Spring 5-15-2022

SPLICEcube Architecture: An Extensible Wi-Fi Monitoring Architecture for Smart-Home Networks

Namya Malik
Namya.Malik.GR@Dartmouth.edu

SPLICECUBE ARCHITECTURE: AN EXTENSIBLE WI-FI MONITORING ARCHITECTURE FOR SMART-HOME NETWORKS

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of

Master of Science

in

Computer Science

by Namya Malik

Guarini School of Graduate and Advanced Studies
Dartmouth College
Hanover, New Hampshire

May 2022

Examining Committee:

David Kotz, Chair

Timothy Pierson

Xia Zhou

F. Jon Kull, Ph.D.
Dean of the Guarini School of Graduate and Advanced Studies

Abstract

The vision of smart homes is rapidly becoming a reality, as the Internet of Things and other smart devices are deployed widely. Although smart devices offer convenience, they also create a significant management problem for home residents. With a large number and variety of devices in the home, residents may find it difficult to monitor, or even locate, devices. A central controller that brings all the home's smart devices under secure management and a unified interface would help homeowners and residents track and manage their devices.

We envision a solution called the SPLICEcube whose goal is to detect smart devices, locate them in three dimensions within the home, securely monitor their network traffic, and keep an inventory of devices and important device information throughout the device's lifecycle. The SPLICEcube system consists of the following components: 1) a main *cube*, which is a centralized hub that incorporates and expands on the functionality of the home router, 2) a *database* that holds network data, and 3) a set of support *cubelets* that can be used to extend the range of the network and assist in gathering network data.

To deliver this vision of identifying, securing, and managing smart devices, we introduce an architecture that facilitates intelligent research applications (such as network anomaly detection, intrusion detection, device localization, and device firmware updates) to be integrated into the SPLICEcube. In this thesis, we design a general-purpose Wi-Fi architecture that underpins the SPLICEcube. The architecture specif-

ically showcases the functionality of the cubelets (Wi-Fi frame detection, Wi-Fi frame parsing, and transmission to cube), the functionality of the cube (routing, reception from cubelets, information storage, data disposal, and research application integration), and the functionality of the database (network data storage). We build and evaluate a prototype implementation to demonstrate our approach is *scalable* to accommodate new devices and *extensible* to support different applications. Specifically, we demonstrate a successful proof-of-concept use of the SPLICEcube architecture by integrating a security research application: an "Inside-Outside detection" system that classifies an observed Wi-Fi device as being inside or outside the home.

Acknowledgements

First, I would like to thank my advisor Professor David Kotz for giving me the opportunity to pursue research work, providing invaluable guidance, and holding my work to a high standard. I would also like to thank Professor Timothy Pierson who provided patient support and insightful feedback at many stages of this project.

A special thank you to Chixiang Wang who helped me conduct my experiments and also provided insights into the Inside-Outside application.

Thank you to Tina Pavlovich who helped purchase project materials and coordinate meetings and project logistics.

Finally, I would like to thank my family and friends who have supported me throughout this experience.

SPLICE research is supported primarily by the National Science Foundation under NSF award number CNS-1955805. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contents

Abstract ii

Acknowledgements iv

1 Introduction **1**

1.1 Problem Statement 2

1.2 Proposed Solution 2

1.3 Contributions 6

2 Background **7**

2.1 Home IoT 7

2.2 Key Stakeholders 9

2.3 Architecture Attributes 10

3 Related Work **11**

3.1 Commercial 11

3.2 Literature 12

4 Architecture **14**

4.1 Conceptual Architecture 14

4.2 Cubelets 15

4.2.1 Capturer Module 15

4.2.2 Parser Module 16

4.2.3 Transmitter Module 16
4.2.4 Support of Architecture Attributes 16
4.3 Cube 17
4.3.1 Router Module 17
4.3.2 Reception Module 19
4.3.3 Storage Module 19
4.3.4 Disposal Module 19
4.3.5 Integration Module 20
4.3.6 Support of Architecture Attributes 20
4.4 Database 21
4.4.1 Support of Architecture Attributes 21

5 Implementation **22**
5.1 Cubelets 22
5.1.1 Thread 1: Capturer 25
5.1.2 Thread 2: Parser 27
5.1.3 Thread 3: Transmitter 30
5.2 Cube 33
5.2.1 Thread 1: Reception 36
5.2.2 Thread 2: Storage 37
5.2.3 Thread 3: Integration 38
5.3 Database 42
5.3.1 Database Structure 42
5.3.2 Database Inserts 44
5.3.3 Database Queries 45
5.3.4 Design choices for database 46

6 Experimental Evaluation **48**

6.1 Experiment 1 48

6.1.1 Testing Procedure 48

6.1.2 Results 52

6.2 Experiment 2 55

6.2.1 Testing Procedure 55

6.2.2 Results 57

6.3 Experimental 3 60

6.4 General Storage Requirements 61

6.5 Experimental Support 62

7 Future Work **63**

7.1 Implementation Improvements 63

7.1.1 Sniff on Multiple Channels 63

7.1.2 Decrypt MAC Frame Data 64

7.1.3 Implement Disposal Module 65

7.1.4 Translate Code to C 66

7.1.5 Conduct More Robust Testing 66

7.1.6 Modify Database 67

7.2 Architecture Improvements 68

7.2.1 Expand Router Functionality 68

7.2.2 Accommodate Other Applications 69

7.2.3 Support Other Protocols 70

7.2.4 Incorporate Security as an Architecture Attribute 70

7.2.5 Optimize the Positioning of the Cubelets 71

8 Conclusion **72**

List of Tables

5.1 Inside-Outside Application Merged Data 42

6.1 Experiment 1 ML Model Metrics 53

6.2 Experiment 2 Database Metrics 61

List of Figures

1.1 SPLICEcube System in a Home 3

4.1 Conceptual Architecture 15
4.2 Cubelet Architecture 16
4.3 Cube Architecture 18

5.1 Cubelet Prototype 23
5.2 Cubelet Threads 24
5.3 Data Link Layer 25
5.4 802.11 MAC Frame Format 26
5.5 MAC Header Format 27
5.6 Cube Prototype 34
5.7 Cube Threads 36
5.8 Database Entity Relationship Diagram (ERD) 44

6.1 Experiment 1 Testing Site 49
6.2 Experiment 1 Testing Site Setup 50
6.3 Experiment 1 Query Time Result for Data Collection Run #1 54
6.4 Experiment 1 Query Time Result for Data Collection Run #2 54
6.5 Experiment 1 Query Time Result for Data Collection Run #3 55
6.6 Experiment 2 Query Time Result for the Home Site 57

6.7 Experiment 2 Query Time Result for the Lab Site 58
6.8 Experiment 2 Insertion Time Result for the Home Site 59
6.9 Experiment 2 Insertion Time Result for the Lab Site 59
6.10 Experiment 3 Query Time Result 60

Chapter 1

Introduction

With the explosion of the Internet of Things (IoT), more smart devices are rapidly entering the home. Light bulbs regulate their brightness based on the available natural light or the current use of the room. TVs connect to the Internet to stream video or music. Refrigerators monitor food quantities and maintain shopping lists. Thermostats are equipped with Wi-Fi and allow residents to remotely control temperatures. Virtual assistants are networked and often perform tasks and answer questions by accessing the Internet. Doorlocks recognize when residents are near and automatically grant access to the home [22]. IoT is becoming a staple in many homes, and almost half of U.S. households will use a smart home device by 2025 [25].

Smart devices in the home provide unprecedented convenience and comfort by automating tasks and allowing remote access, but they also create a management challenge for residents. Without a way to track and monitor the devices, residents may be unaware of a device's communication patterns or even its location in the home. An adversary could plant a rogue device inside or near the home that eavesdrops on conversations. A device that the resident initially installed in the home could be compromised and communicating with external, malicious sources. To avoid these security exploits, it is important for a resident to be able to easily access and monitor

all the devices in the home.

Section 1.1

Problem Statement

There is currently no system to support residents who install a multitude of different types of smart devices in their home, and who desire privacy, security, and usability. To realize a vision of a manageable and secure smart home environment, we must bring all the home's smart devices under a secure and unified management interface, that is underpinned by a scalable and extensible architecture.

Specifically, the implementation of such an architecture needs to be able to capture network data across the smart home, parse the network data for desired information, consolidate this information into a central repository, and allow applications to use the stored information to securely manage the devices within the home. The number and type of devices are expected to continue to increase within the smart home, so the architecture should scale with network activity and be extensible to allow different network technologies and different types of applications.

Section 1.2

Proposed Solution

We envision a system called the SPLICEcube (SPLICE = Security and Privacy in the Lifecycle of IoT for Consumer Environments) whose goal is to detect smart devices, locate them in three dimensions within the home, securely monitor their network traffic, and keep an inventory of devices and important device information throughout the device's lifecycle.

The SPLICEcube is composed of

- a main *cube*, a centralized hub that incorporates and expands on the functionality of the home router, and
- a *database* that holds network data, and
- a set of support *cubelets* to extend the range of the cube and to assist in gathering network data in the home.

The SLPICEcube system can be deployed in a home, as shown in Figure 1.1.

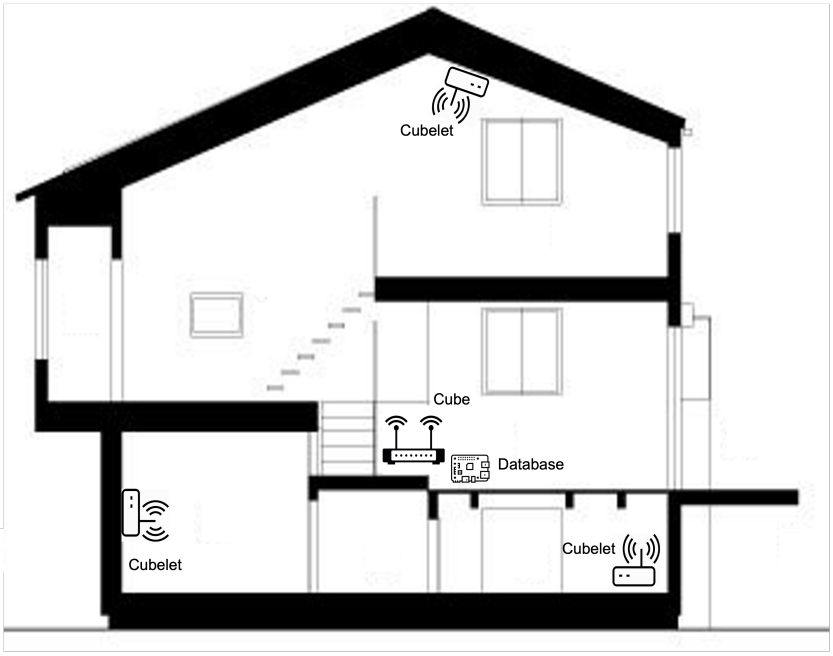

Figure 1.1: **SPLICEcube System in a Home**. The cube and database are located centrally, and the cubelets are dispersed throughout the home.

Our vision is for the SPLICEcube to be able to detect smart devices, securely monitor their network traffic, and keep an inventory of devices and important device information throughout the device's lifecycle. To function as a home's central security and privacy controller and to manage devices throughout their lifecycles, the SPLICEcube must have a comprehensive and robust array of capabilities.

To develop this broad array of capabilities, we envision the SPLICEcube to be able to seamlessly incorporate intelligent research applications that tackle different tech-

nical questions and consequently expand the functionality of the system. Researchers write these applications and work with the SPLICEcube developer to integrate the applications with the SPLICEcube. Smart-home residents who deploy the SPLICEcube in their home then use these applications to secure and manage their smart-home environment. Hence, we envision that the SPLICEcube serves as both a research platform for researchers to test applications and an end-user platform for residents to manage their smart home.

Below are some examples of applications that could be incorporated into the SPLICEcube to enhance its functionality.

- Device Localization: an application to detect the physical location of a device inside or near the home. Several research approaches exist that leverage different characteristics of device transmissions to estimate location. For example, Soltanaghaei et al. propose multipath triangulation, a method that extracts features such as angle of arrival, angle of departure, and relative time of flight, from multipath signals to help triangulate the position of the transmitter relative to the receiver [42]. In contrast, the Inside-Out Detection system proposed by Gralla uses received signal strength indicator (RSSI) data from device Wi-Fi transmissions to classify the device as inside or outside the boundary of a residence [14].

- Network Anomaly Detection: this application recognizes if a device is deviating from its expected behavior, such as if it is communicating with an unrecognized cloud service. Two approaches to anomaly detection are learning-based and specification-based. A *learning-based approach* sniffs network traffic, identifies and extracts certain parameters, makes a model of (learns) the "normal" behavior, and finally compares the behavior of the running system with the one defined by the model [12]. For example, IoTHound uses an unsupervised learn-

ing method to analyze properties of the network traffic to 1) identify IoT device types based on extracted data, and 2) detect deviations from normal network behavior by monitoring over time [3]. In contrast, a *specification-based approach* does not learn but instead uses specifications, parameters, and measurements of the system, that are often stipulated by the manufacturer, to identify when a system behaves unexpectedly [12]. Manufacturer Usage Description (MUD) is an embedded software standard that allows IoT device manufacturers to define device specifications, including the intended communication patterns for their device when it connects to the network [23]. The BoDMitM (Botnet Detection and Mitigation System For Home Router Based on Manufacture Usage Description), monitors network traffic and forwards any traffic that violates the device's MUD policies to an intrusion detection system, which subsequently attempts to identify the attack vector [15].

- Device Firmware Update: a method to securely deploy new firmware on multiple, heterogeneous devices. Nilsson et al. present a wireless firmware update protocol that provides data integrity, data authentication, data confidentiality, and freshness and uses a single central unit to communicate with a number of end nodes [32].

- Cube Interface Design: an application to make the cube interface user-friendly and may include a front-end GUI and a mobile application.

There must be an architecture foundation that underpins the SPLICEcube and facilitates the integration of these research applications. For example, to accomplish network anomaly detection, there must be a way for the system to establish a baseline for device network activity (whether from network observations or from a specification such as MUD) and then recognize anomalous behavior. To accomplish device localization, there must be a way for the system to capture device activity and

estimate physical distance to the device or leverage known observation locations to implement triangulation or trilateration. Although the cube (as a central hub) and cubelets (as sniffers) joint system is apt for exploring these technical questions, several questions arise. What information should the cubelets capture? How should the cubelets communicate with the cube? How often should this communication occur? Should the cube analyze all information or should some intelligence be distributed to the cubelets? In what format should information be stored?

To enable the SPLICEcube vision, we designed and built a general-purpose Wi-Fi architecture and database for the SPLICEcube that also provides a foundation for further research development.

Section 1.3

Contributions

The key contributions of this thesis are

- the development of a general-purpose Wi-Fi architecture and database that underpins the SPLICEcube and facilitates the integration of intelligent research applications, and
- a successful proof-of-concept use of the SPLICEcube architecture through the integration of the Inside-Outside detection system, an application that classifies an observed Wi-Fi device as being inside or outside the home.

Chapter 2

Background

Since the architecture proposed in this thesis is tailored to the smart home environment, it is important to have an understanding of home IoT, the key stakeholders that interact with the SPLICEcube system, and the attributes that the architecture must satisfy to be useful in a smart home context. We address these topics in this chapter.

Section 2.1

Home IoT

We draw on Kotz and Peters' following definitions of IoT, *smart things*, and *smart environment* for the purposes of this thesis [21].

- "IoT refers to the *Internet of Things*, a vision in which everyday objects become *smart things* through the inclusion of digital electronics and a network interface that allows them to communicate with other Things and remote servers on the Internet."
- "*Smart things* typically have the ability to interact with their environment through sensors and actuators. They vary in size, they may be stationary or

mobile, and they may or may not have a human user interface." In this thesis, we use *smart device* as a synonym for smart thing.

- "A *smart environment* is an environment involving a collection of smart things that interact with the environment, with its human occupants, with each other, and with remote services." In this thesis, we focus on a *smart home*, a smart environment that is used as a residence.

Incorporating smart devices in the home has benefits, but it also poses significant privacy, security, and usability risks.

