Rétro-ingénierie et détournement de piles protocolaires embarquées, un cas d'étude sur le système ESP32

Damien Cauquil[1] et Romain Cayre[2]
dcauquil@quarkslab.com
romain.cayre@eurecom.fr

[1] Quarkslab
[2] EURECOM

Résumé. Ces dernières années, les systèmes sur puce (SoC) fournissant une connectivité sans fil ont connu une popularité croissante. En particulier les systèmes sur puce ESP32 d'Espressif sont très répandus : en effet, bien qu'initialement conçus comme des solutions à bas coût pour les *makers* et hobbyistes, ils sont de plus en plus utilisés dans des solutions commerciales en raison de leurs nombreuses fonctionnalités et d'un faible coût de revient, d'autant plus dans cette période où les composants se font rares et chers. L'analyse de ces systèmes nous semble aujourd'hui fondamentale, et, bien que ceux-ci aient déjà été partiellement explorés au sein d'autres travaux, de nombreux composants matériels et logiciels utilisés par ces puces restent aujourd'hui opaques, notamment du fait de l'absence de documentation et l'usage de technologies propriétaires.

Nous étudions dans cet article l'architecture de ces systèmes sur puce, et en particulier le contrôleur matériel en charge de la pile protocolaire Bluetooth Low Energy (BLE) ainsi que leurs composants logiciels associés. Nous démontrons également comment ceux-ci peuvent être détournés de leur usage initial pour mettre en place des techniques offensives avancées visant les couches basses de multiples protocoles sans fil, y compris des protocoles non nativement supportés par la puce. Nous montrons ainsi comment un tel système peut servir à *fuzzer* un périphérique BLE, intercepter les communications d'un clavier sans fil et d'un capteur de rythme cardiaque utilisant des protocoles propriétaires, brouiller de nombreuses communications sans fil simultanément ou encore attaquer un système domotique basé sur le protocole ZigBee. Ces différentes techniques se basent sur le détournement de mécanismes internes dont certains ne peuvent être corrigés par le constructeur. Enfin, nous publions un *framework* développé sur mesure ainsi que des preuves de concept permettant de mettre en œuvre ces différentes attaques.

1 Introduction

On assiste aujourd'hui à une démocratisation de l'utilisation des systèmes sur puce, aux fonctionnalités toujours plus riches et aux architectures

toujours plus complexes. De plus en plus de ces systèmes proposent une connectivité sans fil, en faisant des composants de choix pour de nombreux objets connectés. Dans le contexte de ce déploiement croissant, l'analyse de ces systèmes devient fondamentale pour la communauté sécurité, tant dans une perspective défensive, pour mieux comprendre les systèmes afin d'identifier et corriger leurs vulnérabilités, que dans une perspective offensive, pour explorer les nouvelles surfaces d'attaques liées à leur déploiement. L'analyse de sécurité des composants logiciels et matériels impliqués dans la connectivité sans fil fournie par ces systèmes, en particulier, reste aujourd'hui un enjeu majeur, impactant tant la sécurité sans fil que la sécurité embarquée et mobilisant une approche interdisciplinaire à l'interface entre l'électronique, le traitement du signal et l'informatique. L'analyse des composants logiciels est d'autant plus pertinente que toutes les applications développées sur ces systèmes reposent sur les librairies fournies par le SDK mais également sur les librairies présentes en mémoire morte, la sécurité de toutes les applications développées sur ces modules en est donc dépendante.

Les systèmes sur puce ESP32 d'Espressif offrent des fonctionnalités avancées et une connectivité Wi-Fi et Bluetooth Low Energy pour un coût dérisoire, en faisant un système de choix tant pour les hobbyistes et les *makers* que pour les fabricants d'équipements connectés. Au-delà de l'engouement pour ces systèmes, nous nous sommes intéressés tout particulièrement aux périphériques intégrés aux ESP32 assurant les communications Bluetooth Low Energy.

Cet article détaille ainsi les résultats de cette exploration des systèmes ESP32-WROOM, ESP32-S3 et ESP32-C3. Nous explorons de nombreuses couches, depuis l'architecture de la plate-forme ESP32 elle même, jusqu'au fonctionnement du contrôleur matériel en charge du protocole BLE. Au passage, nous abordons les couches protocolaires bas-niveau intégrées et nous démontrons plusieurs manières d'exploiter ces dernières pour implémenter différentes attaques. Nous détournons l'ESP32 non seulement pour attaquer des équipements supportant le protocole BLE, mais aussi d'autres protocoles de communication, non supportés officiellement par l'ESP32, mais partageant certaines caractéristiques de modulation. Grâce à ces stratégies inter-protocolaires, nous démontrons des attaques sur le ZigBee (utilisé en domotique notamment), Mosart (utilisé principalement dans des claviers et souris sans-fil), ou encore ANT et ANT+ (que l'on retrouve principalement dans des équipements connectés dédiés au sport et à la santé).

2 État de l'art

On observe ces dernières années un intérêt particulier pour l'analyse et le détournement de systèmes sur puce, notamment dans le domaine de la sécurité sans fil. Au-delà de l'intérêt intrinsèque d'étudier le fonctionnement et l'architecture de ces puces du point de vue de la sécurité, on peut noter une connexion étroite entre le détournement de ce type d'équipements et la découverte de nouvelles techniques offensives. De nombreux détournements ont ainsi été réalisés dans le cadre de travaux de recherche appliquée sur la sécurité des protocoles sans fil, et ont permis l'exploration de stratégies d'attaques nouvelles. Ces travaux poussent souvent les puces aux limites de leurs capacités techniques, par l'exploitation de détails d'architecture matérielle ou d'implémentations logicielles vulnérables, ouvrant ainsi de nouvelles perspectives techniques et scientifiques.

Ces différents travaux répondent souvent à des problématiques techniques concrètes. En effet, l'analyse offensive des protocoles sans fil reste aujourd'hui complexe, et se heurte régulièrement aux limites des outils d'analyse existants. Les Radios Logicielles (ou SDRs *Software Defined Radios*) offrent une généricité particulièrement intéressante mais restent onéreuses, nécessitent un travail d'ingénierie conséquent et imposent des limites liées à leur bande passante réduite et à la latence introduite par les composants logiciels. C'est notamment le cas pour l'analyse de protocoles utilisant des algorithmes de saut de fréquence, qui impliquent le suivi en temps réel de la séquence (nécessitant des équipements capables de modifier suffisamment rapidement leur fréquence centrale) ou la surveillance de larges bandes de fréquences (nécessitant l'usage de matériels d'autant plus onéreux que la bande utilisée est large). Ces limites ont motivé le développement de solutions matérielles dédiées, telles que l'Ubertooth [26, 29], un dongle dédié à l'analyse des protocoles Bluetooth et Bluetooth Low Energy. On peut également citer les outils matériels développés dans le cadre de l'analyse des protocoles propriétaires de claviers sans fil, tels que KeyKeriki [27], ou l'analyse des protocoles basés sur la spécification 802.15.4 [18] par le biais de l'APImote [16].

Les détournements de systèmes sur puce à des fins offensives ont permis de complémenter ces outils dédiés, abaissant considérablement le coût des outils d'analyse et permettant de nouvelles stratégies d'attaques. Le détournement d'un registre matériel sur la puce nRF24L01 de Nordic Semiconductor par Travis Goodspeed pour permettre l'écoute passive de protocoles propriétaires dans la bande 2,4 GHz [17] a ainsi constitué une étape décisive, permettant le développement d'un firmware d'analyse dé-

dié [24] et la découverte de nombreuses vulnérabilités visant divers claviers et souris sans fil [23]. De nombreux travaux ont étendu ce détournement aux puces nRF51 et nRF52, permettant le développement d'outils offensifs sur différentes plateformes [6, 7, 12] et la découverte d'attaques bas niveau critiques visant le BLE [8, 9]. On peut également citer le détournement du dongle RZUSBStick d'ATMEL par Joshua Wright pour le doter de fonctionnalités d'injection dans le cadre de ses travaux sur la sécurité du protocole ZigBee [35].

D'autres détournements se sont quand à eux concentrés sur la rétro-ingénierie et le détournement de piles protocolaires embarquées existantes. Mathy Vanhoef et son outil *modwifi* ont permis le développement d'une série d'attaque bas niveau visant le protocole Wi-Fi [34]. De même, les travaux de Matthias Schulz et al. ont exploré le détournement des piles Wi-Fi propriétaires de Broadcom [28]. Dans le cas du Bluetooth et du Bluetooth Low Energy. La suite d'outils *InternalBlue* [20] a quant à elle permis l'analyse et l'instrumentation des piles protocolaires propriétaires de Broadcom et Cypress, servant de support à de multiples travaux offensifs [1, 2, 15]. On peut également citer Matheus Garbelini et son *fuzzer* Bluetooth Classic basé sur ESP32, publié dans le cadre de la série de vulnérabilités nommées *BrakTooth* [14].

Notre travail s'inscrit dans la lignée de ces détournements de systèmes sur puce en poursuivant la rétro-ingénierie à des fins offensives de la plateforme ESP32, et en l'étendant à ses variantes ESP32-S3 et ESP32-C3. Nous nous concentrons notamment sur l'analyse et le détournement de la pile protocolaire Bluetooth Low Energy propriétaire embarquée par Espressif au sein de ces systèmes, ainsi que du contrôleur matériel impliqués dans son fonctionnement. Nous montrons que les couches basses de cette pile protocolaire peuvent être détournées pour mettre au point une série d'attaques sans fil bas niveau, visant le protocole Bluetooth Low Energy mais également d'autres protocoles sans fil co-existant dans la même bande de fréquences.

