingénierie de la pile pour comprendre et instrumenter ses fonctions internes. Cette situation a motivé le design générique et modulaire de notre *framework* ainsi que le développement d'outils de rétro-ingénierie automatisés, destinés à faciliter ces tâches complexes. On peut cependant noter le nombre croissant d'implémentations open source de piles protocolaires (par exemple Zephyr ou NimBLE). De plus, certains fabricants pourraient également choisir d'intégrer les modules de détection directement dans leur pile propriétaire: nous avons en effet démontré que l'approche était suffisamment légère pour être embarquée avec succès au sein d'objets connectés aux ressources très limitées.

8 Conclusion

Dans cet article, nous avons présenté une nouvelle approche de détection embarquée pour le protocole BLE, basée sur l'instrumentation des couches les plus basses de la pile (sur le contrôleur lui même). Nous avons démontré la faisabilité et la pertinence de cette approche embarquée en menant plusieurs expérimentations en conditions réalistes sur différentes cibles, dont un smartphone et des objets connectés aux ressources limitées, représentatifs de l'hétérogénéité des objets embarquant cette technologie sans fil. Nous avons réussi à détecter jusqu'à six attaques bas niveau critiques, incluant diverses attaques liées au mode connecté qui étaient particulièrement difficiles à détecter avec les stratégies existantes.

Nous fournissons également un *framework* modulaire, générique et facile d'utilisation permettant d'instrumenter divers contrôleurs BLE, adaptés à la collecte de caractéristiques de détection de bas niveau et publié en open source[1]. Nous considérons que ce *framework* est une contribution importante à la communauté sécurité, car il fournit un moyen simple d'instrumenter les contrôleurs BLE, et pourrait faciliter les travaux de recherche dans divers domaines tels que la recherche de vulnérabilités ou la détection d'intrusion.

Comme travaux futurs, nous prévoyons d'améliorer notre *framework* pour inclure de nouveaux types de contrôleurs et l'étendre à d'autres protocoles de communication sans fil, tels que Zigbee ou ShockBurst. Nous prévoyons d'explorer la faisabilité de la construction d'un système de détection d'intrusion coopératif, utilisant un ensemble de nœuds décentralisés capables de coopérer ensemble, qui communiquent à l'aide d'un canal de communication sans fil sécurisé. Nous pensons également que notre approche pourrait être pertinente dans une stratégie de prévention d'intrusion, en exploitant les capacités d'instrumentation de notre logiciel embarqué pour prévenir des attaques, par exemple en provoquant la terminaison une connexion malveillante.

[1]Note aux reviewers: le code sera rendu disponible sous license MIT avant la conférence.

References

[1] Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, April 2016.

[2] *2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019*. IEEE, 2019.

[3] Ahmed Aboukora, Guillaume Bonnet, Florent Galtier, Romain Cayre, Vincent Nicomette, and Guillaume Auriol. A defensive man-in-middle approach to filter ble packets. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 365–367, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The knob is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, 2019.

[5] Armis. Blueborne Technical White Paper. `https://go.armis.com/hubfs/BlueBorneTechnicalWhitePaper.pdf`, 2017.

[6] Armis. BleedingBit Technical White Paper. `https://go.armis.com/hubfs/BLEEDINGBIT-TechnicalWhitePaper.pdf`, 2018.

[7] Bluetooth SIG. *Bluetooth Core Specification*, 12 2019.

[8] S. Bräuer, A. Zubow, S. Zehl, M. Roshandel, and S. Mashhadi-Sohi. On practical selective jamming of bluetooth low energy advertising. In *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–6, 2016.

[9] Damien Cauquil. Btlejuice: The bluetooth smart mitm framework, 2016.

[10] Damien Cauquil. Sniffing btle with the micro:bit. *PoC or GTFO*, 17:13–20, 2017.

[11] Damien Cauquil. You'd better secure your BLE devices or we'll kick your butts !, 2018.

[12] Damien Cauquil. Defeating Bluetooth Low Energy 5 PRNG for fun and jamming, 2019.

[13] Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. InjectaBLE: Injecting malicious traffic into established Bluetooth Low Energy connections. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*, Taipei (virtual), Taiwan, June 2021.

[14] Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, and Geraldine Marconato. Mirage: towards a metasploit-like framework for iot. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 261–270. IEEE, 2019.

[15] Jiska Classen and Matthias Hollick. Inside job: Diagnosing bluetooth lower layers using off-the-shelf devices. *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, May 2019.

[16] Jiska Classen and Matthias Hollick. Extracting physical-layer ble advertisement information from broadcom and cypress chips. pages 337–339, 07 2020.

[17] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[18] Jose Gutierrez del Arroyo, Jason Bindewald, Scott Graham, and Mason Rice. Enabling bluetooth low energy auditing through synchronized tracking of multiple connections. *Int. J. Crit. Infrastruct. Prot.*, 18(C):58–70, sep 2017.

[19] Sławomir Jasek. Gattacking Bluetooth Smart Devices. 2017.

[20] Mateusz Krzysztoń and Michał Marks. Simulation of watchdog placement for cooperative anomaly detection in bluetooth mesh intrusion detection system. *Simulation Modelling Practice and Theory*, 101:102041, 2020. Modeling and Simulation of Fog Computing.

[21] Andrea Lacava, Emanuele Giacomini, Francesco D'Alterio, and Francesca Cuomo. Intrusion detection system for bluetooth mesh networks: Data gathering and experimental evaluations. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 661–666, 2021.

[22] Abdelkader Lahmadi, Alexis Duque, Nathan Heraief, and Julien Francq. Mitm attack detection in ble networks using reconstruction and classification machine learning techniques. In *MLCS 2020-2nd Workshop on Machine Learning for Cybersecurity*, 2020.

[23] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. Internalblue - bluetooth binary patching and experimentation framework. *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, Jun 2019.

[24] Akm Iqtidar Newaz, Amit Kumar Sikder, Leonardo Babun, and Selcuk Uluagac. Heka: A novel intrusion detection system for attacks to personal medical devices. pages 1–9, 06 2020.

[25] Sultan Qasim Khan. Sniffle: A sniffer for Bluetooth 5 (LE), 2019. `https://hardwear.io/netherlands-2019/presentation/sniffle-talk-hardwear-io-nl-2019.pdf`.

[26] Mike Ryan. Bluetooth: With Low Energy comes Low Security. 2013.

[27] Mike Ryan. How Smart is Bluetooth Smart ? 2013.

[28] Aiku Shintani. The design, testing, and analysis of a constant jammer for the bluetooth low energy (ble) wireless communication protocol. 06 2020.

[29] Yunsick Sung. Intelligent security IT system for detecting intruders based on received signal strength indicators. *Entropy*, 18(10):1–16, October 2016.

[30] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield: Detecting spoofing attacks in bluetooth low energy networks. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 397–411, San Sebastian, October 2020. USENIX Association.

[31] Muhammad Yaseen, Waseem Iqbal, Imran Rashid, Haider Abbas, Mujahid Mohsin, Kashif Saleem, and Yawar Abbas Bangash. Marc: A novel framework for detecting mitm attacks in ehealthcare ble systems. *Journal of Medical Systems*, 43(11):324, 2019.