- Privacy: The home is inherently a private space where actions, conversations, and living patterns are typically shielded from outsiders. Smart home devices that can listen, record, remember habits, and communicate over the Internet diminish the barrier between the inside of the home and the outside world. These devices introduce many interfaces through which private information can escape the home.

- Security: Having a large number and variety of devices makes it challenging for a home resident to manage these devices. A lack of awareness about devices can introduce security vulnerabilities. If a resident cannot remember or is not alerted when a device is due for a firmware upgrade, an adversary could take advantage of outdated components and hijack the device. If a device does become compromised, an adversary could run bots or install a crypto-miner on the device, causing it deviate from its regular communication patterns. Additionally, an adversary could install their own physical devices near the home such as an access point that residents may unknowingly connect to, or a device that eavesdrops on residents' conversation or even records residents' actions.

- Usability: Introducing a diverse set of smart devices to the home also makes it more difficult for the resident to use the devices. Different devices may utilize different communication protocols or have different configuration settings. To harness the full functionality of devices and take full advantage of the benefits that smart homes offer, a resident must be knowledgeable about the state of the smart home and have complete control of the devices within it.

Section 2.2

Key Stakeholders

In this section, we define the key stakeholders that interact with the SPLICEcube system.

- SPLICEcube developer: a person writing code to develop the SPLICEcube. They have access to and can modify the code for any of the different components of the SPLICEcube system.

- Researcher: a person who is working on a research application and wishes to integrate the application into the SPLICEcube. They do not have access to any SPLICEcube code and cannot modify any of the SPLICEcube system components. They can communicate their research idea to the SPLICEcube developer, and the SPLICEcube developer can then modify the system to integrate the application and subsequently allow the researcher to request network data if necessary. Allowing only the SPLICEcube developer to access the code helps maintain the integrity of the SPLICEcube system.

- Resident: a person living in a smart home. They do not have access to any SPLICEcube code and cannot modify any of the SPLICEcube system components. They can use applications that are already designed to work with the

SPLICEcube to secure and manage their smart home environment.

Section 2.3

Architecture Attributes

The proposed architecture is designed for an evolving, general smart home space. To deliver this vision, we define two attributes that the architecture must satisfy.

First, the architecture must be *scalable* to accommodate increased network traffic and new devices. Different smart home environments may have different number of existing devices, and the network traffic will vary by home. Additionally, residents may want to add more devices after the SPLICEcube has been installed in the home, which may further increase network traffic. The architecture should be able to accommodate various levels of network activity and efficiently manage network data.

Second, the architecture must be *extensible* to support different applications and technologies. Since the SPLICEcube serves as a platform for researchers to deploy and test research applications and as a platform for residents to manage their smart home, the SPLICEcube must be flexible enough to incorporate such applications into its existing framework or be easily extended to accommodate the application. An extensible architecture enables the exploration and customization we envision, when the SPLICEcube is used as a research platform. We focused on designing a Wi-Fi architecture due to the pervasiveness of Wi-Fi as a communication protocol. However, we realize that devices in the home use other protocols such as Bluetooth, Zigbee, or Z-Wave. Additionally, other newer IoT protocols that are currently in development may become increasingly popular. Although our proposed architecture does not currently support these protocols, it could be extended to support such additional technologies.

Chapter 3

Related Work

In this chapter we describe existing work that is similar to the SPLICEcube and that is available commercially or has been described in publically available literature.

Section 3.1

Commercial

Smart home platforms such as Samsung SmartThings [41], Google Home Assistant [13], Amazon Alexa [1], and Apple HomeKit [4] let residents control smart-home devices. These platforms, however, are mostly designed for residents to automate their smart home and not for detailed network analysis. For example, these platforms allow residents to remotely dim the smart lights or raise the smart thermostat's temperature, but they do not alert the resident if the smart TV is deviating from its regular communication behavior.

Commercial routers such as the Netgear Orbi mesh Wi-Fi system provide a friendly web user interface that shows all the connected devices on the network and provides some basic statistics about the network traffic [31]. These products, however, do not provide robust, comprehensive applications for detailed network analysis.

Several commercial security products are available, but they have limited capa-

bilities. Bitdefender BOX is a router designed to protect smart homes by blocking malicious Internet traffic; in effect, it is a consumer-grade firewall intrusion detection and prevention device [7]. The Bull-Guard Dojo claims to protect home connected IoT devices in the home from malware, viruses, and cyberattacks while keeping privacy intact [9]. Its capabilities are unclear, but appear limited to firewall and anomaly detection.

Section 3.2

Literature

We focus our literature search on home hubs and security managers designed to improve security or privacy for smart-home IoT devices.

The Databox is a personal networked device installed in the home and can access a user's personal data from a variety of sources – online, mobile, IoT [2]. Its focus, however, is not on securing and managing smart home devices; instead, its primary purpose is to protect the privacy of the user by securing personal data.

The IoT Inspector is an open-source software that captures network traffic and provides visualizations of device activity [17, 19]. It is well-suited to analyze device behavior and identify anomalous communication. It, however, does not have other features that we envision for the SPLICEcube such as device localization and secure device firmware upgrades.

The Home Manager is a software infrastructure tool that aims to provide usable and secure management of cooperating IoT devices [6]. It has several capabilities including detecting devices, controlling devices, and adding and removing devices from the network. Its proposed architecture, however, is limited to Zigbee communication. Additionally, the Home Manager does not store device data and does not provide applications such as network anomaly detection that could perform analysis of device

behavior or communication patterns.

All the above tools mentioned are limited to their "out-of-the-box" functionality. We found only one tool designed to allow integration of other research applications for future development. Simpson et al. propose a central security manager that is built on top of the smart home's hub or gateway router [40]. The manager is aware of the status of all devices in the home and of reported vulnerabilities. The authors propose that additional modules can be built atop this manager to offer installation of software updates, filter traffic, and strengthen authentication for devices. The manager, however, can only observe communication to and from devices on the home network and does not capture traffic that does not go through the router. Specifically, it does not have helper devices (similar to the cubelets in the SPLICEcube system) that can assist with network traffic capture.

Chapter 4

Architecture

In this chapter we first provide a conceptual view of the proposed architecture that describes how the various high-level components of the SPLICEcube system fit together. We follow with a detailed description of each component.

Section 4.1

Conceptual Architecture

The SPLICEcube system deploys within a smart home as an assembly of three components: a set of cubelets, the cube, and the database, as shown in Figure 4.1. Each cubelet captures Wi-Fi traffic, parses the traffic for relevant features, and sends the extracted features to the cube. In addition to acting as the home's internet router, the cube receives the parsed network information from the cubelets and communicates with the database to store the data as defined by the database schema. The database holds the network data and can be queried by the cube if an application requests data.

In the following sections of this chapter, we identify the requirements of each component and explain how each component supports the scalability and extensibility architecture attributes outlined in Section 2.3.

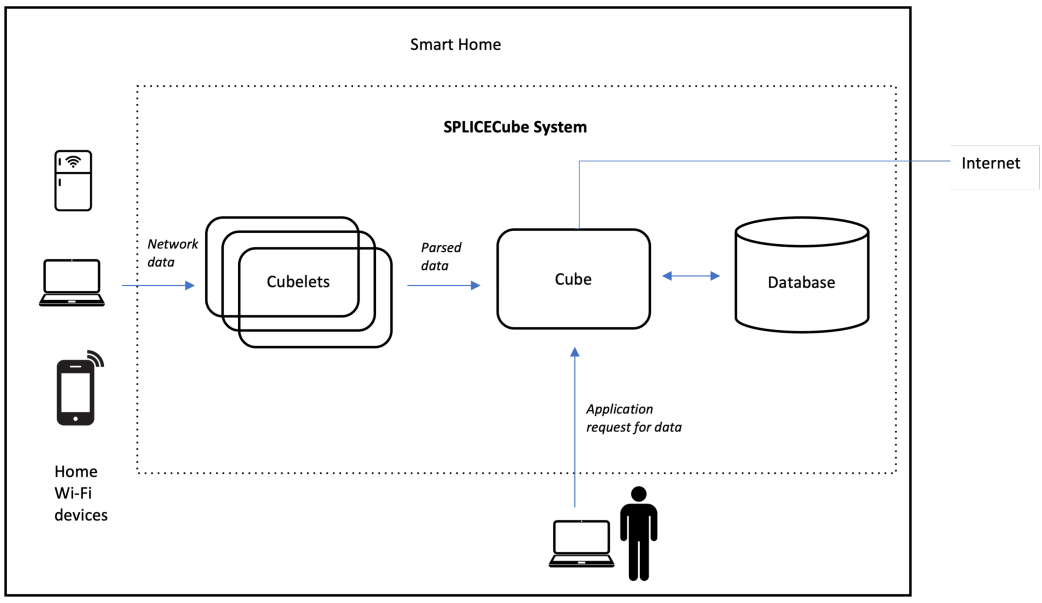

Figure 4.1: **Conceptual Architecture**. The SPLICEcube architecture consists of a set of cubelets, the cube, and the database.

Section 4.2

Cubelets

As shown in Figure 4.2, each cubelet consists of three modules: *capturer*, *parser*, and *transmitter*.

4.2.1. Capturer Module

A primary function of the cubelets is to detect devices that are inside or close to the home. One of the ways to detect devices is to capture wireless traffic with a wireless network sniffer. To accomplish this task, the cubelet must contain an interface that is capable of sniffing all Wi-Fi packets over-the-air. Although the interface will likely only capture packets on one channel (unless channel hopping is implemented), we envision the interface to be dual-band, i.e., capable of sniffing on both the 2.4 GHz and 5 GHz Wi-Fi bands, so that the SPLICEcube developer can choose which channel(s) to sniff.

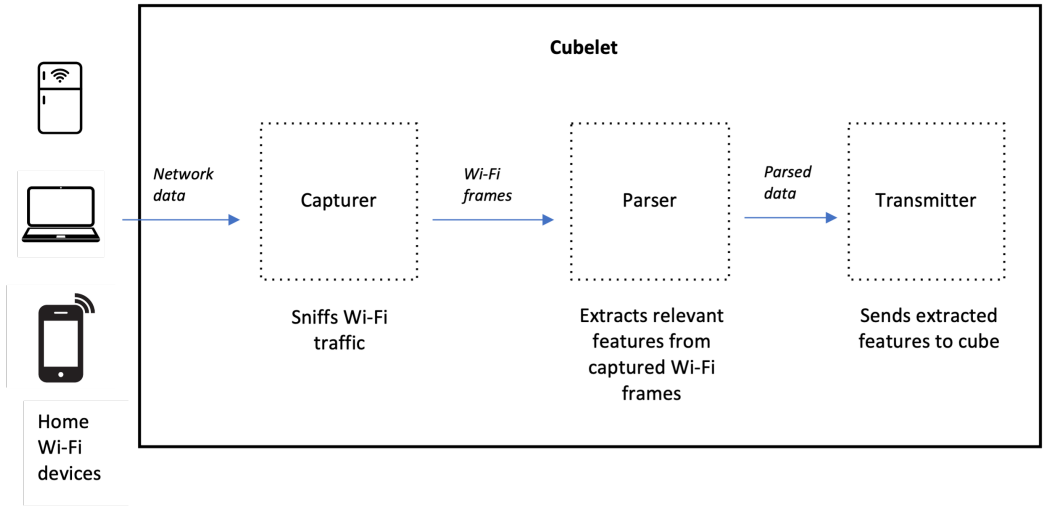

Figure 4.2: **Cubelet Architecture**. Each cubelet consists of the capturer, parser, and transmitter modules.

4.2.2. Parser Module

A Wi-Fi frame contains a large amount of data, some of which may be unnecessary to retain. The parser module parses each captured frame and extracts relevant fields.

4.2.3. Transmitter Module

The next step after the frame parsing and extraction is to send the collected information to the cube for storage in a database. To support this communication, a cubelet must contain a network interface that can send data over a network. This may be a second Wi-Fi interface, or Ethernet, for example.

4.2.4. Support of Architecture Attributes

To facilitate scalability, we envision the cubelet's modules to run concurrently. This structure permits efficient processing of network data even in busy network conditions or with growing number of devices. For example, even if there is high network activity and the parser module is busy parsing frames, the transmission module can execute in parallel and will not be indefinitely blocked by the parser. To further promote scalability, we envision two separate network interfaces, one for the capture and

another for the transmission, to avoid time-sharing the network hardware. Although the capture and the transmission could theoretically occur on the same interface, separating them will allow the system to scale responsively by not missing packets during capture, especially during high network activity.

To facilitate extensibility, we divide and encapsulate the cubelet tasks into discrete modules. This division allows the SPLICEcube developer to undertake specific modifications to modules without revising the entire architecture. For example, if the SPLICEcube developer chooses to modify the module used to capture Wi-Fi packets, they can do so without manipulating other modules. To further support extensibility, additional modules can be added as functionality grows. For example, we can imagine adding a module to the cubelet that captures Bluetooth traffic, parses the data, and transmits to the cube, without changing the overall SPLICEcube architecture.

Section 4.3

Cube

As shown in Figure 4.3, the cube consists of the following modules: *router*, *reception*, *storage*, *disposal*, and *integration*.

4.3.1. Router Module

The cube acts as the home router and provides a Wi-Fi network for clients to join. Currently, this is the only function of the router module in the proposed architecture, but we discuss additional potential capabilities in Section 7.2.1. We envision each cubelet and each home Wi-Fi device to be connected to the cube's network. The Wi-Fi network must be protected by a strong security protocol to protect the residents' information on the network. Appropriate software is required to support routing.

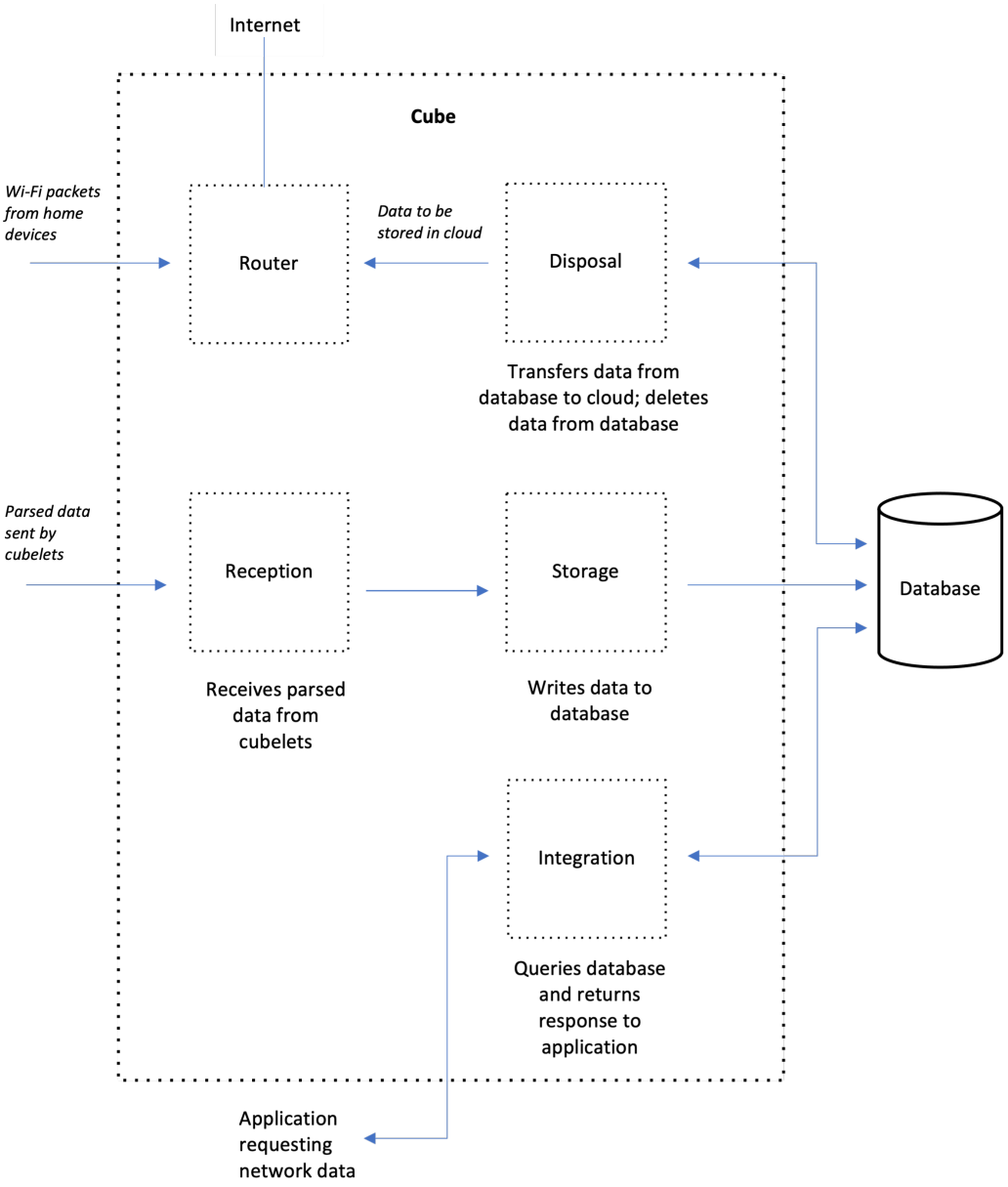

Figure 4.3: **Cube Architecture**. The cube consists of the router, reception, storage, disposal, and integration modules.