3 Architecture matérielle ESP32

3.1 Architecture générale

Les systèmes ESP32 intègrent un ou plusieurs processeurs, une couche gérant les communications radio, une ou plusieurs piles protocolaires (Wi-Fi, Bluetooth Low Energy, Bluetooth BR/EDR), un accélérateur cryptographique ainsi que bon nombre d'autres périphériques variés permettant de les interfacer avec un grand nombre d'équipements : écrans, carte SD,

interface Ethernet, etc. Les images image 1 et image 2 montrent deux exemples d'architectures de systèmes populaires, respectivement *ESP32-D0WDQ6* (utilisé dans les cartes de développement *ESP32-WROOM-32*) et *ESP32-C3*.

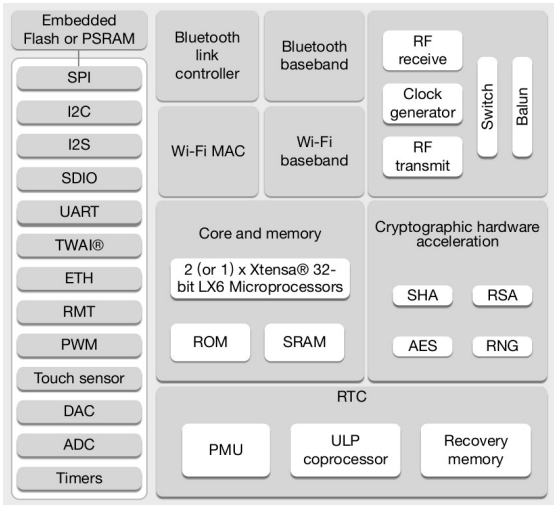

Fig. 1. Architecture ESP32-D0WDQ6 (famille ESP32), extraite de la version 4.2 du datasheet de l'ESP32 [31], (section 1.6, figure 1, page 12).

3.2 Processeurs

Les systèmes ESP32 d'Espressif reposent actuellement sur deux architectures processeurs différentes :

— XTensa de Tensilica (présent dans les familles ESP32 et ESP32-S notamment) ;
— RISC-V (présent dans les familles ESP32-C, ESP32-H, ou encore la récente ESP32-P).

Espressif utilise désormais exclusivement l'architecture RISC-V dans ses nouveaux systèmes sur puce, comme l'a récemment annoncé Teo Swee Ann, présidente-directrice générale d'Espressif Systems [13].

3.3 Organisation de la mémoire

Les figures 1 et 2 présentent aussi le cœur des systèmes ESP32 (*Core System*), composé d'un ou de plusieurs processeurs, mais aussi de mémoire

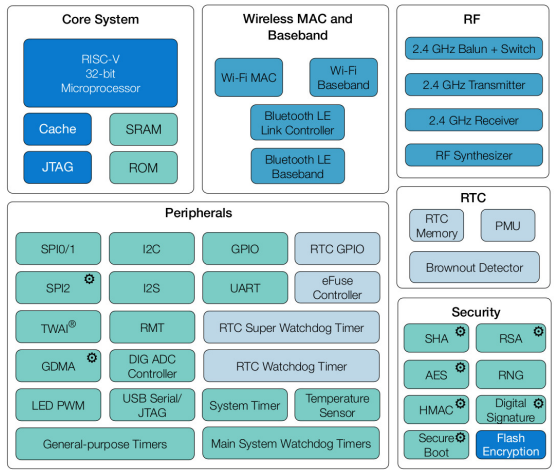

Fig. 2. Architecture ESP32-C3 (famille ESP32-C), extraite de la version 1.4 du datasheet de l'ESP32-C3 [30], (figure 1, page 2).

vive statique (*SRAM*) et de mémoire morte (*Mask ROM*). S'ajoute à cela une mémoire Flash, externe ou embarquée selon les variantes du système sur puce et interfacée via SPI, qui contient l'application développée par l'utilisateur ainsi que les données utilisées par cette dernière.

La documentation technique détaille notamment les différentes mémoires mortes présentes au sein d'un système sur puce ESP32 dans sa section présentant la cartographie mémoire, comme le montre l'image 3 extraite de la documentation du système ESP32-D0WDQ6 [31] (section 3.1.4, table 7, page 26). Deux segments de mémoire sont associés à de la mémoire morte respectivement présents aux adresses `0x40000000` et `0x3FF90000` et de tailles 384 Kio et 64 Kio.

Category	Target	Start Address	End Address	Size
Embedded Memory	Internal ROM 0	0x4000_0000	0x4005_FFFF	384 KB
	Internal ROM 1	0x3FF9_0000	0x3FF9_FFFF	64 KB
	Internal SRAM 0	0x4007_0000	0x4009_FFFF	192 KB
	Internal SRAM 1	0x3FFE_0000	0x3FFF_FFFF	128 KB
		0x400A_0000	0x400B_FFFF	
	Internal SRAM 2	0x3FFA_E000	0x3FFD_FFFF	200 KB
	RTC FAST Memory	0x3FF8_0000	0x3FF8_1FFF	8 KB
		0x400C_0000	0x400C_1FFF	
	RTC SLOW Memory	0x5000_0000	0x5000_1FFF	8 KB

Fig. 3. Cartographie de la mémoire intégrée dans le système sur puce ESP32-D0WDQ6.

Ces mémoires mortes contiennent notamment les implémentations des différentes piles protocolaires (Wi-Fi, Bluetooth BR/EDR et Bluetooth Low Energy), qui sont intégrées au système sur puce et non-modifiables.

Les systèmes ESP32 utilisent une architecture de type Harvard qui sépare les données des instructions consitutives du programme exécuté sur ces derniers. Ainsi, la mémoire RAM interne *SRAM0* est dédiée au stockage des instructions du programme tandis que les autres mémoires RAM (*SRAM1* et *SRAM2*) sont utilisées pour le stockage des données initialisées et non-initialisées, ainsi que du tas. Quant à la mémoire Flash, cette dernière peut atteindre au maximum 16 Mibi-octets et peut-être ajoutée à l'espace d'adressage de la mémoire de stockage des instructions au travers de mémoires tampons à haute vitesse. De la même manière, une puce externe fournissant de la mémoire vive (*pseudo-static RAM*, ou généralement appelée PSRAM) peut être connectée au système-sur-puce et permettre d'étendre la capacité en mémoire vive de ce dernier. Cette mémoire vive externe voit cependant sa vitesse de transfert limitée par celle du bus SPI qui est utilisé par le système-sur-puce pour communiquer avec cette dernière.

4 Analyse de la pile protocolaire

4.1 Extraction du contenu des mémoires mortes du système sur puce

Les piles protocolaires intégrées dans les Systèmes sur Puce ESP32 ont ainsi pu être extraites grâce à l'utilitaire *esptool.py* mis à disposition par Espressif sur le dépôt Github associé [33].

4.2 Chargement dans Gidra des deux segments ROM

Les données extraites des deux mémoires mortes ne sont pas utilisables en l'état, car bien que l'on connaisse leur emplacement en mémoire nous n'avons aucune idée des adresses des fonctions qu'elles contiennent. Cependant le système de compilation d'Espressif, *esp-idf* [32], contient des scripts d'édition de liens qui précisent les adresses des fonctions présentes en ROM, comme le montre le listing 1.

Listing 1: Script d'édition de liens

```
/*
ESP32 ROM address table
Generated for ROM with MD5sum:
ab8282ae908fe9e7a63fb2a4ac2df013  ../../rom_image/prorom.elf
*/
PROVIDE ( Add2SelfBigHex256 = 0x40015b7c );
PROVIDE ( AddBigHex256 = 0x40015b28 );
PROVIDE ( AddBigHexModP256 = 0x40015c98 );
PROVIDE ( AddP256 = 0x40015c74 );
PROVIDE ( AddPdiv2_256 = 0x40015ce0 );
PROVIDE ( app_gpio_arg = 0x3ffe003c );
PROVIDE ( app_gpio_handler = 0x3ffe0040 );
PROVIDE ( BasePoint_x_256 = 0x3ff97488 );
PROVIDE ( BasePoint_y_256 = 0x3ff97468 );
PROVIDE ( bigHexInversion256 = 0x400168f0 );
PROVIDE ( bigHexP256 = 0x3ff973bc );
PROVIDE ( btdm_r_ble_bt_handler_tab_p_get = 0x40019b0c );
```

La compilation d'un code exemple permet ainsi d'obtenir un fichier ELF qui contient un grand nombre de symboles. Parmi ces symboles, on retrouve ceux employés par NimBLE, une implémentation de pile protocolaire BLE open-source particulièrement populaire. Il est possible de l'ouvrir avec Ghidra et de le désassembler.

Les segments de ROM du système ESP32 sont ensuite chargés en mémoire et placés aux bonnes adresses dans Ghidra, et les symboles relatifs à ces derniers (extraits du fichier d'édition de liens) ajoutés grâce à un script réalisé sur mesure. Il est alors possible de décompiler le code de la ROM et de déterminer son interaction avec le code de notre application.

4.3 Architecture de pile protocolaire BLE

Bien que l'environnement ESP32 propose de choisir entre l'implémentation de *NimBLE* ou de *Bluedroid*, ce dernier contrôle néanmoins la plupart des opérations liées au protocole BLE via du code présent en ROM et exposant une interface vHCI. C'est cette dernière qu'utilisent les implémentations *NimBLE* et *Bluedroid*, qui n'ont donc aucune possibilité de manipuler les PDU échangés via BLE.

