HiveGuard: A Network Security Monitoring Architecture for Zigbee Networks

Dimitrios-Georgios Akestoridis
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
akestoridis@cmu.edu

Patrick Tague
Department of Electrical and Computer Engineering
Carnegie Mellon University
Moffett Field, California, USA
tague@cmu.edu

***Abstract*—Zigbee networks can be found in a wide range of smart environments that incorporate low-power devices to report sensed events and accept actuation commands wirelessly. However, there is a lack of open-source software tools that consumers can use to monitor their Zigbee networks and ensure that they remain secure. There are several attacks that malicious users can launch against Zigbee networks that would go unnoticed by their network administrators if they are not making use of an appropriate network security monitoring system, which is especially concerning in cases where the Zigbee devices have critical capabilities such as unlocking doors. In this work we introduce the architecture of a distributed system for monitoring the security of Zigbee networks, called HiveGuard. Additionally, we present an energy depletion attack against battery-powered Zigbee devices that we use to test the monitoring capabilities of our prototype implementation. We show that it is possible for an outside attacker to completely deplete the energy of commercial Zigbee devices that are powered by one 3-volt CR2450 lithium battery in less than 16 hours. Our prototype implementation of HiveGuard successfully generated an alert for each attack that we launched and provided additional information about the operation of the Zigbee network for further inspection. We are publicly releasing the source code that we wrote and the packets that we captured during our experiments in order to enable researchers to closely examine our prototype implementation and study novel intrusion detection techniques for Zigbee networks.**

I. INTRODUCTION

Zigbee [1] is a wireless communication protocol that is often used in smart environments for providing networking capabilities to battery-powered sensors and actuators, such as motion sensors and door locks. The lower layers of the Zigbee stack (i.e., the Physical (PHY) and Medium Access Control (MAC) layers) have been defined by the IEEE 802.15.4-2011 standard [2], whereas its upper layers have been defined by the Zigbee Alliance, which consist of the Network (NWK) and Application (APL) layers [3]. Each Zigbee network uses a Personal Area Network Identifier (PAN ID) to distinguish itself from other nearby networks and utilizes either the distributed or the centralized security model. Distributed Zigbee networks support features that lower their security as a trade-off for higher usability, while centralized Zigbee networks include a Zigbee Coordinator that typically operates as the Trust Center to provide higher security [4]. Zigbee devices use a network key to encrypt and authenticate most of their packets with payload in the upper layers, which is shared among all the devices of a Zigbee network. Link keys are used between pairs of Zigbee devices, with one of their most common uses being the transportation of the network key to newly added devices.

Although the Zigbee protocol provides confidentiality and authenticity services for data originating from its upper layers, Zigbee networks remain largely unmonitored. Several types of attacks can be launched against them that their owners would not be able to notice without utilizing a system that can monitor the security of Zigbee networks. For example, Ronen et al. demonstrated a Zigbee worm against distributed Zigbee networks and made use of Zigbee traffic exclusively due to its unmonitored nature [5]. In this work, we are interested in detecting and monitoring attacks that an outside attacker (i.e., a malicious device that has no knowledge of the victim's network key) can launch against centralized Zigbee networks. For instance, Akestoridis et al. demonstrated a set of attacks that take advantage of the fact that Zigbee networks are not utilizing MAC-layer security services [6]. Furthermore, an outside attacker could be motivated by multiple malicious reasons to launch an energy depletion attack against a battery-powered Zigbee device, including the following:

- To prevent the device from notifying its owner about sensed events (e.g., the detection of motion).
- To prevent the device from receiving actuation commands from its owner (e.g., a command to lock a door).
- To trick the owner of the device to factory reset it and potentially expose the network key [6], [7].
- To force frequent battery replacements for the owner of the device, leading to either an increased maintenance cost or abandonment of the technology.

As we explain in Section II, we were able to advance the state of the art in such attacks. While the impact of our attack can be reduced with appropriate firmware updates, the impact of such attacks cannot be eliminated due to the shared nature of the wireless medium. However, the network administrator could utilize a network security monitoring system that can detect such attacks in order to take appropriate actions.

This research was supported in part by CyLab. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of CyLab or Carnegie Mellon University.

Unfortunately, there is a lack of robust open-source software tools that consumers can use to continuously monitor Zigbee traffic and be notified about potential security issues. *This motivated us to develop HiveGuard: a distributed system that interacts with a set of wireless intrusion detection system (WIDS) sensors, a database server, and a notification server to provide archiving, aggregation, inspection, visualization, and alert services.* More specifically, our system operates independently from the monitored Zigbee network and it follows a rule-based approach for the detection of attacks. In summary, our contributions are the following:

- We present the architecture of a network security monitoring system for Zigbee networks, called HiveGuard, and its prototype implementation in JavaScript.
- We enhance existing open-source Python software tools in order to deploy WIDS sensors that are tailored for detecting attacks against Zigbee networks.
- We develop an energy depletion attack in C and test it against commercial battery-powered Zigbee devices to demonstrate our prototype's monitoring capabilities.
- We show that the energy of four commercial Zigbee devices that use a 3-volt CR2450 lithium battery can be depleted by an outside attacker in less than 16 hours.
- We publicly release the source code that we wrote and the packets that our WIDS sensors captured during the energy depletion attacks that we launched.

The rest of this paper is organized as follows. In Section II we review work related to HiveGuard and our attack. Then, we delineate our system architecture and our energy depletion attack in Sections III and IV respectively. Finally, we report our experimental results in Section V and conclude in Section VI.

II. Related Work

Although there are several open-source software tools that are frequently used in network security monitoring deployments, such as Snort [8] and Zeek (formerly known as Bro) [9], most of them are mainly focusing on analyzing traditional IP-based networks. There are multiple communication protocols that are being used in Internet-of-Things (IoT) applications that are not based on IP, with Zigbee being one of the most widely used ones for controlling and monitoring low-power devices [1]. However, extending such software tools to also support non-IP networks is not a trivial task, given the need for appropriate packet dissectors as well as analysis modules for protocol-specific interactions and attack surfaces. For instance, even though Kismet [10] is frequently used as a WIDS for Wi-Fi networks, its support for Zigbee networks appears to be under development for several years. As a result, we decided to develop HiveGuard as a separate software tool that is tailored for monitoring the security of Zigbee networks.