4.3.2. Reception Module

The cube receives parsed data from the cubelets. As mentioned, we envision this cubelet-cube communication to occur over a network, so the cube must contain a network interface to receive transmissions from the cubelets.

4.3.3. Storage Module

Once the cube receives a cubelet transmission, it must store the information for future access. To store the data, the storage module communicates with a database that holds all received network information. We discuss the database in further detail in Section 4.4.

4.3.4. Disposal Module

Since storage on any device is finite, there must be a way to manage the database size and archive old data. We envision the disposal module to periodically upload some data from the database to cloud storage because storing data in the cloud is inexpensive. The frequency of upload depends on the variables below.

- Device storage: If the device that hosts the database has more available storage, then the database can store more data, and data can be migrated to the cloud less frequently.
- Network conditions: If there is high network activity, then the database will fill up more rapidly and data will need to be uploaded to the cloud more frequently.
- Application requirements: Certain applications may require data to be stored locally for a certain amount of time, which will limit the upload frequency.

Once the data has been uploaded to the cloud, we envision that the disposal module deletes this data from the database. To minimize cloud storage costs, the disposal

module can also be extended to permanently delete some data from the cloud once a certain size threshold is exceeded.

4.3.5. Integration Module

As discussed in Section 1.2, one of the primary motivations of the SPLICEcube system is to serve as a platform for research development. The vision is to let researchers leverage the SPLICEcube's general-purpose architecture to easily test their security and privacy research applications in a real environment.

The integration module must have an API that applications can use to request data. The applications may be running internally on the cube or remotely. If an application is running remotely and requires access to data, we envision application-cube communication to occur over a network (similar to cubelet-cube communication).

4.3.6. Support of Architecture Attributes

We envision the cube's modules to run concurrently. A concurrent structure facilitates scalability since the modules can run in parallel. For example, if the cube's reception module is busy receiving a large amount of data, the integration module can still execute once the CPU switches threads, instead of waiting for the entire chunk of data to be received.

Similar to how the cubelet's modules can be extended, we encapsulate the cube's tasks into modules to promote extensibility, so that individual tasks can be modified easily. For example, the SPLICEcube developer can modify the thread underpinning the integration module to change the query parameters without impacting other modules. Additionally, the goal is that this architecture is extensible to accommodate other applications. Regardless of the research application, if the researcher needs to access device network activity, they can simply request the information from the cube. The cube can build the request into a query to extract the appropriate data from the

database and send this data back to researcher's application.

Section 4.4

Database

The database component is the central repository of parsed network activity. As mentioned in Section 4.3, the cube writes to the database (via the storage module) as it receives network data from the cubelets, and the cube also queries the database (via the integration module) if an application requests network or device information.

To live on a resource-constrained device in a smart home setting, we envision the database to be fast, self-contained and have a small memory footprint. Finally, the database should be well-suited to hold network data, which we envision to be structured with distinct fields.

4.4.1. Support of Architecture Attributes

The database will grow as network data is written to the database. To scale the database and support this growth, we envision the database to be housed on a device with enough flash memory to allow storage of the large quantity of network information. We describe the storage requirements in more detail in Section 6.4. Regardless of the amount of memory, the database will eventually grow to exceed the memory capacity. As described in Section 4.3.4, we envision a data disposal module to contain the database size and facilitate scalability.

The database is also extensible because the schema of the database can be modified to accommodate different applications. The schema model depends directly on the type of data that needs to be stored and that is required by an application. We envision that the SPLICEcube developer may adjust the schema and add more tables as appropriate.

Chapter 5

Implementation

In this section, we describe our implementation choices in building an initial prototype for the proposed architecture. We discuss the cubelets, cube, and database components and the modules within each component.

Section 5.1

Cubelets

In our implementation, there are three cubelets. Each cubelet is a Raspberry Pi 4 or Raspberry Pi 3 running Raspberry Pi OS. There is no specific reason why we use different models of Raspberry Pi; we simply used what was available and any Raspberry Pi model that satisfied the cubelet component architecture requirements identified in Section 4.2 would have sufficed.

As shown in Figure 5.1, each cubelet contains a dual-band *BrosTrend Wi-Fi adapter* [8] plugged into the Pi via USB to sniff Wi-Fi packets. Please see section Section 4.2.1 for details about the use of the Wi-Fi adapter. Each cubelet also contains an internal wireless interface (built into the Pi 3 and Pi 4) for transmitting data to the cube via Wi-Fi. This interface supports both the 2.4 GHz and 5 GHz Wi-Fi bands so it can associate to networks that operate on either frequency band.

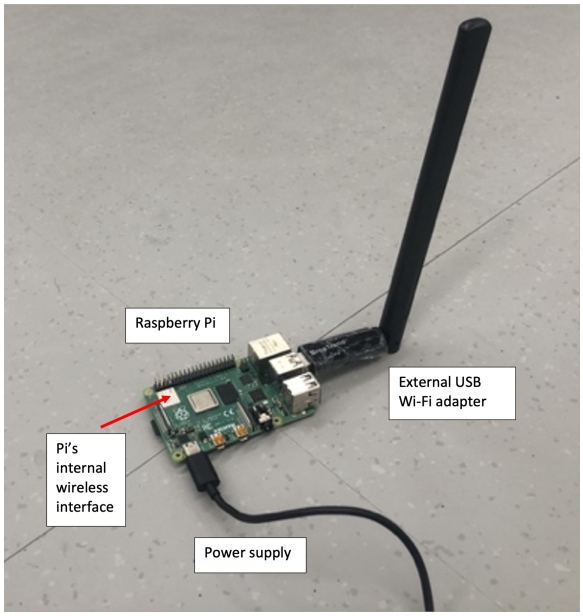

Figure 5.1: **Cubelet Prototype**. The cubelet prototype consists of a Raspberry Pi 3 or 4 and an external Wi-Fi adapter.

Each cubelet has three primary functions, which we separate into modules: Wi-Fi frame detection (capturer module), Wi-Fi frame parsing (parser module), and transmission to the cube (transmitter module). We create a separate thread for each of these modules so that they can run concurrently, as shown in Figure 5.2.

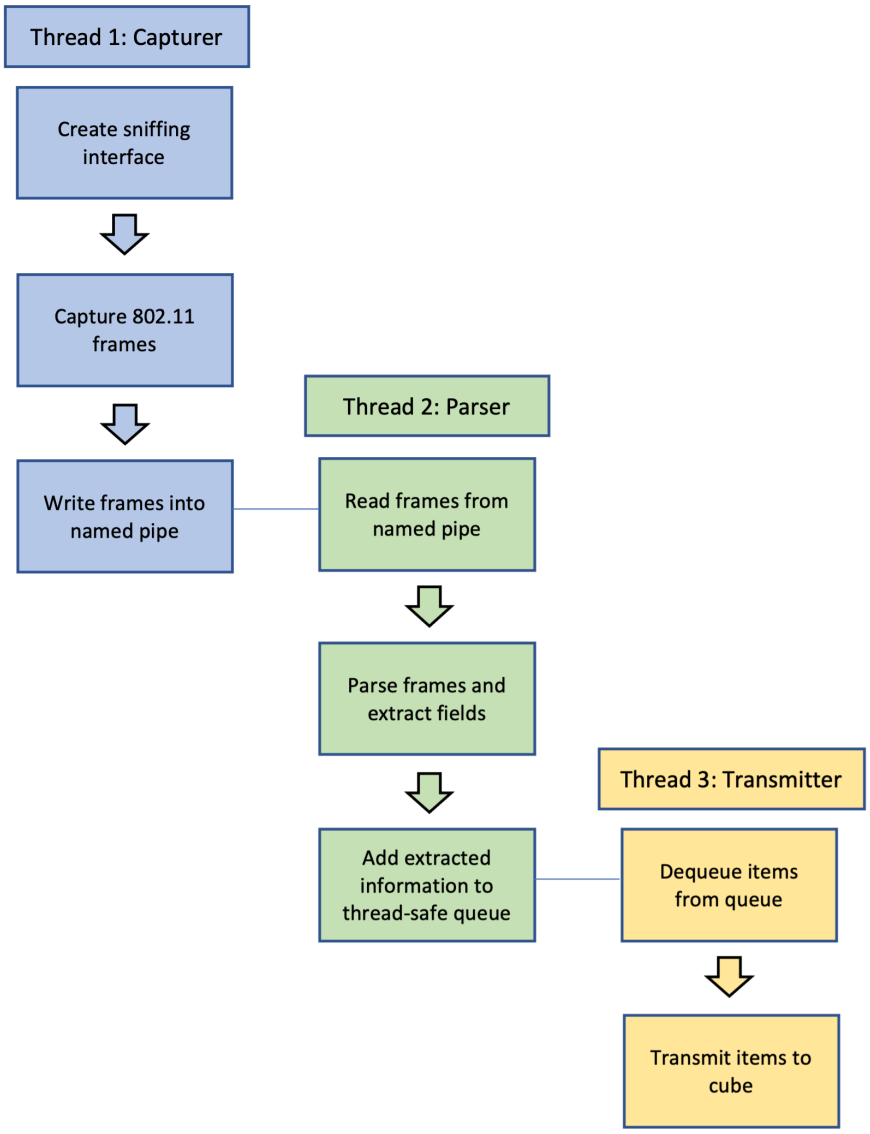

Figure 5.2: **Cubelet Threads**. The executing threads of each cubelet are the capturer thread, the parser thread, and the transmitter thread.

Over the course of the project, we revised the design of the cubelets functionality to make the implementation scalable and extensible. Below we describe and justify the design decisions for each module.

5.1.1. Thread 1: Capturer

Create sniffing interface. As mentioned, we use an external dual-band Wi-Fi adapter to sniff Wi-Fi packets in monitor mode. The built-in Wi-Fi interface on the Raspberry Pi is only capable of capturing packets in *managed mode*. Sniffing in managed mode means that an interface can only receive packets sent to/from the interface that is sniffing. In contrast, sniffing in *monitor mode* allows an interface to sniff all packets over the air that are within range even if the packets are not from, or addressed to, the interface. Since we would like to capture information from all devices within range, monitor mode is the appropriate mode for sniffing.

To capture network traffic with the Raspberry Pi cubelet, we first set the mode of the Wi-Fi adapter to monitor mode and then use *tcpdump* [45], a packet capture program, to continuously capture Wi-Fi packets.

Capture 802.11 frames. The sniffer operates at the Medium Access Control (MAC) sublayer of the Data Link Layer (Layer 2) within the OSI model, as shown in Figure 5.3. This means that the type of data the sniffer captures are 802.11 MAC frames.

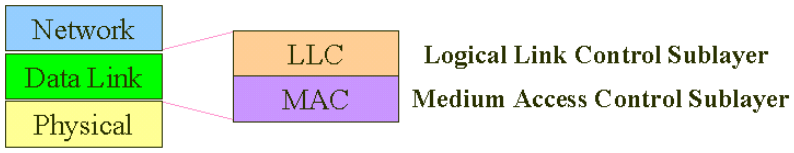

Figure 5.3: **Data Link Layer**. The Data Link Layer consists of the LLC and MAC sublayers [16].

As shown in Figure 5.4, each 802.11 MAC frame contains a MAC header, a frame body, and a frame check sequence (FCS).

The MAC header is unencrypted; the frame body encapsulates all information from higher layers (LLC sublayer, Network layer, Transport layer, Application Layer

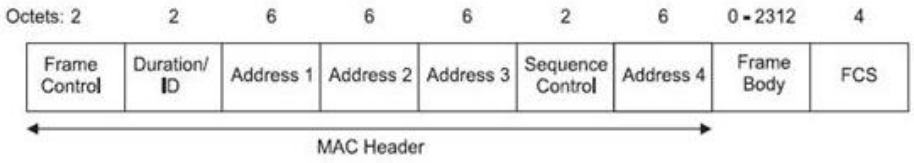

Figure 5.4: **802.11 MAC Frame Format**. A 802.11 MAC frame consists of the MAC header, the frame body, and the frame check sequence [27].

etc.) and is typically encrypted; the FCS is an error-detecting code and is unencrypted. Additionally, the Wi-Fi adapter that we use to perform the frame capture adds an unencrypted pseudo-header called "RadioTap" to each frame it captures. The RadioTap [35] header is not part of the 802.11 frame structure, but it is a common mechanism for drivers to supply additional information about received frames, such as the channel frequency and data rate.

Write frames into named pipe. Thread 1 writes each captured frame directly to a named pipe. By using a named pipe, Thread 1 can sniff and write to the pipe, while Thread 2 simultaneously reads the bytes from the pipe for further analysis (see Section 5.1.2).

Design Decisions for Capture Module.

(a) Sniffing tool: We initially used Scapy [37], a Python packet manipulation program, to sniff Wi-Fi packets. Scapy, however, is slow and misses packets when network traffic is high, so its lackluster (albeit convenient) performance necessitated the switch to tcpdump.

(b) Named pipe: Thread 1 can write to the named pipe as Thread 2 reads from it. A queue would also have allowed this FIFO functionality, but we choose to use a named pipe because it is easy to redirect the output of *tcpdump* to a named pipe. Please see Section 5.1.2 for additional justification of our decision choice.

5.1.2. Thread 2: Parser

Read frames from named pipe. As Thread 1 sniffs frames and writes them to the named pipe, Thread 2 reads each frame from the named pipe using a Python library called *dpkt* [11]. Thread 2 utilizes the *dpkt.pcap.Reader* function to iterate through each frame in the pipe and perform the parsing.

Parse frames and extract fields. For each frame, Thread 2 parses the RadioTap header and extracts the following fields: received signal strength indicator (RSSI), channel frequency, and data rate. Then the thread parses the MAC header of the 802.11 frame and extracts the following fields: source MAC address, destination MAC address, frame type, frame subtype, fragment number, and sequence number. Figure 5.5 shows the fields of a MAC header.

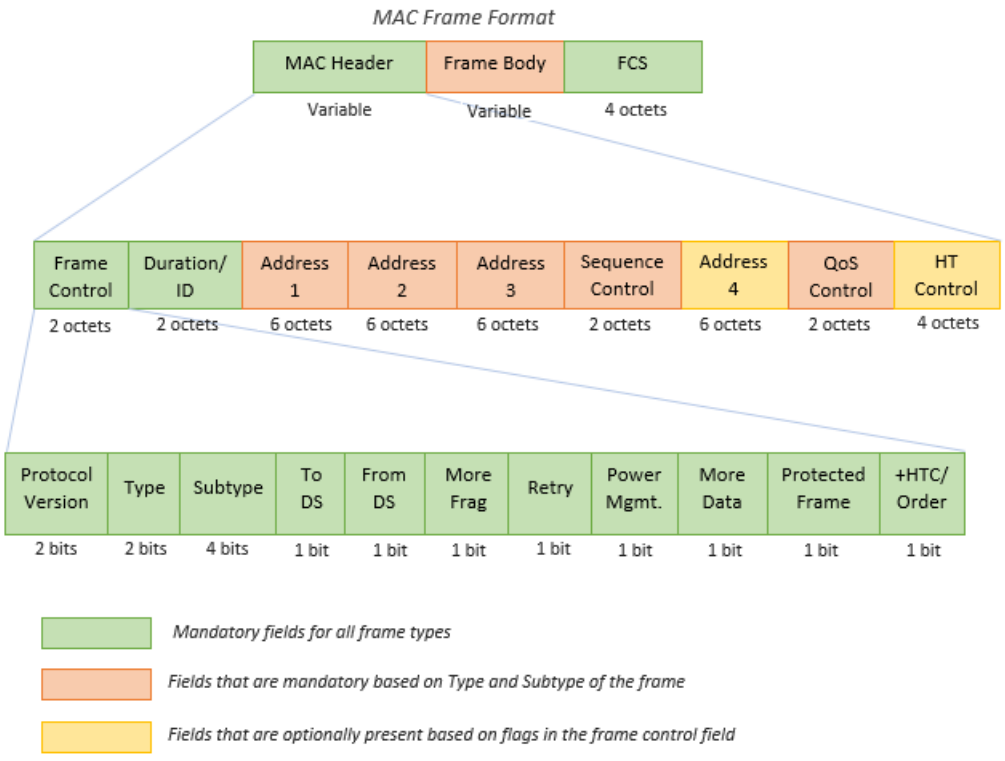

Figure 5.5: **MAC Header Format**. The MAC header is unencrypted and contains several fields describing the frame information [28].