L'image 4 synthétise l'architecture en place et les interconnexions entre l'application que nous avons pu identifier, l'adaptateur vHCI intégré à la ROM et le contrôleur matériel Bluetooth Low Energy intégré au système sur puce.

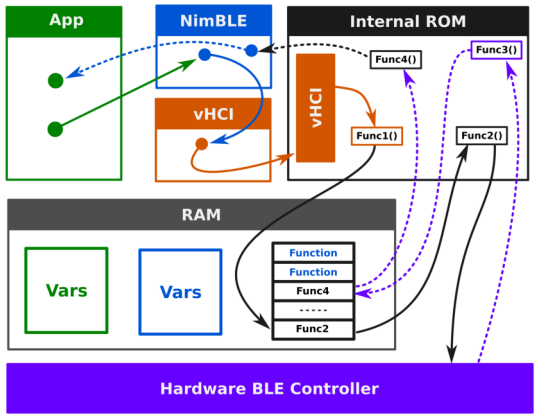

Fig. 4. Architecture Bluetooth Low Energy de l'ESP32.

Ce contrôleur matériel n'étant pas documenté, nous avons toutefois réussi à l'identifier en analysant les adresses des différents registres et en tombant avec un peu de chance sur un *pastebin* référençant le composant DA14681 [25]. Ce dernier possède des registres similaires à ceux employés sur l'ESP32, et nous en avons déduit qu'il était très probable qu'Espressif ait intégré un contrôleur déjà existant, mais placé à une adresse différente (`0x3FF71200`).

Nous avons ainsi pu observer dans le code de la fonction *r_lld_adv_start* des instructions configurant les registres `BLE_ADVCHMAP_REG` et `BLE_ADVTIM_REG` gérant respectivement la configuration des canaux d'annonce et l'intervalle d'annonce, comme le montre l'image 5, ce qui est totalement cohérent avec la documentation du DA14681.

```
_BLE_ADVCHMAP_REG = (uint)*(byte *)(iVar3 + 0x16);
memw();
if (0x1f < *(byte *)(iVar3 + 0x19)) {
  iVar5 = (*(byte *)(iVar3 + 0x19) + 0x10) * 8 + 0x564;
}
memw();
_BLE_ADVTIM_REG = iVar5;
```

Fig. 5. Configuration de l'intervalle d'annonce et des canaux d'annonces par *r_lld_adv_start()*.

En analysant plusieurs fonctions de la pile protocolaire Bluetooth Low Energy, nous avons pu comprendre comment cette dernière échange

des données avec le contrôleur matériel, au travers d'une zone mémoire d'échange dédiée. Cette zone mémoire est utilisée notamment pour stocker des structures spécifiques au contrôleur matériel, des *descripteurs* qui ont vocation à stocker les méta-données des PDU BLE (canal sur lequel un PDU a été reçu, le niveau de signal, le type de PDU) et les données contenues dans ces derniers. Elle est utilisée par la pile protocolaire pour indiquer au contrôleur BLE qu'il a des PDU à envoyer, et par le contrôleur pour transmettre à la pile protocolaire les PDU reçus.

Fonctionnalités du contrôleur DA14681 Le contrôleur BLE DA14681 qui semble être intégré dans les puces ESP32 est en charge de la gestion complète de la couche physique du protocole Bluetooth Low Energy. C'est lui qui gère notamment l'établissement de connexions, le mécanisme de saut de canal (FHSS), la liste des canaux utilisés pour la connexion en cours (aussi dénommée *channel map*), ainsi que les opérations d'émission et de réception de données. Pour ce faire, il utilise une zone mémoire dédiée au stockage de *descripteurs*, des structures spécifiques utilisées pour stocker les différents PDU émis ou reçus, mais aussi des informations liées à la configuration de ce dernier.

Il expose aussi plusieurs registres de 16 bits permettant de contrôler ses caractéristiques et son état, mais sans permettre un accès à son fonctionnement bas-niveau (radio-fréquence). Ainsi, il est possible de changer l'adresse Bluetooth du contrôleur, la clé de chiffrement AES et le MIC utilisé lors des opérations de chiffrement, etc.

Le fait que ce contrôleur soit en charge de la gestion des connexions et ne permette pas d'accéder à un contrôle bas-niveau limite drastiquement les possibilités de détournement. En effet, il n'est pas possible pour le moment de le placer en écoute sur un seul canal et de recevoir n'importe quelle donnée correctement démodulée, ni de gérer de façon logicielle le changement de canal, rendant de fait l'interception passive impossible.

Interface vHCI L'intégration du contrôleur BLE dans la pile protocolaire d'Espressif passe par une couche spécifique faisant l'interface entre le contrôleur et la pile protocolaire BLE qui, pour rappel, peut être choisie au sein du *framework* ESP-IDF entre *NimBLE* et *Bluedroid*. Cette couche spécifique se présente sous forme d'un contrôleur vHCI, implémentée dans la ROM du système-sur-puce, avec lequel les différentes piles protocolaires peuvent interagir. Cette manière de faire est intelligente car cela permet de limiter l'effort requis pour adapter les piles protocolaires existantes, l'interface HCI (*Host Controller Interface*) étant d'une part définie dans

la spécification Bluetooth et utilisant des messages standardisés tout en limitant les risques de détournement.

Lorsqu'une application utilise l'interface BLE, elle initialise une des deux piles protocolaires fournies (par exemple *NimBLE*, qui de fait initie une connexion avec l'interface vHCI du contrôleur Bluetooth Low Energy et le pilote par la suite. La gestion du fonctionnement du contrôleur BLE est totalement masquée par le contrôleur vHCI, et l'application est notifiée des différents évènements via les messages HCI adéquats.

Identification d'une implémentation similaire Nous avons cherché sur Internet de possibles implémentations similaires à celle intégrée dans le *framework* ESP-IDF, et sommes tombés sur un dépôt de code Github [11] conçu pour un contrôleur BLE différent (BK7231). Après comparaison entre le code source et les structures observées lors de l'analyse par ingénierie à rebours, il nous a semblé évident que ces deux implémentations avaient de nombreuses similitudes. Avoir accès à un code source similaire à celui utilisé par Espressif pour le développement de son code intégré en ROM a grandement simplifié son analyse et sa compréhension, et a permis notamment de faciliter son instrumentation et le détournement de fonctions clés.

4.4 Interface ROM / Flash

Le code présent dans la ROM de l'ESP32 est fixe et ne peut être modifié une fois intégré, ce qui empêche évidemment toute modification. Or, il n'est pas rare qu'un bogue soient identifié au sein de ce code non-modifiable et qu'il faille trouver un moyen de le corriger, au moins à l'exécution. C'est pourquoi le code de la ROM utilise des tableaux de pointeurs stockés en RAM pour exposer les fonctions d'interface des différents composants. En effet, lorsqu'une fonction de la ROM veut appeler une fonction spécifique d'un composant, cela passe par une indirection via le tableau de pointeurs de fonctions (en réalité une structure définie dans le code et n'étant composée que de pointeurs de fonctions).

Ce mécanisme permet ensuite de pouvoir remplacer à l'exécution une fonction problématique par une version corrigée, qui elle aussi peut appeler différentes fonctions exposées en utilisant ces pointeurs de fonctions. Un exemple assez flagrant dans le code du *framework* d'Espressif est le cas des fonctions dont le nom se termine par *_hack*, comme le montre l'image 6.

Certaines fonctions des bibliothèques fournies par Espressif utilisent aussi ce mécanisme pour installer des fonctions de rappel afin d'être

```
undefined4 config_lld_funcs_reset(undefined4 param_1)

{
  ip_func_t *piVar1;
  code **ppcVar2;

  piVar1 = r_ip_funcs_p;
  ppcVar2 = (code **)&r_ip_funcs_p->r_lld_scan_start_hack;
  r_ip_funcs_p->r_lld_init = r_lld_init;
  piVar1->r_lld_adv_start = r_lld_adv_start;
  piVar1->r_lld_adv_stop_hack = r_lld_adv_stop_hack;
  *ppcVar2 = r_lld_scan_start_hack;
  piVar1->r_lld_scan_stop_hack = r_lld_scan_stop_hack;
  piVar1->r_lld_con_start = r_lld_con_start;
  piVar1->r_lld_move_to_master_hack = r_lld_move_to_master_hack;
  piVar1->r_lld_move_to_slave_hack = r_lld_move_to_slave_hack;
  piVar1->r_lld_get_mode = r_lld_get_mode;
  piVar1->r_lld_con_update_after_param_req = r_lld_con_update_after_param_req;
  return param_1;
}
```

Fig. 6. Installation de fonctions modifiées au sein des tableau globaux de pointeurs de fonctions

notifiées de certains évènements ou permettre de traiter des données avant qu'elles soient utilisées par des fonctions présentes en ROM.

5 Instrumentation et détournement

Le contrôleur matériel et le fonctionnement de la pile protocolaire Bluetooth Low Energy ayant été déterminés et analysés, nous nous sommes demandés s'il était possible de détourner ces derniers pour :

- intercepter et altérer les différentes données échangées dans le cadre d'une connexion BLE établie à partir d'un ESP32 ;
- injecter des données dans une connexion existante initiée par un ESP32 ;
- détourner le contrôleur pour l'employer avec des protocoles non nativement supportés tels ZigBee, ANT+ ou Mosart ;
- détourner les fonctions bas-niveau du contrôleur pour brouiller des communications ou établir un canal de communication caché.