To the best of our knowledge, there is no readily available open-source WIDS that is comprehensive and tailored for detecting attacks against Zigbee networks. BeekeeperWIDS [11] is the only open-source WIDS that we are aware of that focuses on IEEE 802.15.4-based networks, but it is not currently detecting any high-impact attacks against Zigbee networks and it has not been updated since 2014. Furthermore, its drones are simply making HTTP POST requests to push captured packets to a central database for further analysis. In contrast, we designed HiveGuard to periodically fetch collected data, detected events, and compressed pcap files from stand-alone WIDS sensors with HTTP GET requests. Pulling data from sensors is a common design choice for traditional network security monitoring systems [12], as well as systems that monitor real-time data like Prometheus [13], given that the reporting rate is then controlled by the monitoring system itself. We decided to not include BeekeeperWIDS in our experimental setup because, currently, it does not have any modules for detecting the attacks that we describe in this paper.

Sadikin et al. proposed a hybrid intrusion detection system for Zigbee networks [14], but there are several differences between their work and ours. First, they provide little information about the architecture of their system and its implementation. For example, regarding the collection of Zigbee traffic, they simply mention that it is "sniffed and stored to the rule engine for further analysis" [14], while we provide a detailed description of our system design in Section III. Second, Sadikin et al. used a distributed Zigbee network for their experimental setup, whereas we used a centralized one. We prioritized the development of attack detection rules for the most secure configuration of Zigbee networks instead of focusing on attacks that are limited to the distributed security model. Third, while our systems share some attack detection tasks, we cannot compare the technical aspects of our systems because their source code does not seem to be publicly available. By making our prototype implementation available to the public, we enable other researchers to closely examine it and build on top of it for their own projects. Lastly, it appears that their system goals differ from ours. To avoid potential human errors during the development of attack detection rules, Sadikin et al. combined their rule-based approach with the use of machine learning algorithms. On the other hand, we focus specifically on the development of robust attack detection rules to avoid potential false positives due to overfitting.

Since we were interested in testing our prototype's monitoring capabilities during an aggressive energy depletion attack that an outside attacker could launch, we surveyed the literature for relevant attacks. The attack that appeared to be the closest to meeting our requirements was introduced by Cao et al., where the attacker is spoofing packets with valid unencrypted headers, but with the encrypted payload and message integrity code of each spoofed packet containing arbitrary data, so that the receiver will waste its energy receiving them and performing unnecessary security computations [15]. While the core idea of their attack is sound, it appears that they did not consider the fact that real-world battery-powered Zigbee devices are typically using Data Requests to poll for pending packets whenever they enable their receivers [2, p. 39]. Although other researchers have pointed out that Data Requests can be abused for energy depletion purposes [16], we are not aware of any previous work that demonstrates the impact of a fully developed version of such an attack against commercial

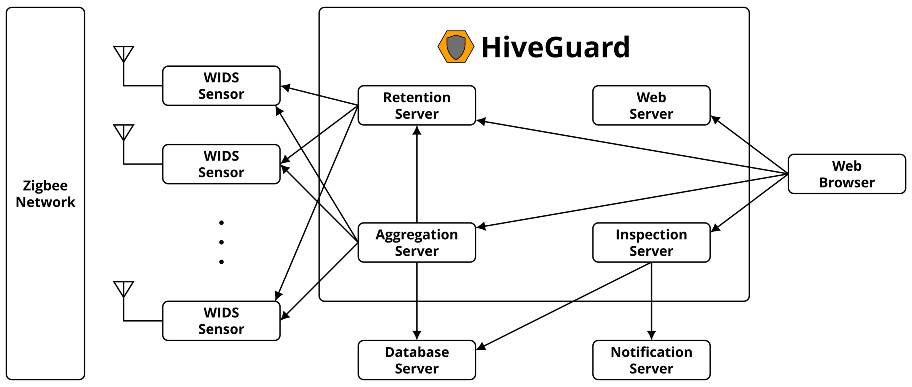


Fig. 1. Overview of the system architecture that we developed for monitoring the security of Zigbee networks.

Zigbee devices. Our attack was specifically designed to exploit the way that Zigbee devices are currently handling Data Requests by selectively jamming them and injecting spoofed packets. In addition, we introduce further improvements to the attacker's main strategy that prevent the targeted device from disconnecting from its network and make the aggressiveness of our attack configurable. Finally, while Cao et al. analyzed their attack using simulations and development boards, we tested our attack against four commercial battery-powered Zigbee devices that can be found in real-world smart homes.

III. System Architecture

As we show in Fig. 1, HiveGuard consists of four components: a retention server, an aggregation server, an inspection server, and a web server. In Section III-A we describe their key responsibilities and how they should interact with the rest components of the architecture, while in Sections III-B and III-C we provide details about our prototype implementation.

A. *Overview of HiveGuard*

The retention server periodically sends HTTP GET requests to the WIDS sensors to retrieve their lists of compressed pcap files and request a copy for each previously unknown file. This process is expected to run on a host machine with sufficient resources for long-term storage of these files. HiveGuard users can then retrieve the full list of archived files from the retention server and download specific ones with HTTP GET requests, while it also accepts HTTP GET and PUT requests regarding the list of WIDS sensors from which it is archiving files.

The aggregation server periodically sends HTTP GET requests to the WIDS sensors to aggregate data about the operation of the Zigbee network and store them in a database. The aggregation server exposes REST API endpoints that enable the registration of new WIDS sensors and deregistration of old ones using HTTP POST and DELETE requests respectively. This information is also stored in the database, with the aggregation server updating the retention server regarding any changes in that list. Furthermore, the aggregation server can collect data about the WIDS sensors themselves, enabling the detection of unexpectedly underutilized or overutilized WIDS sensors. In addition, the aggregation server can make sure that all WIDS sensors are using all known keys, since new ones can be issued during the operation of the Zigbee network.