Thread 2 adds each extracted field, as a text string, to a Python list, and then appends a delimiter character (|) to signify the end of the frame for future parsing. Below is an example of a parsed frame. The *timestamp* field is not part of the extracted information, but it represents the time at which the *dpkt.pcap.Reader* function reads the frame.

```
# src mac, dst mac, timestamp, pkt type, pkt subtype, fragment,
sqn num, rssi, channel freq, rate, delimiter
[60f81da9236a,80cc9c2c84e8,2022.04.14.16.29.58.838484,0,8,15,4192,-45,
5240,48,|]
```

The fields we choose to extract only represent part of a Wi-Fi frame's full information. For example, we extract the information required for the Inside-Outside application (RSSI, timestamp, source MAC address, destination MAC address) and several other fields that provide basic information about the frame. There are, however, many more fields contained within the unencrypted RadioTap and MAC headers, and it is possible for the SPLICEcube developer to extend this work and extract more information as necessary using the dpkt library.

As mentioned previously, the frame body of the 802.11 frame is typically encrypted, since the Wi-Fi network being used by the transmitting device is usually protected with a password. The frame body encapsulates all the packet information from higher layers, so it contains a lot of useful information about a device's network behavior (source and destination IP addresses, source and destination ports, etc.). However, unlike the RadioTap header and MAC header, parsing the frame body requires the data to be decrypted. This decryption process is left to future development, but we give several suggestions about how to approach this problem in Section 7.1.2.

Add extracted information to thread-safe queue. Thread 2 appends the list to a thread-safe queue. So, each item in the queue is a list containing the contents (as a text string) of a single parsed and extracted Wi-Fi frame. Thread 3 then reads this FIFO queue and collects the queue items to transmit to the cube (see Section 5.1.3).

Design Decisions for Parser Module.

(a) Packet reading: We experimented with different tools such as PyShark [34], Pcapreader [38], and rdpcap [38], but these were too slow in parsing the vast volume of data being captured in monitor mode. dpkt proved to be the fastest parsing tool; speed in parsing is important to avoid bottlenecks that may occur during high network activity. dpkt's speed ensures that the architecture can scale appropriately even with a large number of devices in the home. dpkt is also robust enough to fully dissect a Wi-Fi frame since it has parsing support for all packet layers. This extensibility is important because the SPLICEcube developer should be able to extract more bits of data from a frame, if required by a future research application that is being integrated into SPLICEcube.

(b) Named pipe vs. queue: Thread 1 writes sniffed frames into a named pipe, but Thread 2 writes the extracted frame information into a thread-safe queue. Named pipes and queues are both FIFO and provide an avenue for transfer of data between threads, so one might question the choice of one over the other. A named pipe is accessed as a regular file. The dpkt reader (which reads frames from the named pipe) requires a file as input, so it makes sense to use a named pipe. The extracted frame information could have been written into a named pipe; using a queue was simply an easy choice for an initial implementation.

5.1.3. Thread 3: Transmitter

Dequeue items from queue. As Thread 2 adds the extracted fields to the thread-safe queue, Thread 3 periodically dequeues items from the queue.

Transmit items to queue. Thread 3 transmits to the cube via Wi-Fi. There are two conditions under which Thread 3 transmits the current contents of the queue by dequeuing items. The first condition is if the queue reaches a "maximum size", which we define as a parameter and which can be modified by the SPLICEcube developer. The second condition is if the queue has not reached the maximum size but a certain amount of time, which we also define as a parameter and which can be modified by the SPLICEcube developer, has elapsed since the cubelet transmitted to the cube. One can imagine a scenario in which there is sparse network activity so the cubelets are not capturing many frames and subsequently, each cubelet's queue is being populated slowly. If a cubelet has not communicated with the cube in the amount of time set by the parameter, then the cubelet transmits the current contents of the queue to the cube.

When one of the above conditions is met, each item in the queue, which is a list containing the extracted information for a captured Wi-Fi frame, gets dequeued and the contents of each list are concatenated together to form a long string. Below is an example of two items in the queue and the "long string" of concatenated data after both items get dequeued.

```
% queue containing two items
[[60f81da9236a,80cc9c2c84e8,2022.04.14.16.29.58.838484,0,8,15,4192,-45,
5240,48,|],[80cc9c2c84e8,18b43060d4a0,2022.04.14.16.29.58.841057,2,0,15,
24672,-53,5240,12,|]]
```

```
% string to be sent to cube (delimiter character separates each frame)
60f81da9236a,80cc9c2c84e8,2022.04.14.16.29.58.838484,0,8,15,4192,-45,
5240,48,|80cc9c2c84e8,18b43060d4a0,2022.04.14.16.29.58.841057,2,0,15,
24672,-53,5240,12,|
```

Then this string is encoded in a UTF-8 format and sent to the cube as a UDP datagram. The maximum size of a single UDP datagram is 65,535 bytes (8 byte UDP header + 65,527 bytes of data). And the actual limit for the data length, which is imposed by the underlying IP protocol, is 65,507 bytes (65,535 bytes - 8 byte UDP header - 20 byte IP header). This means that if the length of the data exceeds this maximum size, the data would get sent in chunks of 65,507 bytes. Since the UDP protocol is "unreliable" and does not guarantee delivery, it is possible that a datagram could be lost during transmission. To avoid losing 65,507 bytes worth of network activity, we opt to instead divide the data into smaller datagrams. The current datagram size in the program is set as a constant of 3000 bytes, but this can easily be modified by the SPLICEcube developer. Having smaller datagrams increases the overhead by increasing the number of transmissions but it minimizes the loss of data in the event of UDP packet loss.

Further, we ensure that we send only as many complete frames that can fit in a single datagram, given the datagram size set by the SPLICEcube developer. This approach ensures that a parsed frame is not split across two transmissions. To avoid IP layer fragmentation of the datagram, the SPLICEcube developer could limit the datagram size to 1 MTU.

Design Decisions for Transmission Module.

(a) Wi-Fi transmission: We choose to use Wi-Fi for cubelet-cube communication over a network because it enables wireless communication. Alternatives such

as Ethernet or powerline network could be used, but would require cables and are not ideal for a smart home environment, especially if the resident wishes to move cubelets to another location in the home.

(b) Transport layer protocol: Before UDP, we initially implemented the Message Queuing Telemetry Transport (MQTT) protocol [29], a messaging protocol based on a publisher-subscriber model. MQTT is lightweight, efficient, and is built for small, resource-constrained clients. It can scale to connect with millions of IoT devices, it is built on top of the TCP/IP stack and has reliable message delivery, and it enables security by allowing encryption of messages and authentication of clients. In our implementation, the cubelets acted as publishers and published their parsed frames. The cube's router module acted as the broker and forwarded the published data to the database, which acted as the subscriber.

Although the MQTT protocol is convenient and provides certain advantages, it is primarily used by IoT devices to transmit relatively small-sized messages (a few hundred bytes) and is not designed for the large volume of network data that a cubelet must transmit to the cube. In fact, we were unable to send data of more than a few thousand bytes via MQTT.

Additionally, some of MQTT's features are unnecessary for our purpose. MQTT relies on the TCP protocol for data transmission. TCP's reliable message delivery is unnecessary for cubelet-cube communication. Most devices emit packets frequently and the cubelets are continuously monitoring network activity. A dropped transmission from a cubelet to the cube is not necessarily consequential. Acknowledgement or retransmission of the data is typically unnecessary and wasteful of bandwidth.

On the other hand, UDP does not provide reliable delivery, does not encrypt

messages, does not guarantee in-order sequence of delivery, and has a small header size. As a low-overhead protocol, UDP is more suitable for cubelet-cube communication.

(c) Encryption: In our implementation, the cubelet transmissions occur over the cube's Wi-Fi network, which is protected by Wi-Fi Protected Access Version 2 (WPA2), a strong wireless security protocol [48] that provides confidentiality, integrity, and authenticity. Hence, the extracted information of each sniffed Wi-Fi frame, which becomes the frame body in a cubelet-cube transmission, is already encrypted via WPA2 and does not require additional encryption. An adversary would need to 1) know the password of the cube's Wi-Fi network and 2) capture the handshake between the cubelet and the cube during the cubelet's initial association with the network, to be able to decrypt the cubelet transmissions.

Section 5.2

Cube

In our prototype, we implement the cube's router module on one device and the other three cube modules (reception, storage, disposal, and integration) on another device. As shown in Figure 5.6, both devices together make up the cube, but for clarity, we henceforth refer to the device hosting the router module as the *router* device and the device hosting the other four cube modules as the *collector* device.

The collector device is a Raspberry Pi 3B+ running Raspberry Pi OS. The collector contains the reception module and its internal wireless interface (built into the Pi 3B+), is used to receive cubelet transmissions over Wi-Fi. This interface supports both the 2.4 GHz and 5 GHz Wi-Fi bands so it can associate to networks that operate on either frequency. The collector also hosts the database; we discuss the

implementation of this structure in more detail in Section 5.3.1. The router device is a Raspberry Pi 4 running OpenWrt, which is an open-source software primarily used on embedded devices to route network traffic [33]. The router's network is protected by WPA2, and the collector is connected to the router's network.

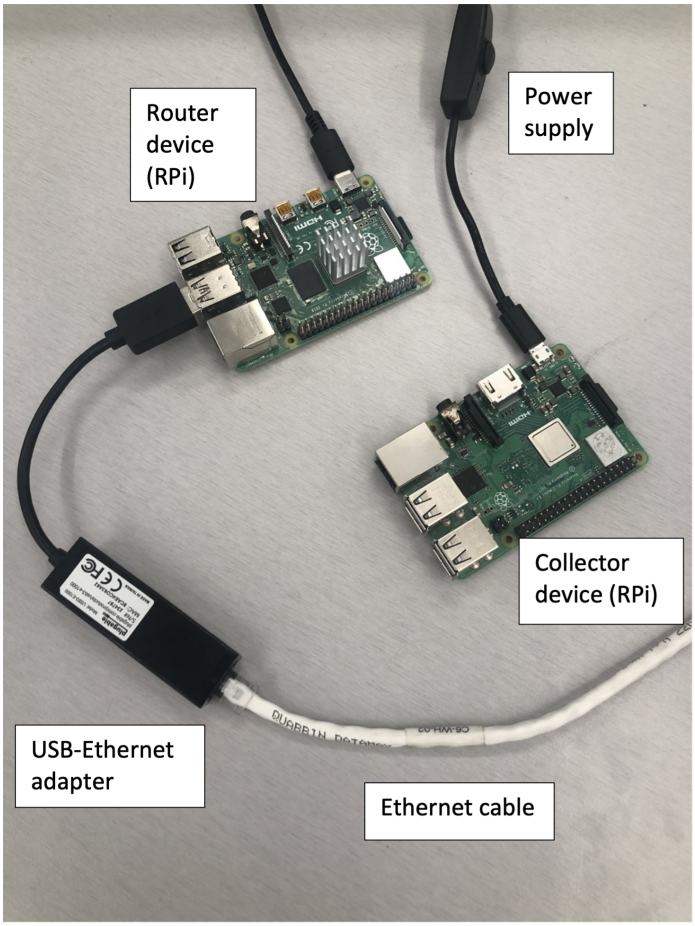

Figure 5.6: **Cube Prototype**. The cube prototype consists of the router device and the collector device. Each of these devices is a Raspberry Pi 3 or 4.

We use two different devices because OpenWrt is tailored specifically to provide router functionality, so it has limited storage and a limited package system that makes it difficult to install certain packages (Git, SQLite3, etc.). On the other hand, Raspberry Pi OS is a robust distribution and offers many packages for easy development. There is no specific reason why we use different Pi models (Pi 3B+ vs. Pi 4) for the

two devices. We simply used what was available, and any Raspberry Pi model that supports the cube component architecture requirements would have sufficed.

In a real deployment, we could imagine two separate pieces of hardware for the router and the collector, but perhaps contained within the same physical enclosure. So, the user would just see one box (the cube), but inside there would be two motherboards to accommodate the separate functions.

The router device requires an Internet connection to be functional. This means that the WAN interface (USB port) of the Raspberry Pi router must be connected to an Ethernet port to provide an Internet connection. There are several options to acquire an Internet connection via Ethernet – one can connect the Pi to an existing home router, to a wall/floor Ethernet port (which is, in turn, connected to a router), or to the Internet service provider (ISP) modem. The WAN interface then dynamically receives an IP address from the ISP. The LAN interface of the cube router is set to be on a separate subnet (we set a static IP address of 192.168.9.1), so that all Wi-Fi clients that connect to the cube's network receive an IP address on this subnet via OpenWrt's DHCP implementation. In a real deployment, we can imagine that the cube router would be the true home router. In our prototype, we create our own router with OpenWrt and deploy a separate Wi-Fi network instead of using an existing home router and its existing network, to avoid interference with existing infrastructure. Additionally, setting up a separate Wi-Fi network for the cube gives us more flexibility for testing since we can control what devices are connected to the cube's network at all times.

Besides acting as a router, the cube has four primary functions: receiving parsed data from cubelets (reception module), communicating with the database to store the parsed data (storage module), transferring data from the database to the cloud (disposal), and communicating with the database to retrieve necessary information

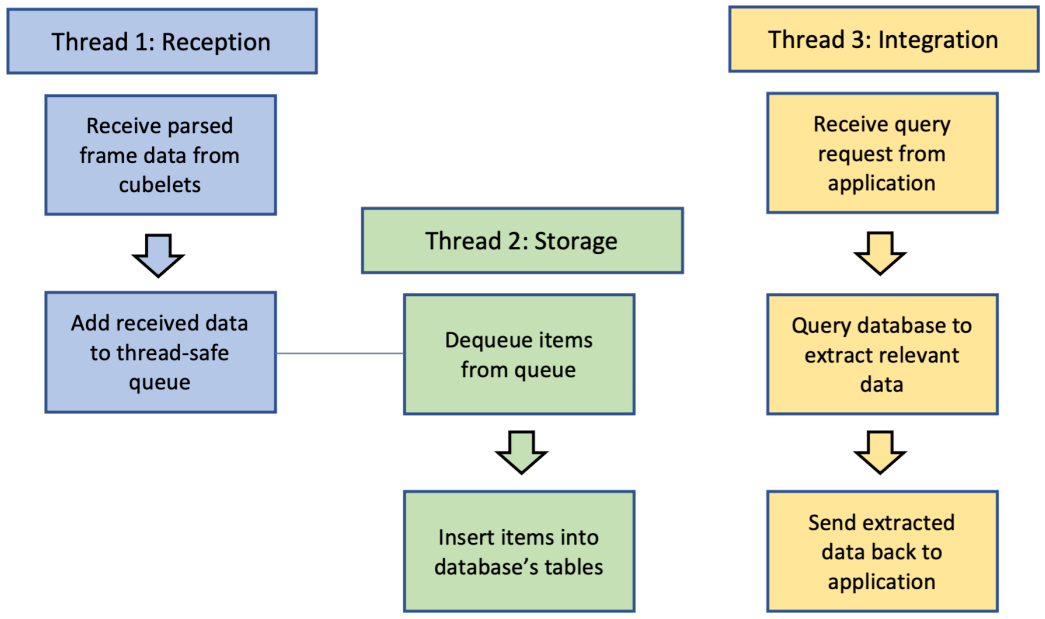

Figure 5.7: **Cube Threads**. The executing threads of the cube are the reception thread, the storage thread, and the integration thread.

for applications requesting network data (integration module).