Ces différentes tâches requièrent une interface directe avec le contrôleur BLE, la possibilité de détourner des appels de fonctions et de manipuler des structures systèmes en mémoire. L'interfaçage avec le contrôleur BLE étant relativement simple et évident à réaliser, nous nous sommes dans un premier temps intéressés à la possibilité d'espionner et d'altérer le flux de traitement des paquets BLE de la pile protocolaire.

5.1 Hooking de fonctions de la pile protocolaire Bluetooth Low Energy

Dans la section précédente, nous avons abordé le fait que la pile protocolaire utilise de manière récurrente des tableaux de pointeurs de

fonction afin de permettre au code de l'application (et des bibliothèques officielles intégrées à cette dernière) d'enregistrer des interfaces spécifiques (des *callbacks*) pour gérer divers évènements. Ce mécanisme peut tout à fait être utilisé pour détourner des appels de fonctions en manipulant directement ces tableaux de pointeurs, ces derniers étant stockés en RAM. Aucun mécanisme de protection d'accès à la mémoire n'est présent, il est donc possible d'intervenir à divers endroits du code de la pile protocolaire.

Cette technique permet d'intercepter des fonctions appelées lorsque certains évènements se produisent (par exemple la réception d'un PDU), comme l'illustre l'image 7. Le tableau de pointeurs de fonctions est stocké en RAM et est modifié de façon à remplacer un pointeur de fonction par l'adresse d'une fonction de l'application, et cette dernière redirige de manière transparente sur la fonction d'origine.

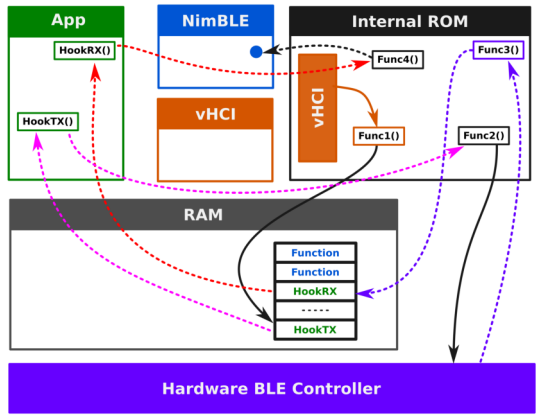

Fig. 7. Détournement des fonctions d'émission/réception de PDU.

Interception des PDU Bluetooth Low Energy À l'aide de la technique de hooking présentée ci-dessus, il est possible de détourner la fonction principale en charge du traitement des PDU reçus. Cette fonction accède normalement à une zone mémoire dédiée à l'échange de données entre le contrôleur BLE et la pile protocolaire, que notre fonction de hook peut aussi lire et modifier.

Nous pouvons dès lors espionner les informations reçues via la connexion BLE d'un ESP32, voire même :

— modifier la taille d'un PDU en la forçant à zéro évitera tout traitement de ce dernier (il sera considéré comme un PDU vide, sera comptabilisé, mais ne déclenchera aucun traitement spécifique) ;
— modifier le contenu d'un PDU et laisser la pile protocolaire le gérer permet d'altérer le fonctionnement de cette dernière.

Le détournement d'une seconde fonction propre à la pile protocolaire servant à envoyer des PDU, `lld_pdu_data_send`, permet quant à lui d'intercepter les PDU avant leur envoi au contrôleur. Il est ensuite possible de les modifier avant leur envoi, voire même de les bloquer. Le code du listing 2 correspond à un hook développé dans le cadre de cette recherche.

Listing 2: Fonction d'interception d'émission de PDU BLE

```
/**
 * _lld_pdu_data_send()
 *
 * This hook is called whenever the BLE stack sends a data PDU.
 **/

int _lld_pdu_data_send(struct hci_acl_data_tx *param)
{
  struct em_buf_tx_desc *p_desc = NULL;
  uint8_t *ptr_data;
  int i, forward=HOOK_FORWARD;
  struct co_list_hdr *tx_desc;
  struct em_desc_node *tx_node;

  if (gpfn_on_tx_data_pdu != NULL)
  {
    /* Should we block this data PDU ? */
    if (gpfn_on_tx_data_pdu(
        0,
        (uint8_t *)(p_rx_buffer + param->buf->buf_ptr),
        param->length
    ) == HOOK_BLOCK)
    {
      /* Set TX buffer length to zero (won't be transmitted,
         but will be freed later. */
      param->length = 0;
    }
  }

  /* Forward to original function. */
  return pfn_lld_pdu_data_send(param);
}
```

Injection de PDU arbitraire Il est tout à fait possible d'appeler des fonctions de la pile protocolaire afin d'injecter un PDU au sein d'une connexion établie, notamment en abusant de la fonction `lld_pdu_data_tx_push` et d'une vulnérabilité dans cette dernière qui permet d'envoyer des PDU de données (L2CAP) ainsi que des PDU de contrôle. L'implémentation présentée dans le listing 3 permet de réaliser cette injection.

Listing 3: Fonction d'injection de PDU BLE

```
1  void IRAM_ATTR send_raw_data_pdu(int conhdl, uint8_t llid, void
   ↪  *p_pdu,
2                                   int length, bool can_be_freed)
3  {
4    struct em_buf_node* node;
5    struct em_desc_node *data_send;
6    struct lld_evt_tag *env = (struct lld_evt_tag *)(
7      *(uint32_t*)((uint32_t)llc_env[conhdl]+0x10) + 0x28
8    );
9
10   portDISABLE_INTERRUPTS();
11   /* Allocate data_send. */
12   data_send = (struct em_desc_node *)em_buf_tx_desc_alloc();
13
14   /* Allocate a buffer. */
15   node = em_buf_tx_alloc();
16
17   /* Write data into allocated buf node. */
18   memcpy((uint8_t *)((uint8_t *)p_rx_buffer + node->buf_ptr),
   ↪  p_pdu, length);
19
20   /* Write information into our em_desc_node structure. */
21   data_send->llid = llid;
22   data_send->length = length;
23   data_send->buffer_idx = node->idx;
24   data_send->buffer_ptr = node->buf_ptr;
25
26   /* Call lld_pdu_data_tx_push */
27   pfn_lld_pdu_data_tx_push(env, data_send, can_be_freed);
28
29   env->tx_prog.maxcnt--;
30
31   portENABLE_INTERRUPTS();
32
33 }
```

Prise d'empreinte à distance d'équipements Bluetooth Low Energy La spécification Bluetooth Low Energy détaille une procédure intéressante dans le cadre de l'analyse d'un équipement connecté : l'échange des informations de version. Cette dernière n'est pas obligatoire mais ne peut être réalisée qu'une seule fois, et consiste en l'échange par les deux équipements participant à une communication d'informations détaillant les éléments suivants :

— la version Bluetooth Low Energy supportée par le système ;
— l'identifiant du constructeur du système-sur-puce supportant le protocole Bluetooth Low Energy ;
— la version du micro-logiciel embarqué, sous forme d'entier non-signé de 16 bits.

Côté implémentation, il suffit d'envoyer un PDU `LL_VERSION_IND` (code opération 0x0C) avec des informations erronées concernant le système émetteur, comme le montre le listing 4, et enfin, d'attendre la réponse capturée grâce au détournement de la fonction `lld_pdu_rx_handler` et traiter le PDU `LL_VERSION_IND` ainsi capturé comme le montre le listing 5

Listing 4: Injection d'un PDU LL_VERSION_IND

```
/* LL_VERSION_IND PDU */
uint8_t pdu[] = {0x0C, 0x08, 0x00, 0x00, 0x00, 0x00};

/* Erase version info. */
g_ble_ctrl.version_ble = 0;
g_ble_ctrl.version_compid = 0;
g_ble_ctrl.version_soft = 0;

/* Send VERSION_IND PDU. */
send_raw_data_pdu(
  g_ble_ctrl.conn_handle,
  0x03,
  pdu, // VERSION_IND PDU
  6,
  true
);
```

Listing 5: Traitement de la réponse au PDU LL_VERSION_IND

```
void on_llcp_pdu_handler(uint16_t header, uint8_t *p_pdu, int
↪ length)
{
  uint8_t ble_version;
  uint16_t *comp_id;
  uint16_t *fw_version;

  /* Ensure control PDU is LL_VERSION_IND. */
  if ((p_pdu[0] == 0x0C) && (length == 6))
  {
    /* Display information about target version. */
    ble_version = p_pdu[1];
    comp_id = (uint16_t *)&p_pdu[2];
    fw_version = (uint16_t *)&p_pdu[4];

    /* Keep track of version info. */
    g_ble_ctrl.version_ble = ble_version;
    g_ble_ctrl.version_compid = (uint16_t)(*comp_id);
    g_ble_ctrl.version_soft = (uint16_t)(*fw_version);
  }
}
```

Nous avons implémenté cette technique de prise d'empreinte dans un système portable, en l'occurrence une montre connectée à base d'ESP32 conçue par *Lilygo* [21], et sommes désormais en mesure d'identifier aisément le type de puce Bluetooth Low Energy présente dans les équipements détectés par cette dernière, comme le montre l'image 8. Le code source de l'application ESP32 utilisé pour cette démonstration est sous licence libre (MIT) et disponible sur le dépôt Github du projet [5].

5.2 Support de protocoles non natifs

L'analyse des couches inférieures de la pile protocolaire nous a également amenés à considérer les possibilités de détournement indirectes de celles-ci, afin notamment d'implémenter des stratégies d'attaques inter-protocolaires, visant à interagir et attaquer des protocoles non nativement supportés par la puce. De précédents travaux [4, 10] ont en effet montré que la co-existence de multiples protocoles de communication sans fil dans les mêmes environnements, dont le fonctionnement des couches physiques est proche et utilise les mêmes bandes de fréquences, ouvrait une nouvelle surface d'attaque potentiellement critique.