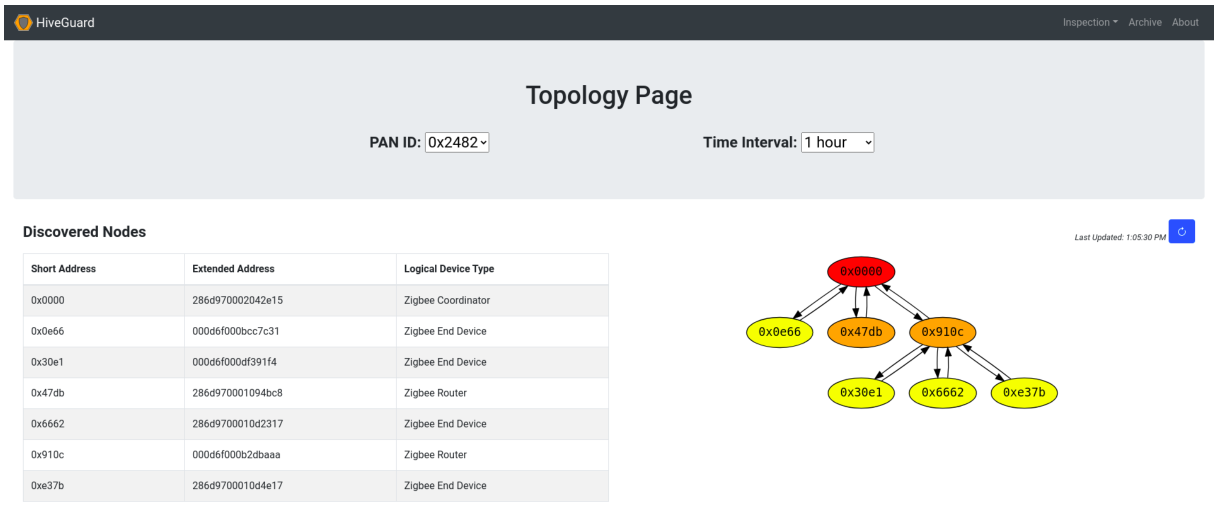
The inspection server periodically processes the data that are stored in the database and exposes them through its REST API endpoints. Furthermore, the inspection server generates alerts for events that were detected either by a WIDS sensor or during its own analysis routine. In our system architecture, the WIDS sensors are performing detection tasks for attacks that can be detected by the stateless examination of a single packet, while the inspection server is performing detection tasks for attacks that require the examination of multiple packets and potentially a holistic view of the Zigbee network. Although the generated alerts can be periodically requested from the frontend application, since the HiveGuard user may not be using it at the time that a critical event was detected, the inspection server should also be able to interact with a server that can notify them within a reasonable amount of time, such as an SMTP server in order to send email notifications.

The web server statically serves the frontend application to run on the HiveGuard user's web browser. The frontend application provides a user interface that enables the network administrator to easily interact with HiveGuard's backend servers. This includes fetching data from the inspection server and presenting them in an appropriate format. At the very least, the frontend application should be capable of generating line charts, bar charts, tables, and graphs. Finally, the frontend application should also enable the network administrator to change the system's configuration and access archived data.

B. *HiveGuard Prototype Implementation Details*

We developed HiveGuard in JavaScript, with its source code being organized into three repositories: one for its command-line interface[1] that has been implemented as a Node.js script [17], one for its backend servers[2] that have been implemented as Node.js modules that utilize the Express framework [18], and one for its frontend application[3] that has been implemented as a set of React components [19]. Our

[1]https://github.com/akestoridis/hiveguard
[2]https://github.com/akestoridis/hiveguard-backend
[3]https://github.com/akestoridis/hiveguard-frontend



Topology Page

PAN ID: 0x2482

Time Interval: 1 hour

Discovered Nodes

Last Updated: 1:05:30 PM

Short Address	Extended Address	Logical Device Type
0x0000	286d970002042e15	Zigbee Coordinator
0x0e66	000d6f000bcc7c31	Zigbee End Device
0x30e1	000d6f000df391f4	Zigbee End Device
0x47db	286d970001094bc8	Zigbee Router
0x6662	286d9700010d2317	Zigbee End Device
0x910c	000d6f000b2dbaaa	Zigbee Router
0xe37b	286d9700010d4e17	Zigbee End Device

Fig. 2. HiveGuard's topology page during the monitoring of a Zigbee network with 7 nodes.

prototype is currently interacting with a PostgreSQL database server [20], whereas our inspection server is currently using the Nodemailer module [21] to send email notifications.

In our current implementation, the inspection server is periodically analyzing NWK auxiliary frame counters to make sure that there are no unexpected decreases in value, since this would indicate that a nearby attacker may be impersonating a device. All alerts are initially considered unread so that they can be requested separately from the archived ones. However, since several alerts with the same message can be generated in a short amount of time, we introduced a configurable cooldown period for the email notification process so that emails will be sent only for alerts whose messages were not included in a recently sent email. We implemented multiple pages for our frontend application that fetch data from the inspection server and then present them in an appropriate format. For example, Fig. 2 demonstrates how our frontend application presents fetched topology data. As for our command-line interface, it allows the user to select the set of HiveGuard's backend servers that they would like to launch, define the necessary environment variables, and override the default configuration.

C. WIDS Sensor Prototype Implementation Details

We wrote our WIDS sensor software on top of Zigator [22] and made enhancements to Scapy [23] in order to dissect the payload of certain Zigbee packets, which we have submitted to their repositories. Essentially, we made Zigator operate as a stand-alone WIDS sensor that can expose REST API endpoints by utilizing the CherryPy framework [24]. Since the WIDS sensors are expected to have very limited storage resources, we made their packet capturing process configurable so that HiveGuard's retention server can archive each compressed pcap file before it would have to be deleted. We now describe the events that our WIDS sensors can currently detect.

PAN ID Conflicts. Security researchers have demonstrated that attackers can disconnect Zigbee devices from their networks by causing PAN ID conflicts [6], [25]. A WIDS sensor can detect PAN ID conflicts simply by examining whether a captured beacon uses the same PAN ID as the user's Zigbee network, but a different Extended PAN ID. Although the firmware of several Zigbee Coordinators has been modified since then to completely ignore PAN ID conflicts, such as the firmware of SmartThings hubs [26], we argue that the network administrator should still be notified about PAN ID conflicts and take appropriate actions based on whether they were malicious or benign ones through closer inspection.

Unsecured Rejoin Requests. Some Zigbee Coordinators accept unsecured rejoin requests, which an attacker can exploit to obtain the victim's network key by spoofing such packets [25], [27]. By monitoring unsecured rejoin requests, the network administrator can identify either the presence of malicious users or benign reconnection attempts from devices that they may decide to replace for security reasons.

Key Leakages. A Zigbee device can receive a key through a Transport-Key command, which is typically protected by a Trust Center link key that the sender and the receiver already share, such as the default one [3, p. 377] or one derived from an install code [28, p. 73]. However, given that the default key is publicly known [7] and install codes can be leaked [6], an attacker may still be able to obtain the transported key. Our WIDS sensors can be configured to detect Transport-Key commands that are protected by a key that an attacker may also have access to. By notifying the network administrator about such events, they can identify when a key could have been leaked to an attacker that was capturing Zigbee traffic.

Low Battery Reports. While some battery-powered Zigbee devices are using the Power Configuration cluster to report their remaining battery percentages [29, p. 3-13], others are using the IAS Zone cluster to report that their battery statuses are below some threshold [29, p. 8-2]. The WIDS sensors should keep track of these low battery reports so that the network administrator can be notified about them and either prepare for a battery replacement or perform an inspection if they are being notified more frequently than expected.