We implemented three modules (reception, storage, and integration), and left the disposal module for future work. As shown in Figure 5.7, we create a separate thread for each of these three modules so that they can run concurrently. Note that these threads run on the collector device.

As with the cubelets, we iteratively revised the design of the cube's functionality to make the implementation scalable and extensible. Below we describe and justify the design decisions for each implemented module.

5.2.1. Thread 1: Reception

Receive parsed frame data from cubelets. We use a single thread (Thread 1) to receive data from all the cubelets via an open UDP socket. It is possible that a cubelet transmission arrives while Thread 1 is still in the process of receiving a previous transmission. The datagram of the more recent transmission will simply be placed in the OS receive buffer until the next call to '*recvfrom()*', at which point

Thread 1 will read the entire datagram from the buffer.

If the receive buffer on the socket is full when a datagram arrives, this datagram will be dropped.

Add received data to thread-safe queue. When Thread 1 receives a message from a cubelet, it appends the message and the IP address of the transmitting cubelet as a tuple to a thread-safe queue.

Design Choices for Reception Module.

(a) One thread for reception: As mentioned, it is possible that a datagram arrives while the receive buffer on the thread's socket is full, which would cause the datagram to be dropped. Implementing a separate thread to receive from each transmitting cubelet could minimize the dropped datagrams since each thread's receive buffer would fill up less frequently. We, however, choose not to implement multiple reception threads because a dropped datagram is not normally consequential. Since devices emit packets frequently and cubelets transmit to the cube frequently, the loss of a single datagram is not significant given the vast amount of network activity.

5.2.2. Thread 2: Storage

Dequeue items from queue. As Thread 1 receives datagrams and adds them to the queue, Thread 2 continuously dequeues items from the queue to store in the database. Each item is a string containing multiple frames' extracted Wi-Fi information. Below is an example of a queue item containing 3 frames.

```
% datagram containing 3 frames
3656eec73483,ffffffffffff,2022.04.14.16.45.35.331970,0,8,15,12304,-72,
5240,12,|4056eec73483,ffffffffffff,2022.04.14.16.45.35.332500,0,8,1,
```

```
28736,-73,5240,12,|
```

When Thread 2 removes an item from the queue, it parses the item to separate each frame. The thread also parses each frame to separate the different Wi-Fi fields.

Insert items into database's tables. A SQL statement inserts all the frames' information into the database according to the database schema. See Section 5.3.2 for details regarding the database insert operation.

Design Choices for Storage Module.

(a) Queue: A queue is FIFO, allows multiple thread access, and was an easy choice for an initial implementation. A named pipe could also have been used.

5.2.3. Thread 3: Integration

In our current implementation, we focus the integration module functionality on accommodating the Inside-Outside application. We chose the Inside-Outside application [14] as the first application to intgerate with the SPLICEcube system because we thought it would be a valuable addition to the SPLICEcube as a security research application. Additionally, this project is internal to our lab at Dartmouth College, so we had convenient access to the project code and resources.

Gralla's Inside-Outside application uses machine-learning algorithms to detect whether a target device is physically located inside or outside the house [14]. The system consists of three (or more) observers that act as Wi-Fi sniffers and a home hub that processes the collected data. The observers measure the received signal strength indicator (RSSI) for frames received from the target device. The observers send their observations to the home hub, which uses the RSSI data to train a classifier.

The SPLICEcube system is generally well-suited to integrate the Inside-Outside application. The cubelets act as observers, sniffing real Wi-Fi frames that have been

transmitted by home devices and extracting RSSI (among other fields) from the frames. The cubelets periodically transmit the extracted information to the cube's collector. The collector acts as the home hub, storing the received information in the database.

Let us now imagine that a researcher wants to run the Inside-Outside application on their machine to detect whether a particular target device is inside or outside the home at some given time. To accomplish this, the Inside-Out application must be able to communicate with the SPLICEcube remotely. The cube's integration module handles this communication, as described below.

Receive query request from application. The application establishes a TCP connection with the cube's collector device and sends a query consisting of some parameters. Thread 3 on the collector accepts the TCP connection and receives the application query. The Inside-Outside application requires the following parameters to classify a device as inside or outside the house: a list of MAC addresses of the target devices, a list of the cubelet/observer IP addresses to consider, the end time, and the start time. Below is a code excerpt showing the parameters sent by the Inside-Outside application.

```
mac_list = ["60f81da9236a"]
cubelet_list = ["192.168.9.130", "192.168.9.203", "192.168.9.219"]
end_time = datetime.utcnow().strftime("%Y.%m.%d.%H.%M.%S.%f")
start_time = (datetime.strptime(end_time,
"%Y.%m.%d.%H.%M.%S.%f") - timedelta(seconds=TIME_WINDOW)).strftime(
"%Y.%m.%d.%H.%M.%S.%f")
```

One challenge is to ensure that each frame that is emitted by the target device is associated with a fixed location. This association is necessary because it is important

to know which cubelets observed frames from a particular transmission location, and which cubelets missed those frames. Cubelets farther from the target's location will observe a low RSSI value or miss more frames emitted by the target when it is in that same location, due to attenuation of the Wi-Fi signal. Our Inside-Outside experimental setup relies on observed RSSI data from different transmission locations to train its machine-learning model.

One potential method to ensure that cubelets receive a frame associated with a fixed location is to keep track of the timestamps at which a frame is received by a cubelet. If multiple cubelets receive a frame at the exact same time, then it means that these cubelets received that frame from the same transmission location. This method, however, requires high granularity of the timestamps and precise clock synchronization among all the cubelets. Another potential method to align frames is to use a frame feature (or a combination of frame features) that uniquely distinguishes each frame. In the end, however, we decided that it is not necessary to verify that multiple cubelets see the same frame each time. If the time window of detection is small enough, then we can assume that the target device did not move in that window and that each frame received by the cubelets in that time window has been transmitted from the same location. The application currently requests a time window of 15 seconds, i.e., the time difference between the *start time* and the *end time* parameters is 15 seconds. In our experiment, we ensure that the target device is stationary in this time window. In future work, shorter time windows and non-stationary devices may be explored.

The researcher can provide any number of target devices and any number of cubelets as parameters to the Inside-Outside application. For example, the researcher may want to consider the impact that using only two observers has on the classification of the physical location of the target. So, the researcher would provide the IP

addresses of only two cubelets in their query.

Query database to extract relevant data. Thread 3 builds a database query based on the sent parameters and queries the database to extract the RSSI values from the appropriate frames stored in the database. See Section 5.3.2 for details regarding the database query operation.

It is possible that one or more cubelets observe fewer than 50 frames from the target in the specified time window. Thread 3 extracts from the database the maximum number of frames (up to 50) that any of the cubelets observed in the time window. If any of the other cubelets did not receive this same number of frames in the window, then Thread 3 fills in an RSSI value of -100 for the frames that they missed. A more negative value corresponds to a lower signal strength, so recording a value of -100 means that the cubelet missed some frames in the specified time window that were, however, picked up by another cubelet. Let us say that in a time window of 15 seconds, cubelet A observes 30 frames from target X, cubelet B observes 20 frames from target X, and cubelet C observes 10 frames from target X. The maximum number of frames observed by any of the three cubelets is 30. Since cubelets B and C missed some frames, Thread 3 records an RSSI value of -100 for the last 10 frames that cubelet B missed and for the last 20 frames that cubelet C missed. This padding means that the -100 is not necessarily aligned with the corresponding frames from other cubelets. Although we decided that it is not necessary to match the same frame across cubelets because we assume that the target device did not move in the given time window, future work may use frame features such as frame length or FCS as clues to match frames across cubelets.

Send extracted data back to application. After padding the missing frames with -100 as necessary, Thread 3 then sends back the list of RSSI values for each

cubelet to the Inside-Outside application. The application then populates a table such that each row of the file corresponds to one frame and each column corresponds to the measurements of one cubelet, as shown in Table 5.1.

ID	RSSI_1	RSSI_2	RSSI_3
1	-89	-45	-47
2	-100	-47	-44
3	-100	-51	-45
4	-100	-43	-46

Table 5.1: Merged Data Format in the Inside-Outside Application

The application then uses this data to train a machine-learning model that classifies the RSSI measurements.

Design Choices for Integration Module.

(a) Communication protocol: Contrary to the UDP connection used for cubelet-collector communication, we chose to use the TCP protocol for application-collector communication to ensure reliable delivery for the query request. In case the request gets dropped, retrying is important so that the application can obtain the information it needs.

Section 5.3

Database

In our implementation, the collector device hosts the database. We chose to use a *SQLite3* database, and we justify this design decision in Section 5.3.4.

5.3.1. Database Structure

In our implementation, there are currently four database tables that contain the parsed network data. We choose this table schema to reflect the four major entities

that the data contains. Below we list the four database tables and delineate the structure of the tables with an Entity Relationship Diagram (ERD).

(a) *devices* table: holds the MAC addresses of all seen devices and their corresponding manufacturers. "Seen devices" refers to devices whose emitted Wi-Fi frames have been captured by a cubelet. The device manufacturer is found by looking up the OUI ID (first six bytes of the MAC address) in a local text file that lists the headquarter location of common device manufacturers.

(b) *manufacturers* table: holds the name of all seen device manufacturers and their corresponding headquarters. The headquarters column is simply a placeholder template for other columns that may be implemented in the future. The headquarters is found by referencing a local text file that lists the headquarter locations of common device manufacturers.

(c) *cubelets* table: holds the IP addresses of cubelets and their locations within the home.

(d) *frames* table: holds the data transmitted by the cubelets (source device MAC address, destination device MAC addresses, timestamp, packet type, packet subtype, fragment number, sequence number, RSSI, channel frequency, and rate) and some additional fields (source device manufacturer, destination device manufacturer, and IP address of transmitting cubelet).

As shown in the ERD in Figure 5.8, each table has a primary key and the tables are linked to each other through foreign keys to enable SQL *join* statements. The current table design provides a template for future development. For example, the *manufacturers* table may not be particularly useful today, but there may be a future research application that groups network traffic by manufacturer to baseline device network activity based on the manufacturer. Further, the table design itself

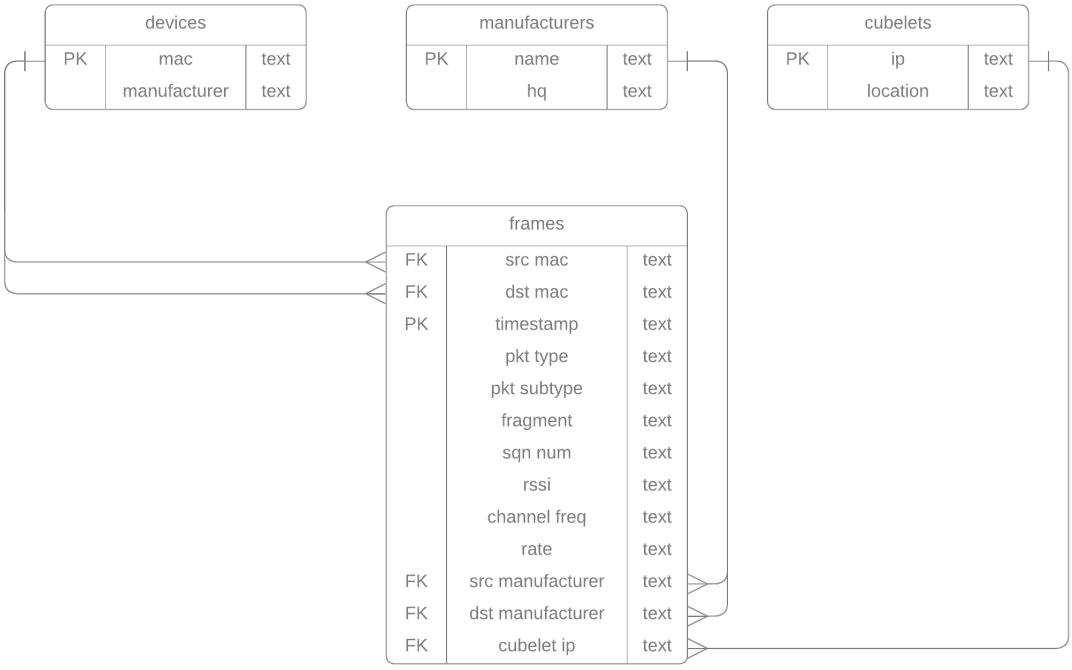

Figure 5.8: **Database Entity Relationship Diagram (ERD)**. The ERD consists of four tables: *devices*, *manufacturers*, *cubelets*, and *frames*.

is extensible to support the storage of additional data for a new network technology or a new research application. For example, if cubelet functionality is expanded to capture Bluetooth traffic, the SPLICEcube developer could add more database tables to store Bluetooth data.

The *frames* table contains an index on the *timestamp* field, so that it is easy to find the transmissions in a given time window. This is an integral part of the query used by the Inside-Outside application, so finding transmissions quickly based on timestamps makes query retrieval time much faster. As other research applications are added, more indices may be added to the tables accordingly.

5.3.2. Database Inserts

Section 5.2.2 describes how the storage module within the cube's collector device dissects a cubelet transmission to separate each frame in the datagram. A SQL *insert*

statement then writes the information in each frame into the different database tables according to the database schema.

Instead of individually inserting the information within each frame information as a row in a table, we implement the SQLite '*executemany()*' method, which expedites the process since it can add multiple rows in a single transaction. The SQLite '*INSERT OR IGNORE*' statement is used to discard an entry if the primary key already exists in the table. This prevents duplicate device MAC addresses in the *devices* table, duplicate manufacturer names in the *manufacturers* table, duplicate cubelet IP addresses in the *cubelets* table, and duplicate frames in the *frames* table. Below we depict the statements used for insertion. Each '?' symbol represents a field of the table row that is being populated.

```
self.cursor.executemany('INSERT OR IGNORE INTO devices
VALUES(?, ?);', devices_list)

self.cursor.executemany('INSERT OR IGNORE INTO manufacturers
VALUES(?, ?);', manu_list)

self.cursor.executemany('INSERT OR IGNORE INTO cubelets
VALUES(?, ?);', cubelets_list)

self.cursor.executemany('INSERT OR IGNORE INTO INTO frames
VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);', frames_list)
```

5.3.3. Database Queries

Section 5.2.3 describes how the integration module within the collector processes an application query request. A SQL *select* statement then queries the database to extract information from the frames that match the query criteria.

The example code below shows the query that extracts RSSI values for the Inside-Outside application, when supplied a target device MAC address, a cubelet IP address, and a time window.

```
self.cursor.execute("SELECT rssi FROM frames WHERE src_mac = ? AND
cubelet_ip = ? AND timestamp >= ? AND timestamp <= ?", (mac, cubelet,
start_time, end_time))
```

5.3.4. Design choices for database

(a) SQLite3: SQLite is a relational database, as opposed to a NoSQL database. A relational database is well-suited to store structured data [44] such as captured network data. A relational database also preserves the distinct relationships within the decomposed network data.

SQLite is a file-based database, as opposed to a database engine. In a database engine such as MySQL, there is a Relational Database Management Server (RDBMS) that sits between the clients and the database and manages file I/O, client connections, query optimization, query processing, and caching [20]. MySQL provides more functionality than SQLite such as authentication of users, support for more data types, and increased scalability. However, MySQL has a large memory footprint and contains a multitude of files [47]. In contrast, SQLite has a low-memory footprint and is an embedded database so it can live on its host device as a single file [47]. Since the SPLICEcube database may live on a resource-constrained device in a smart home, a fast and lightweight setup is more appropriate [43].

SQLite is also low overhead for the SPLICEcube developer since it is easy to install and requires minimal administration [47]. SQLite's simplicity makes it a convenient choice for an initial implementation of the architecture.