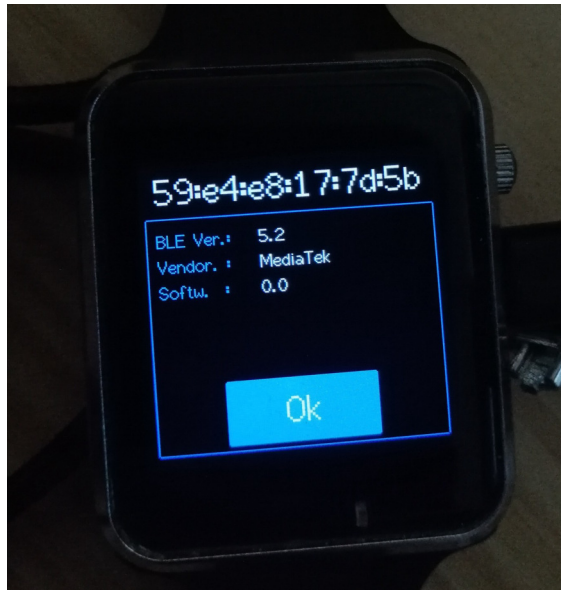

Fig. 8. Prise d'empreinte d'un équipement BLE par une montre connectée T-Watch 2020 de Lilygo.

Primitives de réception et transmission arbitraires L'implémentation de ce type de stratégies inter-protocolaires nécessite un contrôle particulièrement bas niveau sur la pile protocolaire visée, afin d'accéder aux flux de bits en entrée du modulateur et en sortie du démodulateur, mais également de contrôler ou d'altérer divers éléments tels que la fréquence, le débit de données ou les mécanismes de vérification d'intégrité. Dans le cas du contrôleur BLE de l'ESP-32, nous avons détourné les mécanismes liés aux modes de *scanning* et d'*advertising*, afin d'implémenter respectivement une primitive de réception et une primitive de transmission générique, nous permettant de recevoir et d'émettre des trames arbitraires basées sur une modulation de type Gaussian Frequency Shift Keying (GFSK). Ces modes présentent en effet un fonctionnement plus simple que le mode connecté et ne nécessitent pas l'établissement d'une connexion.

En réutilisant la stratégie de hooking décrite en sous-section 5.1, nous avons détourné les fonctions en charge du traitement des PDU d'*advertising*, mais également celles impliquées lors de la configuration des modes considérés. L'instrumentation de ces fonctions de configuration nous a permis d'altérer à la volée divers mécanismes bas niveau du contrôleur, configurés par l'intermédiaire de structures de contrôle stockées dans la zone mémoire d'échange et de registres mappés en mémoire.

Les structures de contrôle exposent une série de registres de 16 bits, définissant le comportement bas niveau du périphérique BLE :

```
09:02: // offset: 0 - CNTL
00:08:
05:10: // offset: 4 - THRCNTL_RATECNTL
6a:5d:
d6:a1:
df:7c:
be:d6: // offset: 12 - SYNCL
8e:89: // offset: 14 - SYNCW
55:55: // offset: 16 - CRCINIT0
55:00: // offset: 18 - CRCINIT1
00:00: // offset: 20 - FILTPOL_RALCNTL
25:00: // offset: 22 - HOPCNTL
0a:00: // offset: 24 - TXRXCNTL
30:80: // offset: 26 - RXWINCNTL
00:14: // offset: 28 - TXDESCPTR
00:00:
30:00:
00:00: // offset: 34 - LLCHMAP0
00:00: // offset: 36 - LLCHMAP1
00:00: // offset: 38 - CHMAP2
00:00: // offset: 40 - RXMAXBUF
```

Nous avons pu identifier et détourner le rôle de plusieurs de ces registres. Il nous a tout d'abord fallu réduire au maximum le nombre de vérifications réalisées automatiquement par le périphérique sur les paquets reçus et transmis, étant donné que nous souhaitions manipuler des trames non conformes à la spécification du BLE. Pour cela, nous avons configuré le registre CNTL pour forcer le format de paquet en *Test mode*. Ce mode, décrit dans la spécification du BLE [3] à des fins de test RF, introduit moins de contraintes sur le format des paquets et évite ainsi certaines vérifications du paquet (telles que la cohérence du format vis à vis du mode configuré) par le périphérique.

Nous avons ensuite détourné les registres SYNCL et SYNCW : ces registres, destinés à configurer l'*access address* utilisée, peuvent être détournés pour servir de mot de synchronisation arbitraire, et détecter des préambules de protocoles non nativement supportés utilisant une modulation proche. Nous avons également modifié le registre RXMAXBUF,

indiquant la taille maximale du buffer de réception des paquets, pour supporter des paquets allant jusqu'à 255 octets.

Le contrôle de la fréquence a nécessité l'altération du registre HOPCNTL, en charge du mécanisme de saut de fréquence et du choix du canal courant. Nous avons pu désactiver le saut de fréquence utilisé en mode *advertising* par l'intermédiaire de ce registre, et configurer le canal sur un canal arbitraire. En l'état, nous sommes cependant limités par les fréquences correspondant aux canaux du BLE. Pour nous affranchir de cette limitation et sélectionner une fréquence arbitraire, nous avons identifié qu'en début de la zone mémoire d'échange, une structure dédiée est utilisée pour stocker un tableau correspondant au *mapping* des canaux du BLE sur leur fréquence respective. Ces valeurs sont stockées sous la forme d'un offset en MHz par rapport à 2402 MHz :

```
00 02 04 06 08 0a 0c 0e 10 12 14 16 18 1a 1c 1e
20 22 24 26 28 2a 2c 2e 30 32 34 36 38 3a 3c 3e
40 42 44 46 48 4a 4c 4e
```

Il est alors possible de forcer une fréquence donnée en modifiant par exemple la dernière valeur du tableau, correspondant au canal 39.

En ce qui concerne la configuration du débit de données, elle est rendue possible sur les variantes ESP32-C3 et ESP32-S3 par l'intermédiaire du registre THRCNTL_RATECNTL. Celui-ci permet en effet de configurer la couche physique LE 1M ou LE 2M, et ainsi de sélectionner un débit au choix entre 1 Mbps et 2 Mbps. La couche physique LE 2M ayant été introduite dans les versions les plus récentes de la spécification Bluetooth, elle n'est pas disponible sur l'ESP32 et le registre correspondant est donc absent de sa structure de contrôle.

Enfin, il a été nécessaire de désactiver le *whitening* et la vérification des CRC. Pour cela, nous pouvons manipuler le registre RWBLECNTL, mappé en mémoire à l'adresse 0x3FF71200 pour l'ESP32 et 0x60031000 pour les variantes ESP32-C3 et ESP32-S3, dont la configuration permet la désactivation de ces fonctionnalités par l'intermédiaire des bits 17 et 18 :

```
RWBLECNTL register definition
 Bits            Field Name   Reset Value
-----   ------------------   -----------
   31       MASTER_SOFT_RST   0
   30     MASTER_TGSOFT_RST   0
   29          REG_SOFT_RST   0
   28             SWINT_REQ   0
```

```
   26          RFTEST_ABORT   0
   25          ADVERT_ABORT   0
   24            SCAN_ABORT   0
   22                MD_DSB   0
   21                SN_DSB   0
   20              NESN_DSB   0
   19             CRYPT_DSB   0
   18              WHIT_DSB   0
   17               CRC_DSB   0
   16         HOP_REMAP_DSB   0
   09         ADVERTFILT_EN   0
   08               RWBLE_EN  0
07:04             RXWINSZDEF  0x0
02:00                SYNCERR  0x0
```

Il est également possible de manipuler le champ SYNCERR, correspondant au nombre d'erreurs tolérées lors de la synchronisation d'un paquet, selon les besoins du protocole considéré.

Une fois cette phase de configuration réalisée, il est possible de réutiliser l'implémentation précédemment décrite pour l'interception et l'injection de PDU BLE, mais cette fois-ci afin de communiquer avec d'autres protocoles non natifs. On dispose ainsi d'une primitive de réception et d'une primitive de transmission de trames modulées en GFSK, configurables par le biais d'un mot de synchronisation de 4 octets à 1 Mbps (ou 2 Mbps dans le cas des variantes ESP32-C3 et ESP32-S3). Il nous est possible de sélectionner une fréquence arbitraire en MHz dans la bande ISM 2,4 GHz, et de désactiver l'ensemble des mécanismes influant sur le contenu du paquet.

Support du protocole ANT Le protocole ANT est un protocole propriétaire développé par Dynastream Innovations Inc., une filiale du groupe Garmin, opérant dans la bande ISM 2,4 GHz. Il est principalement utilisé au sein d'équipements sportifs connectés, tels que des ceintures de monitoring cardiaque ou des montres connectées. Bien qu'il puisse en théorie être utilisé de façon autonome, deux variantes principales sont déployées en pratique :

— **Le protocole ANT+ :** utilisé pour transmettre de faibles volumes de données à intervalle régulier, il spécialise le protocole ANT à des fins d'interopérabilité. Il établit une communication suivant une topologie Maître/Esclave entre des équipements de type capteurs (tels qu'une ceinture de monitoring cardiaque) ou des équipements

destinés à l'affichage ou au traitement de ces informations (tels qu'une montre connectée). De par son large déploiement, il est ainsi utilisé pour la transmission de données de santé potentiellement sensibles.
— **Le protocole ANT-File Sharing (ou ANT-FS) :** utilisé pour la transmission de fichiers entre un équipement jouant le rôle d'un serveur de fichiers (dit *Client*) et d'un client (dit *Hôte*). Il est généralement utilisé pour transférer des rapports et des historiques entre des équipements sportifs et un *dongle USB*, mais également pour la mise à jour *over-the-air* (OTA) du firmware de certains équipements tels que des montres connectées, en en faisant un protocole critique du point de vue de la sécurité.