IV. Energy Depletion Attack

Although we were able to test our prototype's monitoring capabilities during the typical operation of a Zigbee network and against single-packet spoofing attacks, we also wanted to make sure that it was capable of monitoring aggressive attacks. For that reason, we developed an energy depletion attack that improves upon the attack that Cao et al. presented [15], as we explained in Section II. In Section IV-A we describe how an outside attacker can exploit the current handling of Data Requests in Zigbee networks to deplete the energy of Zigbee End Devices, while in Section IV-B we provide technical details about our proof-of-concept implementation. Both our attack and our related findings from Section V were responsibly disclosed to the Connectivity Standards Alliance (formerly known as the Zigbee Alliance) in May 2021.

A. Overview of Our Attack

Battery-powered Zigbee devices typically operate as Zigbee End Devices to conserve their energy by disabling their receivers whenever they are idle and relying on a Zigbee Router or the Zigbee Coordinator for routing their packets. Essentially, each Zigbee End Device is considered a child device and directly interacts only with one Zigbee Router or the Zigbee Coordinator, which is considered its parent device and typically keeps its receiver enabled whenever it is idle.

As we show in Fig. 3, the child device sends a Data Request to its parent device to poll for pending packets, shortly after it has enabled its receiver. In this example, the parent device does not carry any pending packets for that child device, so it responds with a MAC acknowledgment that has the Frame Pending field set to zero. Shortly after the reception of that acknowledgment, the child device disables its receiver again. However, it is possible for an outside attacker to interfere with the transmission of Data Requests by selectively jamming them, since Data Requests can be identified as they are being transmitted by observing certain header fields, as we explain in Section IV-B. With the parent device unable to receive the Data Request, the attacker transmits a spoofed MAC acknowledgment in its place with the Frame Pending field set to one. This causes the child device to keep its receiver enabled in anticipation of a pending packet, which the attacker exploits by impersonating its parent device using a spoofed 127-byte packet that contains supposedly encrypted and authenticated data, with its Frame Pending and Acknowledgment Request fields being set to one, that in turn causes the child device to send a MAC acknowledgment and a new Data Request shortly after that. The attacker can then repeat the same steps to prevent the child device from entering its energy-saving sleep mode and make it waste its energy receiving spoofed packets and performing unnecessary security computations.

If the attacker follows the aforementioned process indefinitely, the parent device will not be able to send any legitimate packets to the child device. That can result in the child device disconnecting from its network, which is not a desired outcome for an attacker whose goal is to completely deplete its energy because that would make it stop sending Data Requests. For

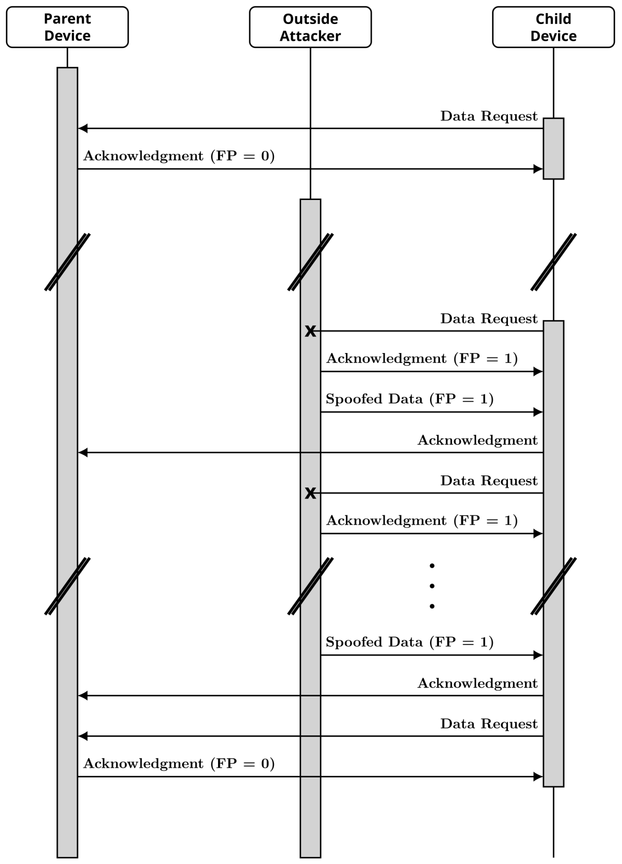


Fig. 3. An outside attacker can prevent a child device from returning to its energy-saving sleep mode by selectively jamming its Data Requests and spoofing packets with supposedly encrypted and authenticated data, while repeatedly claiming that there are additional packets destined for that child device. To prevent the child device from disconnecting from its network, the outside attacker occasionally allows it to communicate with its parent device.

that reason, the attacker is occasionally allowing the Data Requests of the child device to reach its parent device and potentially exchange further packets, either because a specific packet was observed or a specific amount of time has passed since the last time that they were allowed to communicate, as we explain in Section IV-B. The attacker can send spoofed packets for less than the typical time between two Data Requests to go completely unnoticed if there is no appropriate network security monitoring system in place. On the other hand, by sending spoofed packets for longer periods of time, the attacker can deplete the child device's energy faster.

B. Proof-of-Concept Implementation Details

Even though forged Zigbee packets can be injected by using software tools like KillerBee [30] and Zigator [22], selective jamming attacks have to be implemented in firmware to change the transceiver's state in time [6]. We implemented our energy depletion attack in C for an IEEE 802.15.4 USB adapter, called ATUSB [31], by leveraging the framework of the `atusb-attacks` repository [32]. We now describe our proof-of-concept implementation, which we will submit to the `atusb-attacks` repository as the attack with ID 13.

Attack Life Cycle. In our proof-of-concept implementation, we broke down the life cycle of our attack into (a) an active period, (b) an idle period, and (c) a wait period. During the active period, the ATUSB is selectively jamming Data Requests and injecting spoofed packets. After the active period, the ATUSB transitions to the idle period, during which it allows the targeted child device to communicate with its parent device freely. After the idle period, the ATUSB transitions to the wait period, during which it is waiting for a Data Request to transition back to the active period. We configured the Timer/Counter0 of the ATUSB's ATmega32U2 microcontroller so that it can keep track of time [33, p. 91], with the duration of the active and idle periods being specified during the building process of our modified firmware. In our current implementation, we restart the idle period whenever (a) a NWK command was transmitted either by or for the targeted child device or (b) an Association Request was transmitted for the targeted network. The first scenario aids in keeping the devices connected to their network (e.g., by allowing Rejoin Requests), while the second scenario allows the addition of new devices. We also restart the wait period whenever there is a lack of Data Requests during an active period, so that if the targeted child device entered its sleep mode unexpectedly (e.g., because a jamming or spoofing attempt failed), the ATUSB will restart its active period during the next Data Request.