Finally, SQLite supports unlimited number of simultaneous readers, but it only allows one writer at any instant in time [5], which is well-suited for the SPLICEcube system. The collector's information storage module is the only entity that writes to the database. In our current implementation of the proposed architecture, there is only one thread within the collector's integration module that can read from the database, so only one application can request network data at a time. The SPLICEcube developer, however, could add more threads to the integration module to support multiple applications requesting data at the same time.

Chapter 6

Experimental Evaluation

Below we discuss two different experiments to evaluate our implementation.

Section 6.1

Experiment 1: Testing the System with the Inside-Outside Application

Section 5.2.3 describes how we incorporate the Inside-Outside application into our implementation of the SPLICEcube architecture. Below we discuss deploying this combined system in a real home and evaluate its performance.

6.1.1. Testing Procedure

Setup. The experiment was conducted in a residential house, as shown in Figure 6.1. None of the sides of the house were connected to any other buildings.

We used the following devices in this experiment:

(a) Cube, consisting of the router device and the collector device

(b) Three cubelets

(c) Target device (we used a Macbook Pro laptop)

Figure 6.1: **Experiment 1 Testing Site**. A residential house was used as a smart home environment.

(d) Device running Inside-Outside application (we used a laptop). We call this device the *application device*.

To setup the cube, we connected the cube router to the existing home router's Ethernet port and placed the collector adjacent to it. We placed the cube on the first floor in a central location of the house to maximize network coverage. We placed each of the three cubelets in a different corner of the first floor of the house and made sure they were connected to the cube's Wi-Fi network. Finally, we placed the application device inside the house and ensured it was connected to the cube's network, as shown in Figure 6.2.

The experiment consisted of three data collection runs. The first run collected data when the target was placed at 30 different positions *inside the house on the first floor*. The second run collected data when the target was placed at 15 different positions *inside the house on the second floor*. The third run collected data when the target was placed at 50 different positions *outside the house*. As shown in Figure 6.2 we created a

map of the transmission locations (the positions where the target device was placed) based on the home's floor plan. Each transmission location was about 1.5 m apart from each other. We also had the target device actively playing a few YouTube videos to ensure that it would generate some network traffic.

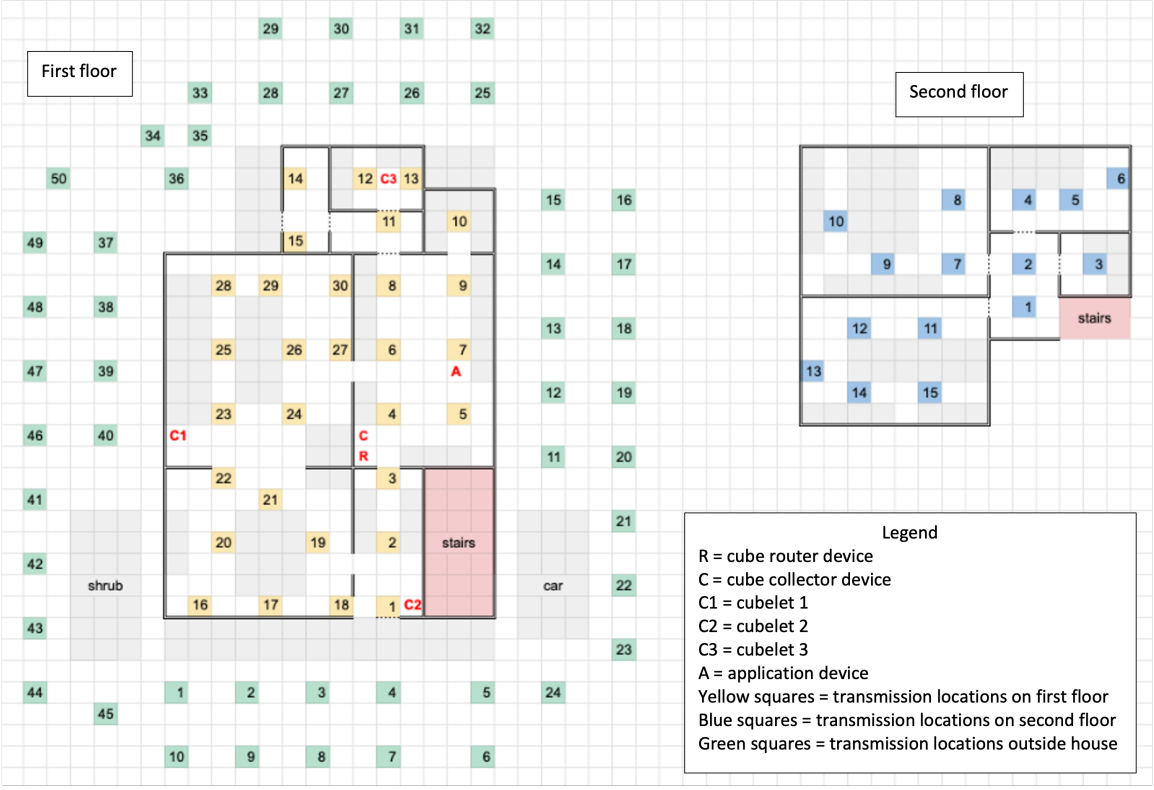

Figure 6.2: **Experiment 1 Testing Site Setup**. The house floor plan shows the location of the system components and the target's transmission locations. *R* represents the location of the cube's router device. *C* represents the location of the cube's collector device. *C1*, *C2*, and *C3* represent the locations of the cubelets. The yellow squares represent the transmissions locations on the first floor of the inside of the house. The blue squares represent the transmission locations on the second floor of the inside of the house. The green squares represent the transmission locations outside the house.

Execution.

(a) We began operating the system. We started with the first data collection run – the first floor of the house (denoted by the yellow squares in Figure 6.2).

(b) We placed the target in the first transmission location of the current data collection run.

(c) We waited for approximately 1 minute with the target in this transmission location. The waiting period was to ensure that 1) the target device had enough time to emit Wi-Fi frames so the cubelets can capture some network activity from the target in its current location and 2) the cubelets had enough time to transmit their recently captured information to the collector.

(d) We sent a query request specifying the necessary parameters (the target MAC address, the IP addresses of the three cubelets, and a 15 second time window) from the Inside-Outside application to the collector, as described in Section 5.2.3.

Since there were three cubelets in this experiment, the collector queried the database three times to extract the RSSI values for the frames observed by each of the three cubelets separately. For each query, the collector retrieved the RSSI values from the 50 (or the maximum number available) most recent frames that were observed by the specified cubelet and were transmitted by the target within the 15 seconds prior to the time of query. The collector then sent back the list of RSSI values for each cubelet to the application, and this data was populated into a .csv file on the application device, as described in Section 5.2.3.

(e) We then moved the target to the next transmission position and repeated steps c-d for all the transmission locations of this data collection run.

(f) We then repeated steps b-e for the next data collection run – the second floor of the house (denoted by the blue squares in Figure 6.2).

(g) We then repeated steps b-e for the third data collection run – outside the house (denoted by the green squares in Figure 6.2).

At the end of the three runs, the application device contained three .csv files, one for each data collection run. We used this data to train and test the Inside-Outside machine-learning model that classifies the RSSI measurements.

6.1.2. Results

Proof-of-concept. This experiment validates our prototype implementation and demonstrates a successful proof-of-concept use of the SPLICEcube architecture. The cubelets, cube, and database components worked in accordance with the proposed architecture in a real home environment.

ML model results. The Inside-Outside application uses three supervised machine-learning classifiers: Decision Tree (DT), K-Nearest Neighbors (KNN), and Random Forest (RF). Additionally, the application uses 5-fold cross-validation to evaluate each classifier. We do not detail the machine-learning classifiers and evaluation methods because they are specific to the application and out of scope of the contributions of this thesis. We simply use the model as a black box to demonstrate an end-to-end integration of the Inside-Outside application with our SPLICEcube implementation. Below we present some of the model's output metrics to illustrate the success of our system.

Table 6.1 shows each classifier's performance in predicting the target device's location when it was located inside the house. The Random Forest classifier performs the best across all the metrics for both data collection samples and shows promising results in classifying a device as inside or outside the home.

Location of Samples	Classifer	Accuracy	Precision	Recall	F1 Score
First floor (inside)	DT	0.913	0.897	0.867	0.881
	KNN	0.763	0.677	0.700	0.686
	RF	**0.925**	**0.875**	**0.933**	**0.903**
Second floor (inside)	DT	0.877	0.769	0.667	0.714
	KNN	0.862	0.688	0.733	0.710
	RF	**0.969**	**1.000**	**0.867**	**0.929**

Table 6.1: Inside-Outside ML model accuracy, precision, recall, and F1 scores

Query time. It is important that query time stays low as the database grows in size. If the execution time of a database query becomes a bottleneck, the application will receive delayed responses after sending a query request to the collector.

To measure this, we logged the query retrieval time and the corresponding database size for each request sent in the experiment. As mentioned, for each received query request, the collector queries the database three times to extract the RSSI values for the frames observed by each of the three cubelets separately. We sum up the three query times to obtain the total query time for a request. As shown in Figure 6.3, Figure 6.4, and Figure 6.5, there is no correlation between query time and the database size in any of the three data collection runs. The total query time remains under 1 second for every query request. We do not define a goal query time since different applications may require query responses within different time frames.

Because we placed an index on the *timestamp* field of the *frames* table, and each query extracts a constant number of RSSI values in a specified time window, query time should theoretically not increase significantly. The results of this experiment validate this theory and confirm that the index on the *timestamp* field of the *frames* table in the database is working correctly. We hypothesize that the spikes in query time, visible in Figure 6.3, Figure 6.4, and Figure 6.5, are the result of the database reorganizing the index due to the addition of new data, as it processes a query. We

test and validate this hypothesis in Section 6.3.

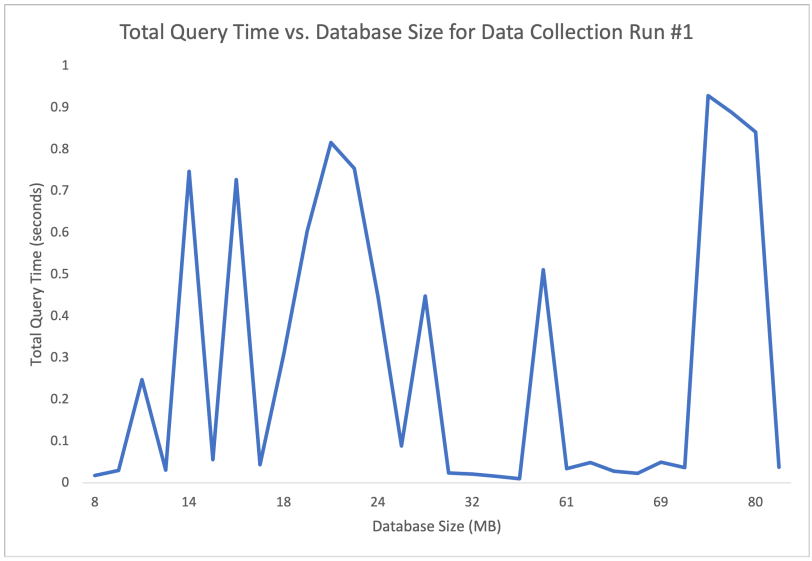

Figure 6.3: **Experiment 1 Query Time Result for Data Collection Run #1**. This graph shows total query time vs. database size for data collection run #1 (first floor of house).

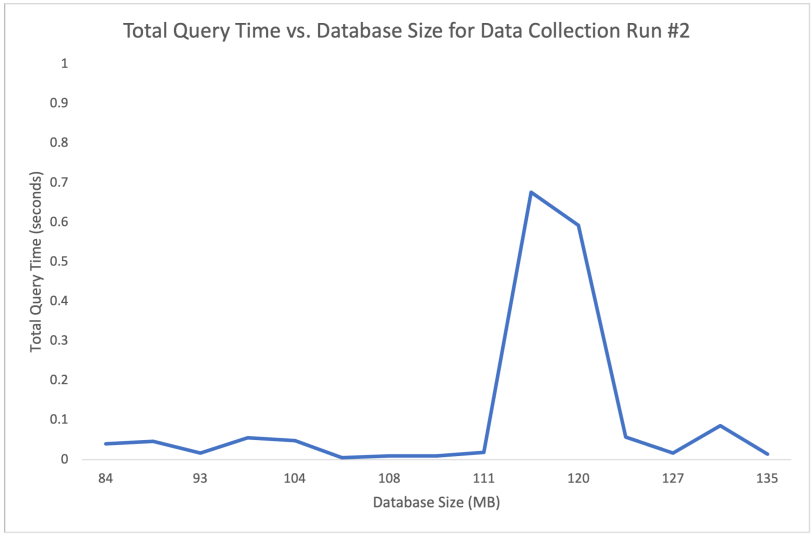

Figure 6.4: **Experiment 1 Query Time Result for Data Collection Run #2**. This graph shows total query time vs. database size for data collection run #2 (second floor of house).

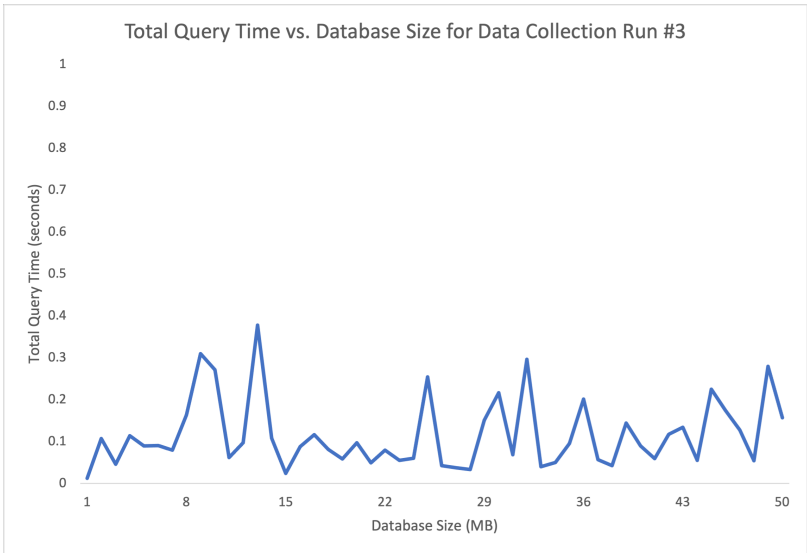

Figure 6.5: **Experiment 1 Query Time Result for Data Collection Run #3**. This graph shows total query time vs. database size for data collection run #3 (outside house).

Section 6.2

Experiment 2: Testing the System with Controlled Tests

To test our implementation more extensively and gather more system metrics, we also ran some controlled tests. These controlled tests were meant to let the system run for a long period of time and periodically observe certain metrics as the system collects and stores network data.

6.2.1. Testing Procedure

Setup. We used the following devices in this experiment:

(a) Cube, consisting of the router device and the collector device

(b) Three cubelets

(c) Device running Inside-Outside application (we used a laptop). We call this device the *application device*.

In contrast with Experiment 1 (Section 6.1), there was no specific target in this experiment.

We ran this experiment at two different sites. The first site was the same residential house used in Experiment 1. The second site was a lab in the Center for Engineering & Computer Science at Dartmouth College. Since a computer lab and a residential home are different environments and contain different number of devices, we chose to test both these locations to observe the difference in the database size and resulting insertion time and query time metrics as the system collected network data.

We followed the same system setup as described in Experiment 1 by ensuring that the cube router had Internet access, the cubelets were spread out across the testing site, and the cubelets and application device were connected to the cube's network.

Execution.

(a) We began operating the system so it was capturing, sending, storing, etc. We made sure that the database was empty before starting the experiment.

(b) We sent a query request specifying the necessary parameters (the target MAC address, the IP addresses of the three cubelets, and a 15 second time window) from the Inside-Outside application to the collector, as described in Section 5.2.3. We arbitrarily selected one device on the home Wi-Fi network and used its MAC address as the target MAC address parameter.

(c) We logged the database insertion time of all the frames in a single cubelet transmission (one datagram sent from a cubelet to the cube). We chose not to

log the insertion time for every cubelet transmission because it would add a lot more overhead to the system and could distort our results.

(d) We repeated steps b and c about every 30 minutes, for a total of 4 hours.