Bien que le constructeur fournisse une documentation [19] des couches hautes de la pile protocolaire, les couches basses ne sont pas documentées et ont donc nécessité un travail de rétro-ingénierie du protocole, afin de déterminer le format des paquets ainsi que la couche physique employée. Cette rétro-ingénierie a été réalisée par l'intermédiaire d'une analyse en boîte noire des communications radios de différents équipements reposant sur le protocole ANT, et complétée par une analyse statique de l'implémentation de pile protocolaire ANT intégrée sur la puce nRF52. Nous avons ainsi pu déterminer que celui-ci reposait sur une modulation de type GFSK à 1 Mbps, similaire à la couche physique LE 1M du Bluetooth Low Energy. Les paquets ont une taille fixe de 16 octets et sont structurés par le format suivant :

— **Préambule (2 octets) :** il est dérivé par le biais d'un algorithme déterministe d'une séquence de 8 octets (nommée *Network Key* dans la spécification) correspondant à la variante utilisée (ANT+ ou ANT-FS). Sa valeur est 0xa6c5 dans le cas du ANT+ et 0x3ba3 dans le cas du ANT-FS.
— ***Device Number* (2 octets) :** il correspond à un identifiant unique de l'équipement, dépendant généralement du numéro de série, et peut être assimilé à une adresse.
— ***Device Type* (1 octet) :** il identifie le type d'équipement communiquant, selon une série de profils prédéfinis dans la spécification (moniteur cardiaque, compteur de vitesse ...).
— ***Transmission Type* (1 octet) :** il définit un certain nombre de caractéristiques liées au canal de communication.
— ***Header* (1 octet) :** il contient un certain nombre d'informations sur le paquet courant, tels que le mode de diffusion (*broadcast*

ou *unicast*) ou si le paquet est un acquittement d'une donnée précédente.
— ***Données* (7 octets) :** il contient la donnée utile, formatée en fonction du profil considéré.
— ***CRC* (2 octets) :** Code de Redondance Cyclique de 16 bits, basé sur le polynôme 0x1021 et la valeur d'initialisation 0xFFFF (CRC-16 CCITT).

Dans le cas du protocole ANT+, un seul canal de communication, à une fréquence centrale de 2457 MHz, est systématiquement utilisé. Le boutisme utilisé étant différent de celui du Bluetooth Low Energy, nous avons implémenté logiciellement une fonction de conversion permettant de passer facilement du *Little Endian* utilisé par le BLE au *Big Endian* utilisé par ANT. Notre primitive de réception nécessitant la synchronisation sur un motif de 4 octets, nous pouvons procéder en deux étapes afin de nous synchroniser sur une communication ANT+ :

— **Scanning :** on configure la fréquence sur 2457 MHz et utilise un mot de synchronisation correspondant à 0xaaaaa6c5. Ce mot de synchronisation correspond à une phase de bruit, arbitrairement démodulée en 0xaaaa, suivi du préambule utilisé par ANT+ (0xa6c5). On capture ainsi un sous-ensemble de trames ANT+, dont nous pouvons extraire les *Device Numbers* utilisés par les équipements environnants.
— **Sniffing d'une communication donnée :** On modifie le mot de synchronisation afin qu'il corresponde à la concaténation du préambule (0xa6c5) et du *Device Number* correspondant à la communication à sniffer. On dispose alors d'une primitive de réception beaucoup plus fiable, capable de capturer l'ensemble du trafic considéré.

Nous avons pu implémenter cette stratégie avec succès sur les trois variantes du système sur puce étudié (ESP32, ESP32-S3 et ESP32-C3), et sniffer la communication ANT+ établie entre une ceinture de monitoring de rythme cardiaque et une montre connectée Garmin Forerunner 45 afin d'en extraire les informations sensibles telles que la fréquence cardiaque.

Support du protocole Mosart Le protocole Mosart est un protocole de communication sans fil propriétaire basé sur une modulation GFSK à 1 Mbps dans la bande 2,4 GHz, principalement utilisé pour les claviers et souris sans fil. Il est aujourd'hui très massivement déployé et utilisé par de nombreux fabricants (EagleTek, Anger, Advance,...).

Sa rétro-ingénierie a été réalisée par Marc Newlin dans le cadre de ses travaux sur MouseJack [23], une série de vulnérabilités critiques touchant divers claviers et souris sans fil. Un format de trame typique est composé d'un préambule de deux octets (0x5555), d'une adresse sur 4 octets, d'un payload de taille variable et d'un CRC. Un mécanisme de *scrambling* est également appliqué sur chaque trame. Une fois mis en place diverses fonctions de conversion pour traiter l'*endianness* et le *scrambling*, il est possible de mettre en place une stratégie très similaire à celle présentée pour le protocole ANT pour l'implémentation du protocole via nos primitives, en scannant dans un premier temps différents canaux avec un mot de synchronisation représentant du bruit et le préambule. Une fois l'adresse identifiée, il est possible de se synchroniser sur celle-ci pour sniffer le trafic et décoder les frappes claviers ou injecter des frappes claviers arbitraires.

En règle générale, les protocoles propriétaires des claviers et souris sans fil utilisent une modulation GFSK à 1 Mbps ou 2 Mbps dans la bande 2,4 GHz, et seraient donc compatibles avec nos primitives inter-protocolaires. De précédents travaux [12, 23, 27] ont souligné de sévères faiblesses et de nombreuses vulnérabilités sur ce type de protocoles, tels que ceux employés dans les équipements Logitech Unifying ou de Microsoft.

Support du protocole ZigBee Le protocole ZigBee [37] est l'un des protocoles majeurs de l'IoT : il fournit une connectivité sans fil à de nombreux objets connectés aux ressources contraintes, et permet la mise en place de réseaux maillés complexes. Les couches basses de sa pile protocolaire sont définies par la norme 802.15.4, et reposent sur l'utilisation d'une modulation de phase de type Offset-Quadrature Phase Shift Keying (O-QPSK). Il utilise la bande 2,4 GHz en la subdivisant en 16 canaux de communication, espacés de 5 MHz.

Nous avons pu implémenter des primitives de réception et d'émission pour ce protocole en réutilisant l'attaque WazaBee [10]. Cette approche exploite des similarités entre la modulation de fréquence utilisée par le Bluetooth Low Energy et la modulation de phase du ZigBee. Il est ainsi possible de construire une table d'équivalence entre les symboles utilisés par ces deux modulations et de mettre en place une série de fonctions de conversion permettant de passer de l'une à l'autre facilement.

Cette approche reposant sur l'utilisation d'un débit de donnée de 2 Mbps, elle n'a pu être implémentée que sur les variantes de l'ESP32 supportant la couche physique LE 2M. Nous avons ainsi pu sniffer et injecter du trafic ZigBee avec succès depuis les variantes ESP32-C3 et ESP32-S3.

5.3 Détournement des fonctions RF bas-niveau

Au-delà du détournement du contrôleur matériel BLE, nous nous sommes également intéressés au fonctionnement du module radio principal, partagé par les contrôleurs Bluetooth et Wi-Fi. celui-ci est en charge de l'étage analogique des communications radio, et se base sur une architecture hétérodyne classique, gérant la modulation et démodulation en quadrature ainsi que la conversion analogique-numérique. Constatant la présence de nombreuses fonctions manipulant le module radio à bas niveau au sein des fonctions stockées en ROM, nous avons exploré les fonctionnalités bas niveau accessibles et les possibilités de détournement associées.

L'utilisation de la connectivité sans fil, que ce soit par l'intermédiaire du contrôleur Bluetooth, BLE ou Wi-Fi, nécessite une calibration préalable du module radio. Dans la la documentation de l'*Espressif IoT Development Framework* [32], Espressif détaille l'existence de deux types de calibrations : *partielle* ou *complète*. En effet, lors d'une calibration complète, les données issues de la calibration sont stockées dans la mémoire non volatile (NVS), permettant ainsi aux calibrations ultérieures de les réutiliser afin d'éviter une nouvelle calibration complète et d'optimiser la durée de l'initialisation du module radio. Si les données de calibration sont absentes ou si la mémoire non-volatile n'est pas accessible, une calibration complète est automatiquement déclenchée. Ainsi, une calibration complète peut être forcée facilement en stoppant le contrôleur Bluetooth, en supprimant les données de calibration de la NVS et en redémarrant le module radio (listing 6).