Selective Jamming. An attacker can identify Data Requests as they are being transmitted because, given the set of MAC commands that are typically observed in Zigbee networks [6], these are the only MAC commands that can have a packet length of 12 bytes due to their use of short addresses and the lack of command payload [2, p. 71]. We programmed our ATUSB to process each receiving packet byte by byte, which is supported by its AT86RF231 transceiver [34, p. 126], so that it can transition from its receive state to its transmit state in time to jam a detected Data Request. We also programmed our ATUSB to perform a set of sanity checks for each receiving packet and to make sure that it is destined for the targeted network. Whenever a received packet satisfies all the jamming conditions during an active period, the ATUSB transmits a 1-byte packet to corrupt its Frame Check Sequence (FCS) field, which causes the parent device to discard it. The ATUSB then transmits a spoofed MAC acknowledgment and a spoofed 127-byte packet, as we described in Section IV-A, after which it transitions back to receiving packets one byte at a time.

Attack Validation. In order to validate our proof-of-concept implementation, we captured I/Q signals to closely observe the interactions between an ATUSB with our modified firmware and the commercial Zigbee devices that we describe in Section V. We achieved that by using a USRP N210 [35] and GNU Radio [36], along with the `gr-foo` and `gr-ieee802-15-4` modules [37]–[39], as well as the GRC flow graphs of the `grc-ieee802154` repository [40]. We provide the magnitude of a captured I/Q signal in Fig. 4, where we initially see one of our Zigbee End Devices sending a Data Request to our Zigbee Coordinator for the first time since it exited from its sleep mode. Notice that the Data

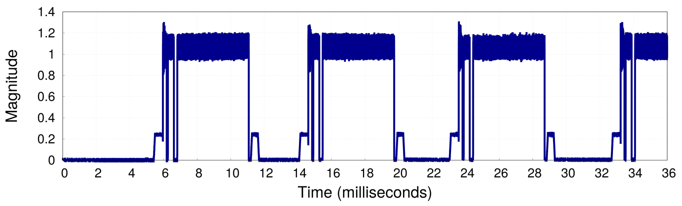


Fig. 4. The magnitude of a captured I/Q signal during the first few milliseconds of our attack against a commercial Zigbee End Device.

Request overlapped with a 1-byte packet that our ATUSB injected, with our Zigbee Coordinator not acknowledging it as a result of that. This was then followed by a spoofed MAC acknowledgment and a spoofed 127-byte packet, with our Zigbee End Device acknowledging the latter, which indicates that it successfully passed its MAC-layer filtering process. Even though the verification process expectedly failed on the NWK layer, our Zigbee End Device still transmitted a new Data Request shortly after that because the Frame Pending field of the spoofed 127-byte packet was set to one. The same pattern was then continued until our ATUSB allowed a Data Request to be successfully received by our Zigbee Coordinator.

V. Experimental Results

We now present the results of the experiments that we conducted in order to test our HiveGuard prototype against our energy depletion attack. Our experimental setup is described in Section V-A, while in Section V-B we discuss our findings.

A. Setup

We conducted four experiments, where in each experiment HiveGuard was monitoring a Zigbee network that consisted of a Zigbee Coordinator and one Zigbee End Device that was powered by a brand new 3-volt CR2450 lithium battery, in an attempt to minimize the number of factors that would contribute towards energy depletion, other than our attack. More specifically, we did not send any actuation commands and we avoided triggering their sensors. An ATUSB with our modified firmware was placed next to the Zigbee Coordinator, while the Zigbee End Device was placed about 4.5 meters away from them. We did not launch our attack until after about one hour had passed since we had paired the Zigbee End Device with the Zigbee Coordinator. Our attack was then launched by the ATUSB, which was configured to set the NWK auxiliary frame counter of spoofed 127-byte packets equal to 2600000, while the duration of each active and idle period was equal to 300 seconds and 3 seconds respectively. The ATUSB continued launching the attack until we noticed that the Zigbee End Device was not transmitting any packets for at least one hour. At the end of each experiment, we stopped the attack and replaced the battery of the Zigbee End Device in order to remove it from its network explicitly before starting the next experiment. The Zigbee Coordinator was a SmartThings Hub (IM6001-V3P01) for all four experiments, while the battery-powered Zigbee devices that we used are shown in Fig. 5 and are named in the first column of Table I.

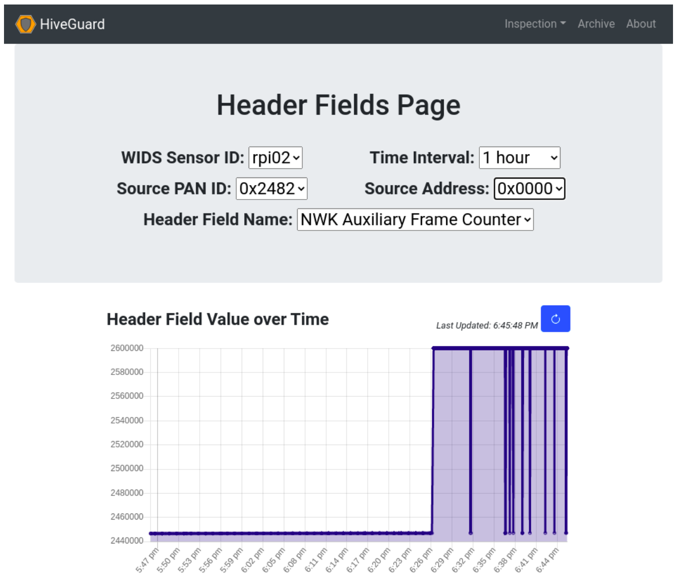
Fig. 5. The commercial battery-powered Zigbee devices that we used.

Fig. 6. One of the WIDS sensors that we used for our prototype.

HiveGuard's backend servers and a PostgreSQL server were running as different processes on the same laptop computer and were interacting with each other over the localhost interface. Our HiveGuard prototype was also interacting with two Raspberry Pis that were equipped with one ATUSB each, as shown in Fig. 6, and our enhanced versions of Scapy and Zigator. We had to cross-compile and flash a Linux kernel image with the ATUSB transceiver driver built into it [41], so that our Raspberry Pis could configure ATUSBs as IEEE 802.15.4 interfaces in monitor mode with wpan-tools [42]. Their interactions took place over a private LAN that we assumed as trusted, so we used plain HTTP request methods, and they were registered to operate as WIDS sensors using `rpi01` and `rpi02` as their identifiers respectively. We placed `rpi01` close to the Zigbee Coordinator and `rpi02` close to the Zigbee End Device, so that we could examine potential differences in their captured packets due to the presence of the selective jammer. We organized the pcap files that HiveGuard archived, from the start of each experiment until the Zigbee End Device was not transmitting any packets for at least one hour, into a dataset that will be available on CRAWDAD [43].