6.2.2. Results

Query time. Similar to Experiment 1, there was no correlation between query time and database size in both the home and lab tests (see Figure 6.6 and Figure 6.7). The database grew to about 1 GB in both tests. Apart from the initial spike in Figure 6.6, the query time stays relatively constant, confirming that the index on the *timestamp* field of the *frames* table is functioning correctly. The total query time represents the sum of the duration of the three queries that the collector undertakes for a single application query request. Once again, we hypothesize that the occasional spikes in query time, visible in Figure 6.6 and Figure 6.7, are the result of the database reorganizing the index due to the addition of new data, as it processes a query. We test and validate this hypothesis in Section 6.3.

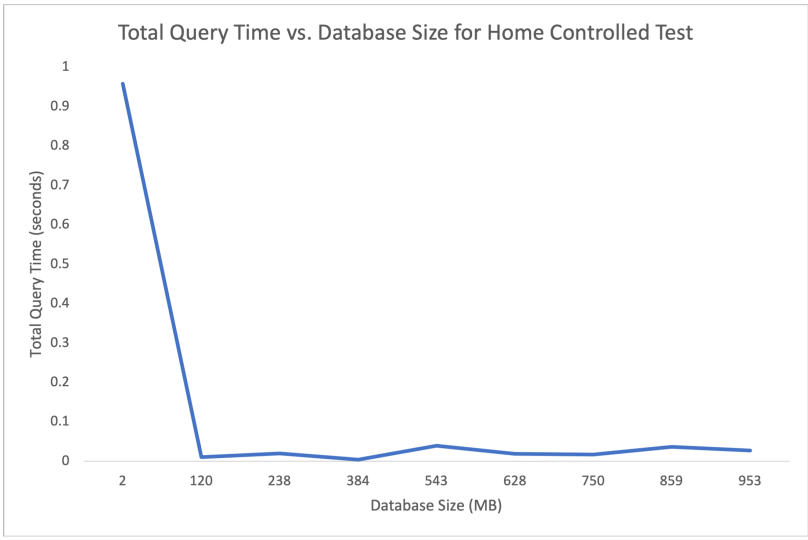

Figure 6.6: **Experiment 2 Query Time Result for the Home Site**. This graph shows the total query time vs. database size for the home testing site.

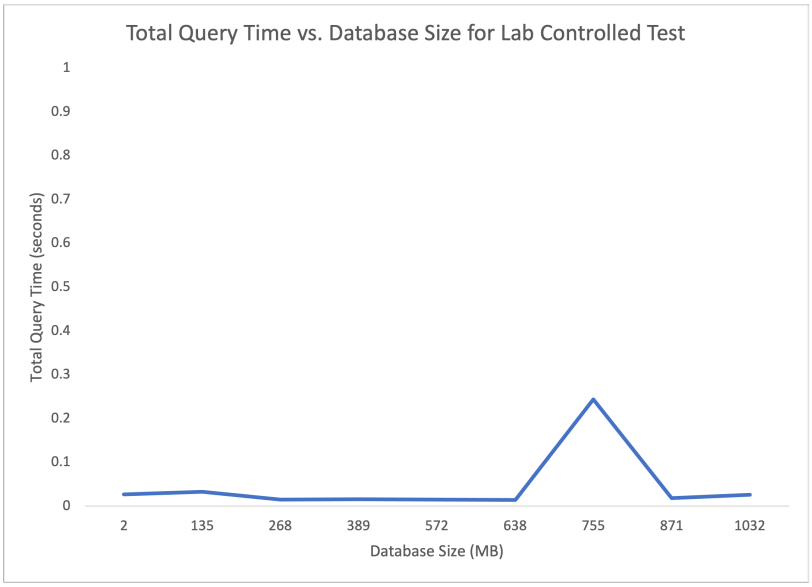

Figure 6.7: **Experiment 2 Query Time Result for the Lab Site**. This graph shows the total query time vs. database size for the lab testing site.

Insertion time. Figure 6.8 and Figure 6.9 show the insertion time for a single cubelet datagram at different points in the home experiment and lab experiment, respectively. The insertion times stay low (less than 0.015 seconds) and relatively constant even as the database grows in size. The amount of data being inserted into the database each time was about 3000 bytes since we set the size of a cubelet-cube transmission datagram to be 3000 bytes, as described in Section 5.1.3.

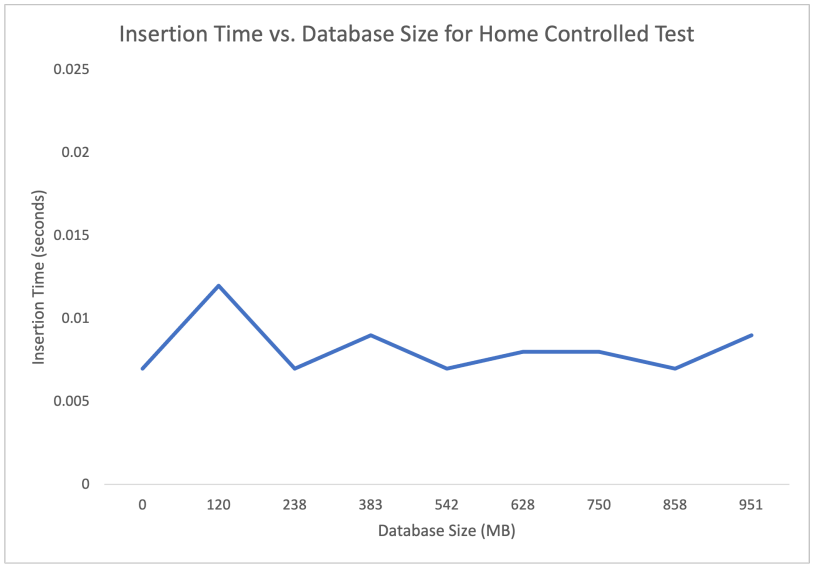

Figure 6.8: **Experiment 2 Insertion Time Result for the Home Site**. This graph shows the insertion time vs. database size for the home testing site.

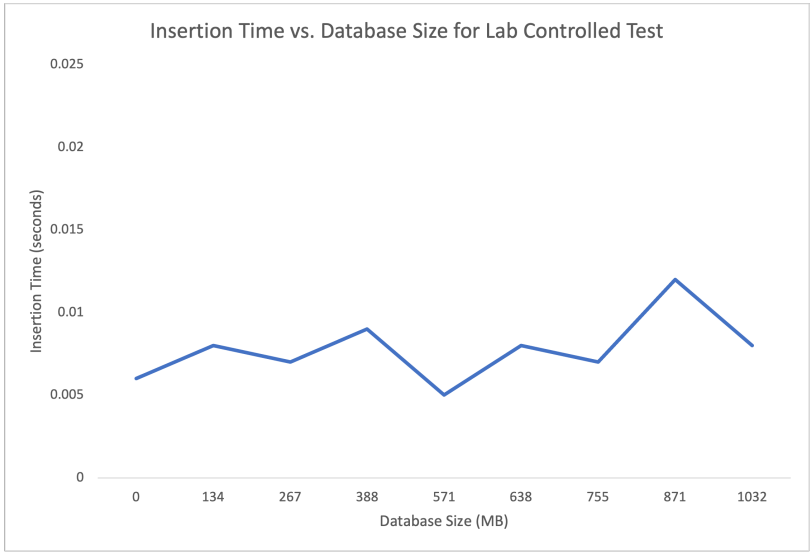

Figure 6.9: **Experiment 2 Insertion Time Result for the Lab Site**. This graph shows the insertion time vs. database size for the lab testing site.

Section 6.3

Experiment 3: Testing Query Time with a Constant Database Size

We hypothesize that the spikes in query time in Experiment 1 and Experiment 2 are because the database is reorganizing the index due to the addition of new data, as it processes a query. To test this hypothesis, we query the database while keeping the database size constant. In other words, the cubelets are not transmitting to the cube, and the cube is not adding any new data to the database. The database size is about 1 GB. We remotely query the database 20 times from an application device over the span of about 13 minutes, with varying amounts of time between each query.

We see in Figure 6.10 that the query times remain low and average about 0.055 seconds over the 20 trials. We can infer that there are no spikes in query time because there is no addition of data, so the database does not need to rearrange the index.

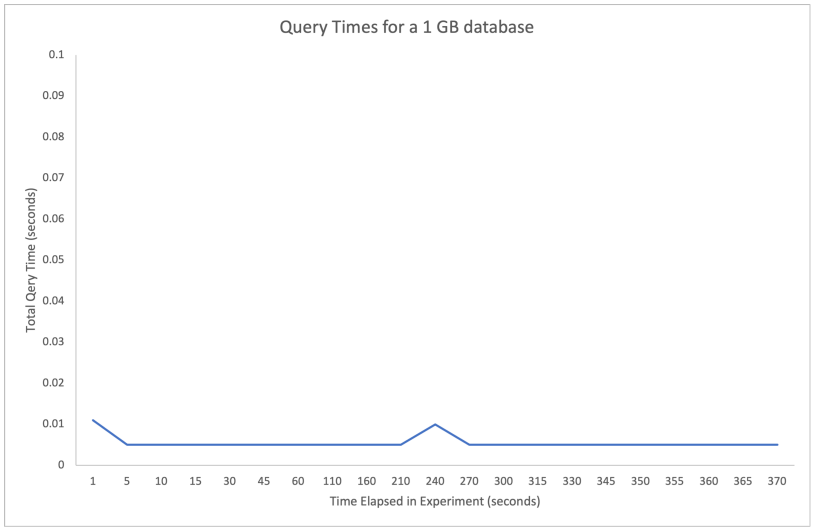

Figure 6.10: **Experiment 2 Query Time Result**. This graph shows the query times when the 1 GB database stays at a constant size.

Section 6.4

General Storage Requirements

Table 6.2 shows some final statistics of the database after 4 hours of recording data at each testing site.

Testing Site	Total Records	Total Size
Smart Home	4,737,225	953 MB
Computer Lab	5,161,737	1032 MB

Table 6.2: Database metrics after 4 hours of system operation

We see that for both testing sites, the database reached a size of about 1000 MB (1 GB) after four hours of operation. In this experiment, the database was housed on a Raspberry Pi with a 32 GB Samsung EVO Select microSD memory card [39]. If we assume that the network conditions were to remain constant and that the other programs on the Pi take up negligible memory, we can extrapolate and say that it would take about 128 hours (a little over five days) for the database to completely fill up the disk space. We envision that before reaching this storage threshold, the data disposal module would transfer older data to the cloud and subsequently delete this data from the database to free up space.

This experiment illustrates the general database storage requirements for two different network environments. Although the frequency of data uploads to the cloud depends on the network conditions, this experiment provides the SPLICEcube developer insight to better manage the database storage requirement for a given amount of device storage.

It is interesting that the amount of network data observed in a freestanding residential house was about the same as that observed in a busy computer lab in a period of four hours. One possible explanation is that the access points in the house

had lower beacon intervals than the access points in the lab, which would mean that the access points in the house sent out beacon frames more frequently. More testing would be helpful to further understand the network conditions for both the testing sites.

Section 6.5

Experimental Support of Architecture Attributes

These experiments showcase that the architecture underpinning the SPLICEcube is scalable and extensible. In Experiment 1 and Experiment 2, we modified the integration module on the collector to enable three different queries for a single application request, as required by the Inside-Outside application. Similarly, if other applications were to be added, the architecture could be extended such that the SPLICEcube developer could tailor the collector to accommodate these applications as necessary. Additionally, in Experiment 2, there were different number of devices and different network traffic at the two testing sites, yet insertion and query times remained relatively stable, illustrating the scalability of the system.

Chapter 7

Future Work

The proposed architecture that underpins the SPLICEcube solution aims to be a scalable and extensible framework to facilitate smart-home management, and our current implementation serves as an initial prototype for this architecture. Below we suggest several improvements to both the initial prototype implementation and the overall architecture.

Section 7.1

Implementation Improvements

This section describes areas of future work to improve the current implementation of the prototype.

7.1.1. Sniff on Multiple Channels

Each cubelet currently has one interface dedicated to sniffing and sniffs on a single channel. Capturing traffic on multiple channels is important in gathering a more comprehensive sample of frames from the network and potentially detecting the presence of a rogue access point [10]. One approach is to add additional interfaces that capture Wi-Fi packets on different channels across both bands of the Wi-Fi frequency

spectrum. Another approach is to enable channel hopping on an interface so that the sniffer visits each channel periodically. An intelligent hopping strategy and an optimal hopping interval duration is important in maximizing packet collection and device detection. Looping too slowly could result in missed information on other channels. Looping too fast could result in extra overhead and delay due to excessive channel switching. One possible option is to always monitor the router channel, and then efficiently switch through the other channels. Deshpande et al. proposes a method in which measurement applications can dynamically modify the sampling strategy to refocus the monitoring system on certain types of traffic [10].

7.1.2. Decrypt MAC Frame Data

We discuss in Section 5.1.2 that our implementation can currently extract only the RadioTap header and the MAC header from a layer 2 MAC frame. This is because the home Wi-Fi devices are transmitting on the cube's network, which is protected by WPA2 in our implementation, so the frame body of the MAC frame is encrypted. Since the frame body encapsulates all the information from the layers above, decrypting it would yield useful, higher-level information about a device's transmissions. Below we have outlined the decryption process for WPA2 traffic and suggested a potential approach.

Pre-Shared Key (PSK). To decrypt the layer 2 frame body of a device transmission, we first must know the password of the Wi-Fi network to which the device is connected. The Wi-Fi password is known as the Pre-Shared Key (PSK). In our implementation, the home Wi-Fi devices are connected to the cube's network, and the SPLICEcube developer would typically know the cube network's Wi-Fi password. As a result, we can only decrypt transmissions from devices that are connected to the cube's network. Transmissions from devices that are connected to some other

password-protected Wi-Fi network cannot be decrypted since we do not know the password of these networks.

Capturing Four-Way Handshake. Additionally, we must be able to capture the four-way handshake that occurs when a device attempts to connect to the cube's network. The four-way handshake involves the exchange of random data, the authenticator nonce and supplicant nonce, between the client and the access point (AP) every time a client associates [46].

In our implementation, the cubelets could be used to capture the key exchange since they are actively sniffing. Whatever device is responsible for decrypting the frame body must then have access to this key. For example, if the cubelets are responsible for decrypting, then each cubelet must store the key for the duration of a client's connection to the network. Alternatively, the cube's collector device could be responsible for decrypting; as in, the cubelets could capture and transmit the data to the collector in its encrypted form, and then the collector could decrypt it before storing in the database. In this case, the collector would need access to the key. The design decisions are left to the discretion of the SPLICEcube developer.

It is possible that a cubelet misses capturing the handshake when the client first joins the network. In this case, we can implement a *deauthentication attack* to interrupt the connection between the client and the router [36]. The cube or a cubelet can send a deauthentication frame to the client device that forces the device to disassociate from the network. We can then have the client join the network again and capture the subsequent handshake.

7.1.3. Implement Disposal Module

In Section 4.3.4 of the proposed architecture, we envision a data disposal module in the cube to contain the database size. Although we did not implement this module

in our prototype, here is one potential approach.

The disposal module on the collector device could periodically check for data that was written to the database prior to a certain time threshold. The module could then use the SQLite *dump* utility to write this outdated data into a text file. Then the module could upload this file to cloud storage and subsequently access the database again to delete the data that has been migrated to the cloud.

7.1.4. Translate Code to C

All of the current code is written in Python, but it can be rewritten in C to increase the speed of the programs and maximize efficiency. Python was a reasonable choice for an initial implementation because it has low development cost and is easy to revise if a design decision needs modification. For future iterations of the implementation, it could make sense to write the lower-level capabilities such as sniffing, parsing, transmission, reception, and database extraction in C. The presentation layer for a GUI or front-end application may be written in a higher-level language at the discretion of the SPLICEcube developer.

Although the architecture components will remain the same, some implementation choices will need to be revised. For example, the current method for parsing Wi-Fi frames utilizes the dpkt Python library which will need to be replaced if the code is translated to C. The advantage is that C does have similar libraries for parsing, such as libwifi [24], so the concept of extracting relevant features from the frames should translate seamlessly.

7.1.5. Conduct More Robust Testing

Although the current experiments provide insight into the operation of the SPLICEcube system, further testing, such as observing the query and insertion times over a longer period of time and more trials, could enhance our understanding of the current

implementation.