Listing 6: Forçage d'une calibration complète

```
// Désactivation du contrôleur Bluetooth et du module radio
esp_bt_controller_shutdown();
// Suppression des données de calibration dans la NVS
esp_phy_erase_cal_data_in_nvs();
// Activation du module radio et déclenchement de la calibration
esp_phy_enable();
```

Les fonctions liées à cette calibration sont, comme dans le cas du contrôleur BLE, stockées en ROM et appelées par l'intermédiaire d'un tableau de pointeurs de fonctions. Ce tableau, stocké en RAM, est illustré par l'image 9. L'adresse de ce tableau peut être récupérée par l'intermédiaire d'une fonction dédiée, `phy_get_romfuncs`. Par conséquent, il est possible de réutiliser la technique de hooking présentée en sous-section 5.1, et d'intercepter les différents appels à une fonction pour exécuter du code d'instrumentation.

```
3ffae0c0 c4 e0 fa 3f     addr        g_phyFuns_instance

                    g_phyFuns_instance
3ffae0c4 6c 2f 00 40     addr        rom_phy_disable_agc
3ffae0c8 88 2f 00 40     addr        rom_phy_enable_agc
3ffae0cc a4 2f 00 40     addr        rom_disable_agc
3ffae0d0 cc 2f 00 40     addr        rom_enable_agc
3ffae0d4 00 30 00 40     addr        rom_phy_disable_cca
3ffae0d8 2c 30 00 40     addr        rom_phy_enable_cca
3ffae0dc 44 30 00 40     addr        rom_pow_usr
3ffae0e0 3c 3e 00 40     addr        rom_gen_rx_gain_table
3ffae0e4 60 30 00 40     addr        rom_set_loopback_gain
3ffae0e8 b8 30 00 40     addr        rom_set_cal_rxdc
3ffae0ec f8 30 00 40     addr        rom_loopback_mode_en
3ffae0f0 2c 31 00 40     addr        rom_get_data_sat
3ffae0f4 a4 31 00 40     addr        rom_set_pbus_mem
3ffae0f8 8c 34 00 40     addr        rom_write_gain_mem
3ffae0fc 1c 35 00 40     addr        rom_rx_gain_force
```

Fig. 9. Tableau de pointeurs des fonctions RF bas niveau, extrait de Ghidra.

La rétro-ingénierie d'une partie de ces fonctions nous a permis de déterminer que, lors d'une calibration complète, le module radio active un mode de *loopback* entre les canaux de transmission (canal TX) et de réception (canal RX) et transmet une série de signaux sinusoïdaux sur le canal de transmission. Cette opération est probablement destinée à ajuster certains paramètres liés au canal de réception. Le mode de loopback étant déclenché par l'intermédiaire de la fonction `rom_loopback_mode_en`, accessible via le pointeur de fonction, il est possible d'exécuter une boucle infinie lors de son déclenchement et ainsi empêcher son activation. Le signal généré est alors transmis directement à l'antenne, et peut être manipulé logiciellement par l'intermédiaire de fonctions bas niveau. Le contrôle de la fréquence centrale est possible en désactivant le contrôle matériel de l'oscillateur grâce à la fonction `phy_dis_hw_set_freq`, puis en spécifiant un offset en MHz par rapport à 2400 MHz par l'intermédiaire du premier paramètre de la fonction `set_chan_freq_sw_start`, comme illustré dans le listing 7.

Listing 7: Envoi d'une sinusoïde sur la fréquence 2420 MHz

```
1 // Désactivation du contrôle matériel de l'oscillateur
2 phy_dis_hw_set_freq();
3 // Configuration de la fréquence centrale à 2420 MHz
4 set_chan_freq_sw_start(20,0,0);
```

Il est également possible de générer deux sinusoïdes indépendantes et de contrôler leurs paramètres par l'intermédiaire de la fonction

`ram_start_tx_tone`. En jouant sur les différents paramètres, nous avons ainsi pu générer plusieurs types de signaux, de la simple sinusoïde à des glitchs particulièrement invasifs. Nous avons mené une série de captures par l'intermédiaire d'une SDR et du logiciel GQRX dans la bande 2,4 GHz illustrant ces différents signaux, comme présenté dans l'image 10.

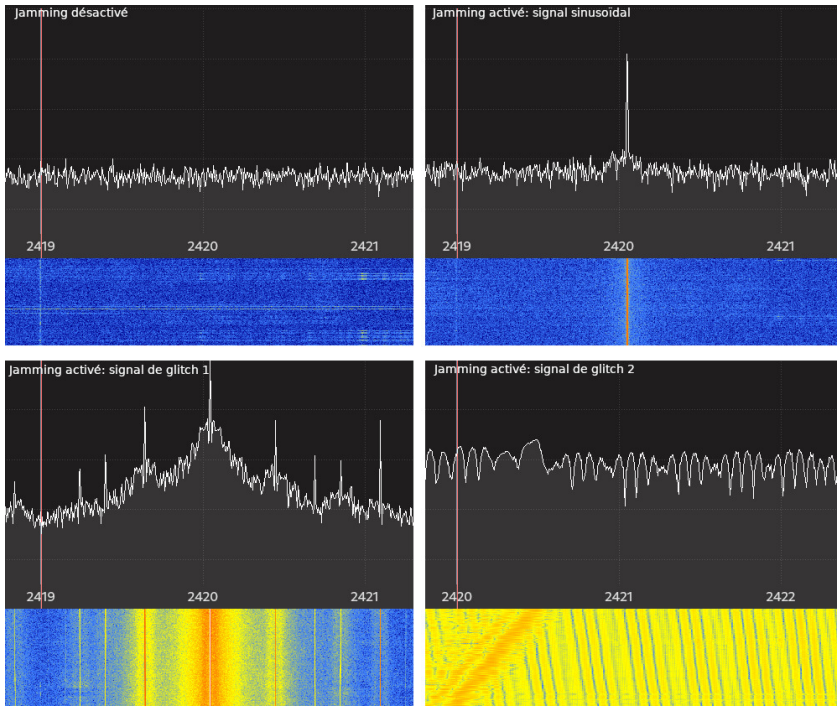

Fig. 10. Différents types de signaux générés via le détournement du processus de calibration, capturés grâce à GQRX.

Ce détournement est particulièrement intéressant d'un point de vue offensif, car il nous offre la possibilité d'impacter le spectre radio en disposant d'un contrôle très bas niveau sur le signal généré. Il nous est ainsi possible de manipuler le signal afin de moduler des données arbitraires pour établir un canal caché, ou d'impacter les communications environnantes en utilisant l'ESP32 comme brouilleur. Nous avons notamment exploré différentes stratégies de *jamming*.

Nous avons mis en place une stratégie de brouillage simultané des trois canaux d'*advertising* utilisés par le Bluetooth Low Energy, en modifiant

la fréquence d'un signal de glitch de façon cyclique entre 2402, 2426 et 2480 MHz présentée dans le listing 8.

Listing 8: Brouillage des canaux d'annonce BLE

```
while (jammer_enabled) {
    // Configuration de la fréquence à 2402 MHz (canal 37)
    set_chan_freq_sw_start(2,0,0);
    // Génération un signal de glitch
    ram_start_tx_tone(1,0,10,0,0,0);
    // Configuration de la fréquence à 2426 MHz (canal 38)
    set_chan_freq_sw_start(26,0,0);
    ram_start_tx_tone(1,0,10,0,0,0)
    // Configuration de la fréquence à 2480 MHz (canal 39)
    set_chan_freq_sw_start(80,0,0);
    ram_start_tx_tone(1,0,10,0,0,0);
}
```

Ces canaux étant nécessaires tant pour la diffusion de données d'annonce que pour l'initiation des connexions, leur brouillage simultané impacte la plupart des fonctionnalités du protocole. Une fois mise en place, cette stratégie nous a permis d'empêcher la réception de tout paquet d'*advertising* par les équipements à proximité de l'ESP32.

En étendant cette approche aux canaux utilisés par le Wi-Fi au sein de la bande 2,4 GHz, nous avons également pu brouiller simultanément l'ensemble des 13 canaux Wi-Fi, provoquant ainsi la déconnexion forcée de tous les équipements connectés et empêchant toute découverte des points d'accès utilisant la bande.

La possibilité de contrôler logiciellement certaines propriétés du signal, et notamment la capacité de modifier très rapidement la fréquence centrale sur une bande de fréquence large, permet ainsi d'envisager la mise en place de stratégies de brouillage complexes permettant d'impacter la disponibilité de nombreux protocoles sans fil utilisant la bande 2,4 GHz.

6 Discussions

Dans cet article, nous avons présenté la rétro-ingénierie d'une pile protocolaire Bluetooth Low Energy embarquée au sein d'une gamme de systèmes sur puce particulièrement répandue, ainsi que le détournement d'un certain nombre de mécanismes logiciels et matériels bas niveau à des fins offensives. Nous avons ainsi pu montrer que ce détournement rendait possible l'implémentation de stratégies d'attaques sans fil complexes, offrant de nouvelles capacités offensives et visant tant le protocole BLE lui

même que les autres protocoles sans fil coexistant dans la même bande de fréquence. Le fait que de telles attaques puissent être implémentées au sein de systèmes sur puce grand public et peu onéreux souligne l'importance d'améliorer la sécurité des nombreux protocoles de communication sans fil co-existant dans la bande ISM 2,4 GHz. Des nombreuses vulnérabilités touchant des protocoles ouverts et activement étudiés tels que le BLE [1, 2, 8, 9, 14] à l'abondance de protocoles sans fils propriétaires reposant principalement sur de la sécurité par l'obscurité [12, 19, 23], le spectre radio est aujourd'hui saturé de communications sans fil, parfois sensibles, basées sur des technologies hétérogènes et potentiellement vulnérables. L'omniprésence de la technologie Bluetooth Low Energy, massivement déployée au sein des objets connectés, des smartphones et des ordinateurs, introduit un risque significatif en terme de surface d'attaque, y compris vis à vis d'équipements n'utilisant pas nativement cette technologie mais pouvant co-exister dans les mêmes environnements, tels que des claviers sans fil ou des capteurs de santé.