B. Discussion

As we would expect, HiveGuard was able to detect the energy depletion attack and alert us that a nearby device may be impersonating the `0x0000` node of our network, i.e., our Zigbee Coordinator. More specifically, the attack was detected by HiveGuard's inspection server during its periodic analysis of NWK auxiliary frame counters, where it found that there where unexpected decreases in value, which can also be observed from the corresponding page of HiveGuard's frontend application, shown in Fig. 7. For the spoofed 127-byte packet to be processed by the Zigbee End Device, its NWK auxiliary frame counter has to be higher than the one that the Zigbee Coordinator last used. However, the Zigbee Coordinator continues incrementing its NWK auxiliary frame counter normally. As a result, the next time that the Zigbee Coordinator transmits a legitimate packet with NWK-layer security enabled, it will appear as if it decreased in value. Two additional patterns that would give away the presence of such an attacker are the rapid usage of different MAC sequence numbers, as we can observe in Fig. 8, and the increase in



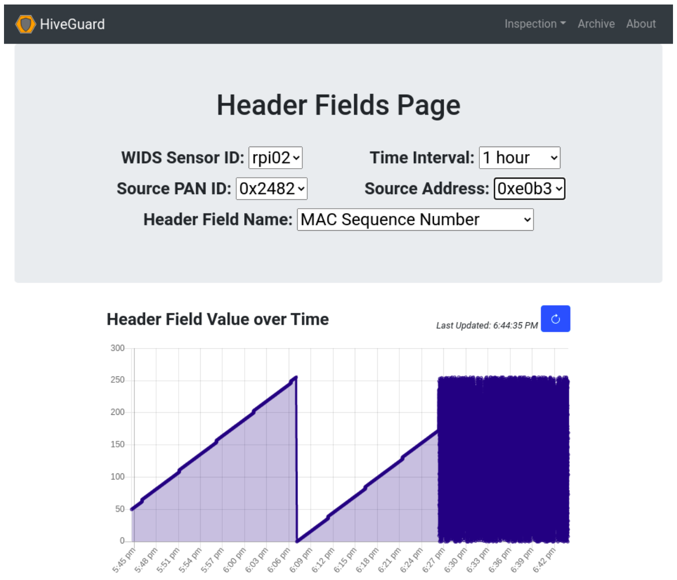
Header Fields Page

WIDS Sensor ID: rpi02 **Time Interval:** 1 hour
Source PAN ID: 0x2482 **Source Address:** 0x0000
Header Field Name: NWK Auxiliary Frame Counter

Header Field Value over Time

Last Updated: 6:45:48 PM

Fig. 7. HiveGuard's header fields page, shortly after the attack against our SmartThings Button started, showing the NWK auxiliary frame counters that our SmartThings Hub used according to the `rpi02` WIDS sensor.



Header Fields Page

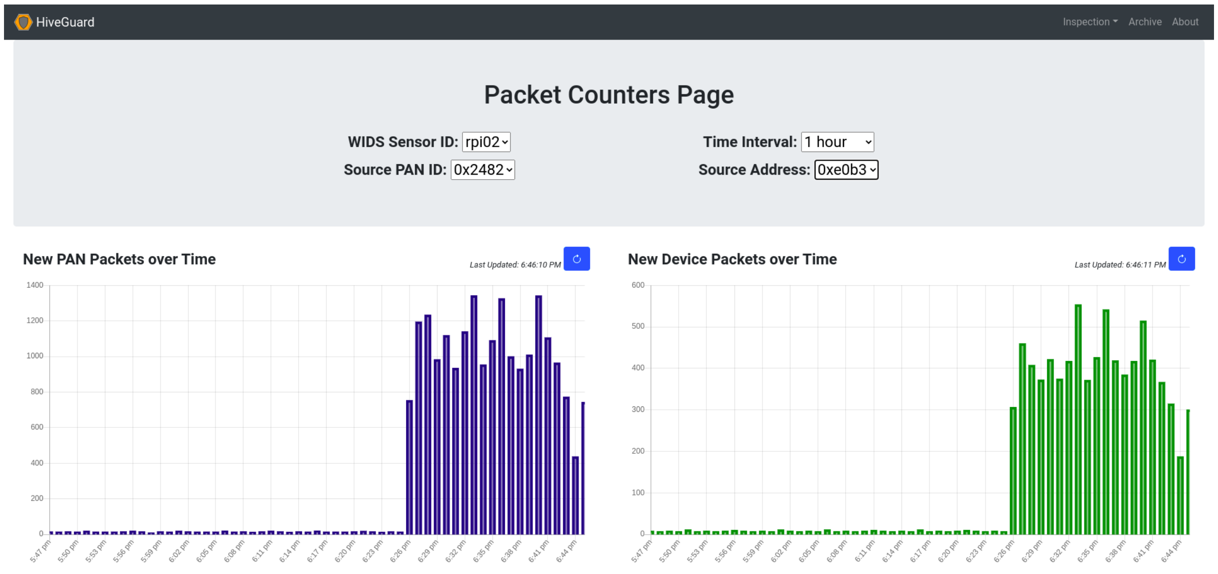
WIDS Sensor ID: rpi02 **Time Interval:** 1 hour
Source PAN ID: 0x2482 **Source Address:** 0xe0b3
Header Field Name: MAC Sequence Number

Header Field Value over Time

Last Updated: 6:44:35 PM

Fig. 8. HiveGuard's header fields page, shortly after the attack against our SmartThings Button started, showing the MAC sequence numbers that our SmartThings Button used according to the `rpi02` WIDS sensor.

the number of new packets per minute, which can be seen in Fig. 9. Furthermore, the observation of packets with valid FCS values but invalid message integrity codes could also be used as another detection rule for such attacks. HiveGuard raised two more unique alerts during our experiments. The first alert was for potential key leakage due to the usage of the default Trust Center link key during the association processes of our SmartThings Motion Sensor and our SmartThings Multipurpose Sensor (F-MLT-US-2). These two devices were also the cause of the second alert, which was about their low battery reports. We did not observe any low battery reports from



HiveGuard

Packet Counters Page

WIDS Sensor ID: rpi02 — Time Interval: 1 hour

Source PAN ID: 0x2482 — Source Address: 0xe0b3

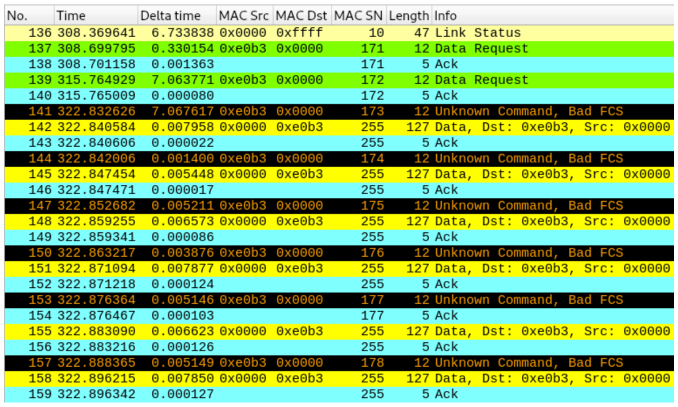
New PAN Packets over Time — Last Updated: 6:46:10 PM

New Device Packets over Time — Last Updated: 6:46:11 PM

Fig. 9. HiveGuard's packet counters page, shortly after the attack against our SmartThings Button started, showing data collected by the `rpi02` WIDS sensor.

our SmartThings Button and our SmartThings Multipurpose Sensor (IM6001-MPP01), while there was no key leakage alert about these two devices because we were able to use their install codes during their association processes.