One key metric that the current evaluation does not acknowledge is the throughput of data in the cubelet-cube transmissions. The approximate bytes sent per transmission and the approximate number of transmissions per minute could be helpful metrics. The cubelet-cube transmissions occur on the cube's Wi-Fi network and add additional overhead to the network. To avoid a congested network, it is important to ensure that these transmissions do not consume excessive bandwidth.

To detect devices and keep an accurate log of device network information, the cubelets must be able to maximize capture performance and successful transmission to the cube. If the cubelets cannot capture sufficient frames, they will miss device activity. If the cubelets-cube transmissions get dropped frequently, the cube will not receive data about device activity. The current evaluation does not measure capture or transmission performance. We could evaluate capture performance by comparing the cubelets' captures with a high-performance, high-speed sniffer. A simple test to observe the number of frames captured by both systems over the span of a day could provide insight into the cubelets' capturing ability. Additionally, we could evaluate cubelet transmission performance by adding a sequence number to each UDP datagram sent to the cube, allowing us to observe the number of successful transmissions and detect which datagrams were dropped.

7.1.6. Modify Database

In Section 5.3.4, we discussed that we chose to use SQLite, a file-based database, as opposed to a database engine such as MySQL. SQLite was an easy initial implementation choice and its low-memory footprint makes it well-suited to live on a resource-constrained device in a smart home. MySQL would provide more functionality because it requires a Relational Database Management Server (RDBMS) that sits between the clients and the database and manages file I/O, client connections,

query optimization, query processing, and caching [20].

Additionally, MySQL is widely used for online transactional processing (OLTP) [30]. An OLTP-based database includes 1) frequent data modification, 2) high volume of concurrent users accessing data, 3) simple transactions, and 4) indexed data sets for fast search, query, and retrieval [26]. An OLTP-based database is well-suited for the SPLICEcube system: 1) there is frequent addition of network data to the SPLICEcube database, 2) there may be multiple applications (represented by multiple threads in the integration module) requesting data, 3) the SPLICEcube database operations are relatively simple (inserts and simple queries), and 4) query response time is important so that the application can receive its requested data quickly.

We leave such database design decisions to the future SPLICEcube developer, but introduce MySQL as a potential alternative database.

Section 7.2

Architecture Improvements

This section describes areas of future work to improve the overall proposed architecture.

7.2.1. Expand Router Functionality

The proposed architecture does not specify additional capabilities for the router besides its routing functionality. Future iterations of the architecture could expand the functionality of the cube's router to allow the cube to also sniff frames. Indeed, because the router already receives frames from the home Wi-Fi clients, it could parse the frames it is already handling and record the relevant features in the database.

The cubelets are still needed because we want more than one observation point. By using the router as an additional observation location, we could gain more information

about frames, which could be useful for applications such as device localization or authentication.

7.2.2. Accommodate Other Applications

Although the Inside-Outside detection application, a type of device localization application, was successfully integrated into the SPLICEcube, there are many opportunities for other research applications to be integrated into the SPLICEcube to enhance its robustness as the home's central security and privacy controller. Some ideas of applications discussed in section Section 1.2 were device localization, network anomaly detection, device firmware update, and cube interface design. The architecture may need to be expanded to accommodate such other applications. Modifications are inevitable, but the current architecture provides the foundation for further development to support the integration of other applications.

For example, the SPLICEcube is well-suited to accommodate a learning-based network anomaly detection system that sniffs network traffic, extracts certain parameters, and learns the "normal" behavior, to subsequently detect unexpected behavior. However, the architecture may need to be extended to fully integrate such an application. For example, IoTHound, a learning-based network anomaly detection system mentioned in Section 1.2, employs a data aggregation method to segment the observed network traffic into time windows before feature extraction because having a single data point for each observed packet becomes a computational burden and because one single packet is usually not indicative of an IoT device behavior [3]. Although the database in our proposed architecture would allow the IoTHound application to retrieve necessary features from the stored network data, the database may need to be extended to allow for data aggregation before extracting the features.

Additionally, a device firmware update process, such as one proposed by Nilsson et al. in which the cube downloads new firmware and securely deploys it on a home

device [32], will require a path of communication from the cube to the device. The extensible nature of the architecture allows this expansion, and we envision adding another module to the cube to handle this communication.

7.2.3. Support Other Protocols

Currently, the SPLICEcube architecture only supports Wi-Fi but can be expanded to support other protocols such as Bluetooth and Zigbee. For example, a Bluetooth sniffing interface can be added to each cubelet to allow detection of Bluetooth frames emitted by Bluetooth devices. We envision a separate module to handle Bluetooth frame capture. A Bluetooth parsing tool would be required to parse the Bluetooth and frames and extract relevant information. We envision a separate module to handle Bluetooth frame parsing. The cubelet's transmission and the cube's reception of the parsed Bluetooth frames could occur within the existing transmission and reception modules, respectively. Additional tables may be required in the database to store the Bluetooth data in an organized and efficient way.

Supporting more protocols will allow the capture and storage of traffic from a wider range of smart devices in the home that may communicate using protocols other than Wi-Fi. Applications such as device localization or network anomaly detection can then be used to secure and manage these non-Wi-Fi devices.

7.2.4. Incorporate Security as an Architecture Attribute

In addition to being scalable and extensible, the envisioned architecture must also be secure. It is possible that an adversary targets the SPLICEcube system to gain access to the home's private information, so the SPLICEcube architecture must be designed to protect against security exploits.

Communication channels that connect the SPLICEcube components must be secure. In our current implementation of the architecture, the application-cube and

cubelet-cube communication occurs over the cube's network which is protected by WPA2. If a future SPLICEcube developer chooses to modify the communication protocol, they must think about how to maintain the confidentiality, integrity, and authentication of the transmitted data.

There must be a robust method to authenticate the devices and services that communicate with the cube and ensure the integrity of information added to the database. Any device on the cube's network can currently send information to the cube that could subsequently be stored in the database. More protective measures are needed to disallow such communication.

7.2.5. Optimize the Positioning of the Cubelets

The current architecture stipulates the use of multiple cubelets and expects the cubelets to be dispersed around the smart home to provide several observation points during network data capture. An extension of the architecture could be to explore the optimal number of cubelets and the optimal positions for the cubelets to maximize coverage. Farkas et al. propose an algorithm to find the optimal number and placement of WLAN access points for indoor positioning [18]. Similar work could be done to find the optimal setup for the cubelets.

Chapter 8

Conclusion

To realize the vision of a manageable and secure smart-home environment, we propose a system called the SPLICEcube, which consists of the cube, cubelets, and database. The cube acts as a router and central hub, the cubelets extend network coverage and assist in gathering network data, and the database stores network data.

In this thesis, we design a scalable and extensible Wi-Fi architecture and database that underpins the SPLICEcube. The architecture facilitates intelligent research applications to be integrated into the SPLICEcube to expand the functionality of the system. The architecture is designed to capture network data across the smart home, parse the network data for desired information, consolidate this information into a central repository, and allow applications to use the stored information to securely manage the devices within the home.

We built a prototype implementation of the proposed architecture and integrated the Inside-Outside research application with our implementation. We tested our implementation by deploying the SPLICEcube system in a real environment to demonstrate a successful proof-of-concept use of the SPLICEcube architecture. The integrated Inside-Outside application operated successfully and showed promising results in classifying home devices as inside or outside the home.

The SPLICEcube system can serve as a platform for researchers to test research applications that a resident within a smart home may then use to secure and manage their smart home. Our implementation may become obsolete as technology evolves, but we envision the proposed SPLICEcube architecture to remain a scalable and extensive framework for identifying, securing, and managing smart devices.

Bibliography

[1] Amazon. Amazon Alexa. Online at https://alexa.amazon.com/spa/index.html#new-oobe, visited May 2022.

[2] Y. Amar, H. Haddadi, and R. Mortier. Privacy-Aware Infrastructure for Managing Personal Data. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 571–572. ACM, 2016. DOI 10.1145/2934872.2959054.

[3] P. Anantharaman, L. Song, I. Agadakos, G. Ciocarlie, B. Copos, U. Lindqvist, and M. Locasto. IoTHound: environment-agnostic device identification and monitoring. In *Proceedings of the International Conference on the Internet of Things (IoT)*, pages 1–9. ACM, 2020. DOI 10.1145/3410992.3410993.

[4] Apple. Apple Home. Online at https://www.apple.com/ios/home/, visited Apr. 2022.

[5] Appropriate Uses for SQLite. Online at https://www.sqlite.org/whentouse.html, visited Apr. 2022.

[6] F. L. Bellifemine, C. Borean, G. Dini, P. Perazzo, and M. Tiloca. A Home Manager Application for ZigBee Smart Home Networks. In *Proceedings of the International Workshop on Networks of Cooperating Objects*, 2010.

[7] Bitdefender. Online at https://www.bitdefender.com/smart-home/#box_section, visited Apr. 2022.

[8] BrosTrend 650Mbps Linux WiFi Adapter. Online at https://www.brostrend.com/collections/linux-wifi-adapter/products/ac5l, visited Apr. 2022.

[9] Bullguard. Online at https://www.bullguard.com, visited Apr. 2022.

[10] U. Deshpande, C. McDonald, and D. Kotz. Refocusing in 802.11 Wireless Measurement. In *Proceedings of the Passive and Active Measurement Conference (PAM)*, volume 4979 of *Lecture Notes in Computer Science*, pages 142–151. Springer-Verlag, April 2008. DOI 10.1007/978-3-540-79232-1_15.

[11] dpkt. Online at https://dpkt.readthedocs.io/en/latest/, visited Apr. 2022.

[12] D. Fauri, D. dos Santos, E. Costante, J. Hartog, S. Etalle, and S. Tonetta. From System Specification to Anomaly Detection (and back). In *Proceedings of the ACM Workshop on Cyber-Physical Systems Security and Privacy (CPS-SPC)*, pages 13–24, 2017. DOI 10.1145/3140241.3140250.

[13] Google. Google Assistant. Online at https://assistant.google.com, visited May 2022.

[14] P. Gralla, Dartmouth College. An inside vs. outside classification system for Wi-Fi IoT devices, 2021. Online at https://digitalcommons.dartmouth.edu/senior_theses/215/.

[15] H. J. Hadi, S. M. Sajjad, and K. un Nisa. BoDMitM: Botnet Detection and Mitigation System for Home Router Base on MUD. In *International Conference on Frontiers of Information Technology (FIT)*, pages 1390–1394. IEEE, 2019. DOI 10.1109/FIT47737.2019.00035.

[16] M. Haenggi. 802.11 Data Link Layer. Online at https://www3.nd.edu/~mhaenggi/NET/wireless/802.11b/Data\%20Link\%20Layer.htm, visited Apr. 2022.

[17] D. Y. Huang, N. Apthorpe, F. Li, G. Acar, and N. Feamster. IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, pages 1–21. ACM, 2020. DOI 10.1145/3397333.

[18] Á. Huszák, G. Gódor, and K. Farkas. Investigation of wlan access point placement for indoor positioning. In *Information and Communication Technologies*, pages 350–361. Springer, 2012. DOI https://doi.org/10.1007/978-3-642-32808-4_32.

[19] IoT Inspector, Apr. 2022. Online at https://inspector.engineering.nyu.edu.

[20] Jay A. Kreibich. Using sqlite. Online at https://www.oreilly.com/library/view/using-sqlite/9781449394592/ch01s01.html#:~:text=Unlike%20most%20RDBMS%20products%2C%20SQLite,makes%20up%20the%20database%20engine, visited Apr. 2022.

[21] D. Kotz and T. Peters. Challenges to ensuring human safety throughout the life-cycle of Smart Environments. In *Proceedings of the ACM Workshop on the Internet of Safe Things (SafeThings)*, pages 1–7. ACM, 2017. DOI 10.1145/3137003.3137012.

[22] V. Kumar. How is Smart Home Now Becoming a Reality? Online at https://industrywired.com/how-is-smart-home-now-becoming-a-reality/, visited Apr. 2022.

[23] V. Kumar. What is MUD? Online at https://developer.cisco.com/docs/mud/#!what-is-mud/what-is-mud, visited Apr. 2022.

[24] libwifi - An 802.11 Frame Parsing and Generation library written in C. Online at https://libwifi.so, visited Apr. 2022.

[25] J. Lis. Smart Home Forecast 2021. Online at https://www.investopedia.com/terms/s/smart-home.asp, visited Apr. 2022.

[26] B. Lutkevich. OLTP (online transaction processing). Online at https://www.techtarget.com/searchdatacenter/definition/OLTP, visited May 2022.

[27] WLAN MAC protocol, WLAN MAC frame format, 802.11 Wi-Fi MAC. Online at https://www.rfwireless-world.com/Articles/WLAN-MAC-layer-protocol.html, visited Apr. 2022.

[28] 802.11 MAC Frame Generation. Online at https://www.mathworks.com/help/wlan/ug/802-11-mac-frame-generation.html, visited Apr. 2022.

[29] MQTT: The Standard for IoT Messaging. Online at https://mqtt.org, visited Apr. 2022.

[30] MySQL: Get the best insights from your data, faster than ever. Online at https://www.stitchdata.com/resources/mysql, visited May 2022.

[31] Netgear. Netgear Orbi. Online at https://www.netgear.com/home/wifi/mesh/orbi/?cid=us-best-wifi6-srch-cpc&utm_source=search&utm_medium=cpc&utm_campaign=us-best-wifi6-srch-cpc, visited Apr. 2022.

[32] D. K. Nilsson and U. E. Larson. Secure Firmware Updates over the Air in Intelligent Vehicles. In *IEEE International Conference on Communications Work-*

shops, pages 380–384, Munich, Germany, 2008. IEEE. DOI 10.1109/ICCW.2008.78.

[33] OpenWrt. Online at https://openwrt.org, visited Apr. 2022.

[34] PyShark. Online at https://pypi.org/project/pyshark/, visited Apr. 2022.

[35] Radiotap. Online at https://www.radiotap.org, visited Apr. 2022.

[36] A. Rodriguez. Deauthentication Attacks with Python. Online at https://python.plainenglish.io/deauthentication-attacks-with-python-aa5cc6eeb331, visited May 2022.

[37] Scapy. Online at https://scapy.net, visited Apr. 2022.

[38] scapy.utils. Online at https://scapy.readthedocs.io/en/latest/api/scapy.utils.html, visited Apr. 2022.

[39] Samsung. EVO Select microSD Memory Card 32 GB. Online at https://www.samsung.com/us/computing/memory-storage/memory-cards/microsdhc-evo-select-memory-card-w--adapter-32gb--2017-model--mb-me32ga-am/, visited May 2022.

[40] A. K. Simpson, F. Roesner, and T. Kohno. Securing vulnerable home IoT devices with an in-hub security manager. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 551–556. IEEE, 2017. DOI 10.1109/PERCOMW.2017.7917622.

[41] Samsung. SmartThings. Online at https://www.smartthings.com, visited Apr. 2022.

[42] E. Soltanaghaei, A. Kalyanaraman, and K. Whitehouse. Multipath Triangulation: Decimeter-level WiFi Localization and Orientation with a Single Unaided Receiver. In *Proceedings of the ACM Symposium on Mobile Computing Systems and Applications (MobiSys)*, pages 376–388. ACM, 2018. DOI 10.1145/3210240.3210347.

[43] SQLite. Online at https://devopedia.org/sqlite, visited Apr. 2022.

[44] IBM. Structured vs. Unstructured Data: What's the Difference? Online at https://www.ibm.com/cloud/blog/structured-vs-unstructured-data, visited Apr. 2022.

[45] TCPDUMP & LIBPCAP. Online at https://www.tcpdump.org, visited May 2022.

[46] D. White. Wifi Hacking & WPA/2 PSK traffic decryption. Online at https://sensepost.com/blog/2013/wifi-hacking-wpa\%2F2-psk-traffic-decryption/, visited Apr. 2022.

[47] M. Wolfe. MySQL vs. SQLite. Online at https://towardsdatascience.com/mysql-vs-sqlite-ba40997d88c5, visited Apr. 2022.

[48] Wi-Fi Security: WEP vs WPA or WPA2. Online at https://www.avast.com/c-wep-vs-wpa-or-wpa2, visited Apr. 2022.