Plusieurs scénarios offensifs tirant parti de ces détournements peuvent être soulignés. Il est possible d'implémenter une plateforme d'attaques sans fil inter-protocolaires embarquée, peu onéreuse et mobile. Le développement d'une telle plateforme a été initié dans le cadre de cette recherche, sous la forme d'une montre connectée basée sur un ESP32 et embarquant un firmware offensif open-source exploitant en partie les nouvelles capacités offensives bas niveau décrites dans cet article. Il est également possible d'envisager la compromission d'un équipement contenant un système sur puce ESP32, par l'intermédiaire d'une vulnérabilité menant à une exécution de code ou le détournement d'une mise à jour de firmware *over-the-air*, permettant à un attaquant de déclencher diverses stratégies d'attaques, allant de la bombe logique capable de brouiller les communications sans fil de l'environnement à des scénarios d'attaques pivots inter-protocolaires, en passant par l'écoute de communications sans fil sensibles. Si le déploiement des ESP32 au sein d'équipements commerciaux semble aujourd'hui relativement limité (bien qu'existant, y compris pour des solutions industrielles [22, 36]), il est particulièrement populaire dans les sphères des *makers* et des hobbyistes et fournit de nombreuses fonctionnalités séduisantes pour les développeurs et les fabricants.

Comme souvent avec le détournement de systèmes sur puce à des fins offensives, la marge de manœuvre du fabricant pour prévenir ou corriger ces faiblesses est limitée, les détournements étant principalement rendus possibles par une architecture matérielle et logicielle flexible, ainsi qu'au fonctionnement structurel du protocole BLE. La possibilité d'accéder

directement à des composants bas niveau tels que le module radio, peu courant sur ce type de systèmes embarqués, la facilité d'instrumenter le code bas niveau, ainsi que le faible coût de ces systèmes sur puce et leur polyvalence en font cependant une plateforme prometteuse pour le développement d'outils d'analyse des protocoles sans fils. Les différentes techniques développées ici permettent ainsi de faciliter considérablement l'analyse des couches inférieures du BLE et l'implémentation d'attaques bas niveau innovantes, telles que le fingerprinting d'équipements BLE via l'injection de PDU de contrôle *VERSION_IND*, le brouillage simultané de plusieurs canaux ou l'interaction avec d'autres protocoles sans fil. Dans cette optique et pour des raisons de reproductibilité, nous comptons publier une librairie facilitant l'implémentation de ces stratégies d'attaque ainsi que les différentes preuves de concept associées sous license libre d'ici la date de la conférence.

7 Conclusion

Dans cet article, nous avons présenté une méthodologie de rétro-ingénierie de la pile protocolaire Bluetooth Low Energy embarquée au sein de la gamme de systèmes sur puces ESP32, de son architecture logicielle à ses composants matériels les plus bas niveau. Nous avons également montré qu'il était possible de tirer partie de leur architecture logicielle et matérielle très flexible pour détourner les fonctionnalités bas niveau de celles-ci, afin notamment d'implémenter des stratégies d'attaques complexes, permettant non seulement de manipuler l'ensemble du trafic traité par la couche liaison du protocole Bluetooth Low Energy, mais également d'impacter d'autres protocoles de communication sans fils non nativement supportés, mais présentant des similarités au niveau de leur couche physique et coexistants dans la même bande de fréquence.

Ces travaux nous ont ainsi permis d'explorer en profondeur les capacités offertes par le détournement d'une gamme de systèmes sur puces particulièrement populaire, mais également de mettre en lumière une série de problématiques liés au déploiement chaotique de protocoles de communication sans fil propriétaires dans la bande de fréquence 2,4 GHz. Nous avons pu constater que plusieurs de ces protocoles, tels que ANT+ ou Mosart, manipulent des données potentiellement sensibles tout en offrant peu ou pas de garantie de sécurité vis à vis d'un attaquant passif ou actif. Nous avons montré que la surface d'attaque de ces protocoles est significativement impactée par le déploiement massif d'équipements Bluetooth Low Energy, dans la mesure où la proximité des couches physiques utilisées

par ces différents protocoles et leur coexistence au sein des mêmes environnements rend possible une série d'attaques inter-protocolaires depuis des équipements ne supportant pas nativement ces technologies propriétaires.

En perspective, nous souhaitons poursuivre ces travaux par l'exploration de deux axes complémentaires. La manipulation à bas niveau du trafic niveau liaison développée dans le cadre de cette recherche ouvre de nouvelles perspectives offensives visant le protocole Bluetooth Low Energy : l'implémentation d'une approche de prise d'empreinte des équipements BLE, notamment, permet d'envisager la mise au point d'une approche de détection automatique de vulnérabilités au sein des piles applicatives et protocolaires de ces derniers. Le second axe se concentrera sur la généralisation et la systématisation des problématiques liés à la proximité des couches physiques au sein de protocoles sans fil hétérogènes, afin de mieux comprendre et anticiper cette nouvelle catégorie de menaces.

Références

1. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias : Bluetooth impersonation attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2020.
2. 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.
3. Bluetooth SIG. *Bluetooth Core Specification*, 07 2021. Rev. 5.3.
4. Sergey Bratus, Travis Goodspeed, Ange Albertini, and Debanjum S. Solanky. Fillory of PHY : Toward a periodic table of signal corruption exploits and polyglots in digital radio. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association.
5. Damien Cauquil. Hackwatch firmware. `https://github.com/virtualabs/hackwatch`.
6. Damien Cauquil. Radiobit, a BBC Micro :Bit RF firmware, 2017. `https://github.com/virtualabs/radiobit`.
7. Damien Cauquil. Weaponizing the bbc micro bit. `https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEF%20CON%2025%20-%20Damien-Cauquil-Weaponizing-the-BBC-MicroBit-UPDATED.pdf`, 2017.
8. Damien Cauquil. You'd better secure your BLE devices or we'll kick your butts ! In *DEF CON*, volume 26, 2018. Available at `https://media.defcon.org/DEFCON26/DEFCON26presentations/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf`.
9. 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.

10. Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. WazaBee : attacking Zigbee networks by diverting Bluetooth Low Energy chips. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*, Taipei (virtual), Taiwan, June 2021.
11. cornrn. Freertos implementation for beken bk7231. `https://github.com/cornrn/bk7231u_freertos_sdk`.
12. Rogan Dawes. LogiTacker GitHub Repository, 2019. Available at `https://github.com/RoganDawes/LOGITacker`.
13. Nick Flaherty for eeNews Europe. Espressif moves exclusively to risc-v. `https://www.eenewseurope.com/en/espressif-moves-exclusively-to-risc-v/`, 2022.
14. Matheus Garbelini. Braktooth esp32 br/edr active sniffer/injector. `https://github.com/Matheus-Garbelini/esp32_bluetooth_classic_sniffer`, 2021.
15. 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.
16. Travis Goodspeed. Apimote ieee 802.15.4/zigbee sniffing hardware. `https://www.riverloopsecurity.com/projects/apimote/`.
17. Travis Goodspeed. Promiscuity is the nrf24l01+'s duty. `http://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html`.
18. IEEE. 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.
19. Dynastream Innovation Inc. Ant message protocol and usage, rev 5.1. `https://www.thisisant.com/`.
20. SEEMOO Lab. Bluetooth experimentation framework for broadcom and cypress chips. `https://github.com/seemoo-lab/internalblue`, 2020.
21. Lilygo. Lilygo t-watch 2020. `http://www.lilygo.cn/prod_view.aspx?TypeId=50053&Id=1290&FId=t3:50053:3`.
22. Moduino. Moduino x series - industrial iot controller based on esp32. `https://moduino.techbase.eu`.
23. Marc Newlin. MouseJack : White Paper. In *DEF CON*, volume 24, 2016. Available at `https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/DEFCON-24-Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.whitepaper.pdf`.
24. Marc Newlin. RFStorm nRF24LU1+ Research Firmware GitHub repository, 2016. `https://github.com/BastilleResearch/nrf-research-firmware`.
25. Renesas. Da14681 datasheet. `https://www.renesas.com/us/en/document/dst/da14681-datasheet`.
26. Mike Ryan. Bluetooth : With low energy comes low security. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association.
27. Thorsten Schroeder and Max Moser. Keykeriki resources, 2010. Available at `http://www.remote-exploit.org/articles/keykeriki_v2_0__8211_2_4ghz/`.

28. Matthias Schulz, Daniel Wegemer, and Matthias Hollick. The nexmon firmware analysis and modification framework : Empowering researchers to enhance wi-fi devices. *Computer Communications*, 129 :269–285, 2018.
29. Dominic Spill. Ubertooth One website, 2012. `http://ubertooth.sourceforge.net/`.
30. Espressif Systems. Esp32-c3 series datasheet, version 1.4. `https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf`.
31. Espressif Systems. Esp32 series datasheet, version 4.2. `https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`.
32. Espressif Systems. Espressif iot development framework. official development framework for espressif socs. `https://github.com/espressif/esp-idf`.
33. Espressif Systems. Espressif soc serial bootloader utility. `https://github.com/espressif/esptool`.
34. Mathy Vanhoef and Frank Piessens. Advanced wi-fi attacks using commodity hardware. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 256–265, New York, NY, USA, 2014. Association for Computing Machinery.
35. Joshua Wright. Killerbee : Practical zigbee exploitation framework. `https://www.willhackforsushi.com/presentations/toorcon11-wright.pdf`, 2009.
36. Zerynth. Industrial iot device - 4zerobox. `https://zerynth.com/products/hardware/4zerobox`.
37. Zigbee Alliance. *ZigBee Specification*, 2015.