We broke down each experiment into three phases, which we are using in Table I to report the perceived amount of time that our battery-powered Zigbee devices spent in each phase. The first phase started when our Zigbee End Device began its association process and ended when our ATUSB transmitted the first spoofed 127-byte packet. As we described in Section V-A, this phase lasted about one hour without launching any attack. In Fig. 10 we provide a screenshot from Wireshark's GUI under Zigbee configuration [44], [45] that shows the transition from the first to the second phase of the experiment where our SmartThings Button was used. As we can see, our SmartThings Button was transmitting a Data Request approximately every 7 seconds when there were no pending packets for it. However, once our ATUSB entered its active period, indicated by the 12-byte packets with invalid FCS values due to selective jamming and the spoofed 127-byte packets with supposedly encrypted and authenticated data, our SmartThings Button was transmitting Data Requests much more frequently and was wasting its energy processing our spoofed packets. By the time the second phase ended, our battery-powered Zigbee devices were practically unusable because they stopped sending Data Requests. After that, our battery-powered Zigbee devices repeatedly transmitted either Orphan Notifications or Beacon Requests, which we consider as the third phase. We believe that this behavior change was caused by their very low remaining battery percentages. The end of the third phase was determined by the last captured packet that was transmitted by the targeted device with a valid FCS value. As it can be observed by summing the values in each row of Table I, we were able to deplete the energy of each of our battery-powered Zigbee devices in less than 16 hours, even though we started each experiment by using a 3-volt CR2450 lithium battery that was never used before.



```
No.  Time       Delta time MAC Src MAC Dst MAC SN Length Info
 136 308.369641 6.733838 0x0000 0xffff     10     47 Link Status
 137 308.699795 0.330154 0xe0b3 0x0000    171     12 Data Request
 138 308.701158 0.001363                  171      5 Ack
 139 315.764929 7.063771 0xe0b3 0x0000    172     12 Data Request
 140 315.765009 0.000080                  172      5 Ack
 141 322.832626 7.067617 0xe0b3 0x0000    173     12 Unknown Command, Bad FCS
 142 322.840584 0.007958 0x0000 0xe0b3    255    127 Data, Dst: 0xe0b3, Src: 0x0000
 143 322.840606 0.000022                  255      5 Ack
 144 322.842006 0.001400 0xe0b3 0x0000    174     12 Unknown Command, Bad FCS
 145 322.847454 0.005448 0x0000 0xe0b3    255    127 Data, Dst: 0xe0b3, Src: 0x0000
 146 322.847471 0.000017                  255      5 Ack
 147 322.852682 0.005211 0xe0b3 0x0000    175     12 Unknown Command, Bad FCS
 148 322.859255 0.006573 0x0000 0xe0b3    255    127 Data, Dst: 0xe0b3, Src: 0x0000
 149 322.859341 0.000086                  255      5 Ack
 150 322.863217 0.003876 0xe0b3 0x0000    176     12 Unknown Command, Bad FCS
 151 322.871094 0.007877 0x0000 0xe0b3    255    127 Data, Dst: 0xe0b3, Src: 0x0000
 152 322.871218 0.000124                  255      5 Ack
 153 322.876364 0.005146 0xe0b3 0x0000    177     12 Unknown Command, Bad FCS
 154 322.876467 0.000103                  177      5 Ack
 155 322.883090 0.006623 0x0000 0xe0b3    255    127 Data, Dst: 0xe0b3, Src: 0x0000
 156 322.883216 0.000126                  255      5 Ack
 157 322.888365 0.005149 0xe0b3 0x0000    178     12 Unknown Command, Bad FCS
 158 322.896215 0.007850 0x0000 0xe0b3    255    127 Data, Dst: 0xe0b3, Src: 0x0000
 159 322.896342 0.000127                  255      5 Ack
```

Fig. 10. Captured packets by the `rpi01` WIDS sensor at the beginning of the attack against our SmartThings Button.

VI. Conclusion

In this work we present a distributed system for monitoring the security of Zigbee networks, called HiveGuard, that consists of four components. The first one corresponds to a retention server, whose main purpose is the archiving of compressed pcap files. The second one is an aggregation server, which is mainly responsible for aggregating data and events. The third one corresponds to an inspection server that is responsible for analyzing aggregated data and events, as well as generating alerts. The fourth component is a web server, which is serving HiveGuard's frontend application that in turn provides visualization services. We implemented HiveGuard in JavaScript and enhanced Python tools to deploy WIDS sensors. We also implemented an energy depletion attack in C, which exploits the current handling of Data Requests in

TABLE I
PERCEIVED AMOUNT OF TIME SPENT BY OUR BATTERY-POWERED ZIGBEE DEVICES IN THE DIFFERENT PHASES OF OUR EXPERIMENTS.

Device Name	Time Between the First Beacon Request and the First Spoofed NWK Data	Time Between the First and the Last Spoofed NWK Data	Time Between the Last Spoofed NWK Data and the Last Valid Packet
SmartThings Motion Sensor (F-IRM-US-2)	1.02 hours	6.53 hours	2.66 hours
SmartThings Multipurpose Sensor (F-MLT-US-2)	1.01 hours	3.89 hours	5.58 hours
SmartThings Button (IM6001-BTP01)	1.01 hours	3.93 hours	1.92 hours
SmartThings Multipurpose Sensor (IM6001-MPP01)	1.05 hours	14.17 hours	0.29 hours

Zigbee networks, in order to test our prototype's monitoring capabilities. Our experiments show that it is possible for an outside attacker to deplete the energy of four commercial Zigbee devices, each powered by one 3-volt CR2450 lithium battery, in a relatively short amount of time. Our HiveGuard prototype sent us appropriate notifications about the attacks that we launched and enabled us to examine the operation of our Zigbee network. Finally, we are publicly releasing our source code and our captured packets to enable others to examine them and potentially use them for their own projects.

REFERENCES

[1] Connectivity Standards Alliance. Zigbee. [Online]. Available: https://zigbeealliance.org/solution/zigbee/

[2] *Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE Std. 802.15.4, 2011, doi: 10.1109/IEEESTD.2011.6012487.

[3] *ZigBee Specification*, ZigBee Document 05-3474-21, ZigBee Alliance, 2015.

[4] "zigbee: Securing the wireless IoT," White Paper, ZigBee Alliance, 2017.

[5] E. Ronen, C. O'Flynn, A. Shamir, and A.-O. Weingarten, "IoT goes nuclear: Creating a ZigBee chain reaction," in *Proc. IEEE S&P'17*, 2017, pp. 195–212, doi: 10.1109/SP.2017.14.

[6] D.-G. Akestoridis, M. Harishankar, M. Weber, and P. Tague, "Zigator: Analyzing the security of Zigbee-enabled smart homes," in *Proc. ACM WiSec'20*, 2020, pp. 77–88, doi: 10.1145/3395351.3399363.

[7] T. Zillner and S. Strobl, "ZigBee exploited - the good, the bad and the ugly," presented at Black Hat USA, 2015.

[8] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. USENIX LISA'99*, 1999, pp. 229–238.

[9] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23, pp. 2435–2463, 1999, doi: 10.1016/S1389-1286(99)00112-7.

[10] Kismet Wireless. Kismet. [Online]. Available: https://www.kismetwireless.net/

[11] River Loop Security. A IEEE 802.15.4 wireless intrusion detection system, using the KillerBee framework. [Online]. Available: https://github.com/riverloopsec/beekeeperwids

[12] C. Sanders and J. Smith, "The sensor platform," in *Applied Network Security Monitoring*. Syngress, 2013.

[13] Prometheus. Prometheus - monitoring system & time series database. [Online]. Available: https://prometheus.io/

[14] F. Sadikin, T. van Deursen, and S. Kumar, "A ZigBee intrusion detection system for IoT using secure and efficient data collection," *Internet Things*, vol. 12, p. 100306, 2020, doi: 10.1016/j.iot.2020.100306.

[15] X. Cao, D. M. Shila, Y. Cheng, Z. Yang, Y. Zhou, and J. Chen, "Ghost-in-ZigBee: Energy depletion attack on ZigBee-based wireless networks," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 816–829, 2016, doi: 10.1109/JIOT.2016.2516102.

[16] N. Vidgren, K. Haataja, J. L. Patiño-Andres, J. J. Ramírez-Sanchis, and P. Toivanen, "Security threats in ZigBee-enabled systems: Vulnerability evaluation, practical experiments, countermeasures, and lessons learned," in *Proc. HICSS'13*, 2013, pp. 5132–5138, doi: 10.1109/HICSS.2013.475.

[17] OpenJS Foundation. Node.js. [Online]. Available: https://nodejs.org/en/

[18] expressjs. Fast, unopinionated, minimalist web framework for node. [Online]. Available: https://github.com/expressjs/express

[19] Facebook. A declarative, efficient, and flexible JavaScript library for building user interfaces. [Online]. Available: https://github.com/facebook/react

[20] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. [Online]. Available: https://www.postgresql.org/

[21] nodemailer. Send e-mails with Node.JS. [Online]. Available: https://github.com/nodemailer/nodemailer

[22] D.-G. Akestoridis. Zigator: Security analysis tool for Zigbee networks. [Online]. Available: https://github.com/akestoridis/zigator

[23] SecDev. Scapy: the Python-based interactive packet manipulation program & library. [Online]. Available: https://github.com/secdev/scapy

[24] CherryPy. CherryPy is a pythonic, object-oriented HTTP framework. [Online]. Available: https://github.com/cherrypy/cherrypy

[25] F. Brown and M. Gleason, "ZigBee hacking: Smarter home invasion with ZigDiggity," presented at Black Hat USA, 2019.

[26] SmartThings Community. Hub firmware release notes - 0.31.4. [Online]. Available: https://community.smartthings.com/t/hub-firmware-release-notes-0-31-4/197941

[27] ——. Security of SmartThings ecosystem. [Online]. Available: https://community.smartthings.com/t/security-of-smartthings-ecosystem/30827/5

[28] *Base Device Behavior Specification*, ZigBee Document 13-0402-13, ZigBee Alliance, 2016.

[29] *ZigBee Cluster Library Specification*, ZigBee Document 07-5123, ZigBee Alliance, 2016.

[30] River Loop Security. IEEE 802.15.4/ZigBee security research toolkit. [Online]. Available: https://github.com/riverloopsec/killerbee

[31] Qi Hardware Inc. Ben-WPAN overview. [Online]. Available: http://downloads.qi-hardware.com/people/werner/wpan/web/

[32] D.-G. Akestoridis. Modified ATUSB firmware that supports selective jamming and spoofing attacks. [Online]. Available: https://github.com/akestoridis/atusb-attacks

[33] *ATmega8U2/16U2/32U2 datasheet*, 7799E–AVR–09/2012, Atmel Corporation, 2012.

[34] *AT86RF231/ZU/ZF datasheet*, 8111C–MCU Wireless–09/09, Atmel Corporation, 2009.

[35] Ettus Research. USRP N210 Software Defined Radio (SDR). [Online]. Available: https://www.ettus.com/all-products/un210-kit/

[36] GNU Radio. GNU Radio – the free and open software radio ecosystem. [Online]. Available: https://github.com/gnuradio/gnuradio

[37] B. Bloessl. Some GNU Radio blocks that I use. [Online]. Available: https://github.com/bastibl/gr-foo

[38] ——. IEEE 802.15.4 ZigBee transceiver. [Online]. Available: https://github.com/bastibl/gr-ieee802-15-4

[39] B. Bloessl, C. Leitner, F. Dressler, and C. Sommer, "A GNU Radio-based IEEE 802.15.4 testbed," in *Proc. FGSN'13*, 2013, pp. 37–40.

[40] D.-G. Akestoridis. A collection of GNU Radio Companion flow graphs for the inspection of IEEE 802.15.4-based networks. [Online]. Available: https://github.com/akestoridis/grc-ieee802154

[41] Raspberry Pi. Kernel source tree for Raspberry Pi Foundation-provided kernel builds. [Online]. Available: https://github.com/raspberrypi/linux

[42] linux-wpan. Userspace tools for Linux IEEE 802.15.4 stack. [Online]. Available: https://github.com/linux-wpan/wpan-tools

[43] CRAWDAD. CRAWDAD: A community resource for archiving wireless data at Dartmouth. [Online]. Available: https://crawdad.org/

[44] Wireshark Foundation. Wireshark's official Git repository. [Online]. Available: https://gitlab.com/wireshark/wireshark

[45] D.-G. Akestoridis. Wireshark configuration profile for Zigbee traffic. [Online]. Available: https://github.com/akestoridis/wireshark-zigbee-profile