

6-29-2018

# A Simplified Secure Programming Platform for Internet of Things Devices

Halim Burak Yesilyurt

*Florida International University*, [hyesi001@fiu.edu](mailto:hyesi001@fiu.edu)

**DOI:** 10.25148/etd.FIDC006845

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

 Part of the [Information Security Commons](#), [OS and Networks Commons](#), [Other Computer Engineering Commons](#), and the [Software Engineering Commons](#)

---

## Recommended Citation

Yesilyurt, Halim Burak, "A Simplified Secure Programming Platform for Internet of Things Devices" (2018). *FIU Electronic Theses and Dissertations*. 3788.

<https://digitalcommons.fiu.edu/etd/3788>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact [dcc@fiu.edu](mailto:dcc@fiu.edu).

FLORIDA INTERNATIONAL UNIVERSITY  
Miami, Florida

A SIMPLIFIED SECURE PROGRAMMING PLATFORM FOR INTERNET OF  
THINGS DEVICES

A thesis submitted in partial fulfillment of the  
requirements for the degree of  
MASTER OF SCIENCE  
in  
COMPUTER ENGINEERING  
by  
Halim Burak Yesilyurt

2018

To: John L. Volakis  
College of Engineering and Computing

This thesis, written by Halim Burak Yesilyurt, and entitled A Simplified Secure Programming Platform for Internet of Things Devices, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

---

Kemal Akkaya

---

Alexander Pons

---

A. Selcuk Uluagac, Major Professor

Date of Defense: June 29, 2018

The thesis of Halim Burak Yesilyurt is approved.

---

John L. Volakis  
College of Engineering and Computing

---

Andres G. Gil  
Vice President for Research and Economic Development and Dean of the  
University Graduate School

Florida International University, 2018

© Copyright 2018 by Halim Burak Yesilyurt

All rights reserved.

## ACKNOWLEDGMENTS

I would like to thank the Cyber-Physical Systems Security (CSL) group and their members, my academic advisor, my committee members, my family, and friends for being supportive and helpful during my studies and to the Department of Electrical and Computer Engineering, FIU for providing me with this opportunity. I would also like to acknowledge the support of my thesis by US National Science Foundation under the grant number NSF-CAREER-CNS-1453647, NSF-ACI-1339781, and Florida Center for Cybersecurity's Capacity Building Program.

ABSTRACT OF THE THESIS  
A SIMPLIFIED SECURE PROGRAMMING PLATFORM FOR INTERNET OF  
THINGS DEVICES

by

Halim Burak Yesilyurt

Florida International University, 2018

Miami, Florida

Professor A. Selcuk Uluagac, Major Professor

The emerging Internet of Things (IoT) revolution has introduced many useful applications that are utilized in our daily lives. Users can program these devices in order to develop their own IoT applications; however, the platforms and languages that are used during development are abounding, complicated, and time-consuming. The software solution provided in this thesis, PROVIZ+, is a secure application development software suite that helps users create sophisticated and secure IoT applications with little software and hardware experience. Moreover, a simple and efficient domain-specific programming language, namely Panther language, was designed for IoT application development to unify existing programming languages. In addition to these contributions, PROVIZ+ supports a novel secure over-the-air programming framework, namely SOTA, using Bluetooth and WiFi as well as serial programming.

In this thesis, we explain the features of PROVIZ+'s components, how these tools can help develop IoT applications, and SOTA. We also present the performance evaluations of PROVIZ+ and SOTA.

## TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION . . . . .	1
2. BACKGROUND INFORMATION . . . . .	3
2.1 Internet of Things (IoT) . . . . .	3
2.2 Over-the-air Programming . . . . .	6
2.3 Automatic Code Generation Tools . . . . .	6
2.4 Parser Generators . . . . .	7
3. RELATED WORK . . . . .	8
3.0.1 Visual IoT Sensor Application Development Tools . . . . .	9
3.0.2 Secure Over-the-Air Programming Frameworks . . . . .	10
3.0.3 IoT Platforms . . . . .	11
4. PROVIZ+: A SIMPLIFIED SECURE PROGRAMMING PLATFORM FOR INTERNET OF THINGS DEVICES . . . . .	13
4.1 Introduction . . . . .	13
4.2 Design and Implementation . . . . .	15
4.2.1 Main Application . . . . .	16
4.2.2 Proviz Client Applications . . . . .	26
4.2.3 PROVIZ+ Android Tablet Application . . . . .	29
5. SOTA: SECURE OVER-THE-AIR PROGRAMMING FRAMEWORK . . . . .	32
5.1 Introduction . . . . .	32
5.2 Threat Model and Assumptions . . . . .	35
5.3 SOTA Architecture . . . . .	36
5.3.1 Remote Programmer . . . . .	36
5.3.2 IoT Device Communication Module . . . . .	40
5.3.3 IoT Device . . . . .	41
5.3.4 Provision of Security . . . . .	42
6. PERFORMANCE EVALUATION . . . . .	46
6.1 Testbed and Methodology . . . . .	46
6.2 Results . . . . .	48
6.2.1 Time Analysis . . . . .	48
6.2.2 CPU, Memory, and Storage Analysis . . . . .	51
6.2.3 Security Analysis . . . . .	53
7. CONCLUSION . . . . .	55
7.1 Benefits of the PROVIZ+ Software Suite . . . . .	55
7.2 Conclusions and Future Work . . . . .	56

BIBLIOGRAPHY . . . . . 58

## LIST OF TABLES

TABLE	PAGE
2.1 Specifications of Raspberry Pi models . . . . .	4
2.2 Arduino IoT device specifications . . . . .	5
6.1 Specification of devices that were used in the experiments. . . . .	46
6.2 The PROVIZ+ main application task completion time statistics. . . . .	49
6.3 The SOTA Framework experiment results. . . . .	50
6.4 The PROVIZ+ Software Suite resource usage statistics. . . . .	51
6.5 Memory Occupation of the STK500 Bootloader vs. the SOTA Bootloader. . . . .	52

## LIST OF FIGURES

FIGURE	PAGE
2.1 Raspberry Pi IoT device [RASa]. . . . .	4
2.2 Arduino IoT device [ARD]. . . . .	5
4.1 PROVIZ+ main components. . . . .	16
4.2 Visual-based programming tool. . . . .	17
4.3 Code-based programming tool. . . . .	17
4.4 Explanation of the PROVIZ+ main application. . . . .	18
4.5 Steps of IoT device configuration task. . . . .	19
4.6 Steps of adding a sensor to IoT device in the visual-based programming tool. . . . .	19
4.7 PROVIZ+ code generation process diagram. . . . .	21
4.8 Sample distance measurement sensor application in the Panther language.	25
4.9 Components of the PROVIZ+ Raspberry Pi client application. . . . .	29
4.10 Notification of the PROVIZ+ Android tablet application to the existing IoT devices around. . . . .	30
4.11 PROVIZ+ topology importer to tablet application tool. . . . .	31
4.12 Screenshots of the PROVIZ+ Android tablet application. . . . .	31
5.1 The SOTA system structure . . . . .	35
5.2 Major components of the SOTA Framework. . . . .	37
5.3 STK500 communication protocol parser. . . . .	38
5.4 SOTA packet structure built upon the STK500 Protocol. . . . .	39
5.5 Watchdog supported micro-controller reset function. . . . .	42
5.6 Steps of the authentication process. . . . .	44
6.1 The SOTA Framework performance analysis testbed. . . . .	47
6.2 Experimental setups: 4 different topologies studied. . . . .	47

# CHAPTER 1

## INTRODUCTION

The Internet of Things (IoT) devices have become very popular in various critical and non-critical usage fields. Many people have started to use IoT devices realizing their benefits in their lives and scientific research [MSPC12, CGOF13]. Because of the popularity of IoT devices, developers gravitate toward the development of IoT applications. These applications utilize IoT devices that contain various peripherals (e.g, sensors, cables, communication shields) and controller units (e.g, micro-controllers, micro-processors, security modules); they read data from sensors and then process it in the controller unit. A wide variety of sensors can be used within the IoT context with purposes such as measuring the temperature of a room, detecting movement in a vehicle, and establishing the identity of a user.

The continuous increase of attention to IoT applications has yielded a new ecosystem of platforms and languages [DEDP15, CBS<sup>+</sup>18]. This ecosystem caused a need for a framework that facilitates secure IoT application development, which unifies a broad range of IoT development environments and languages. Those interested in developing applications for various IoT devices have to learn programming languages and how to operate several development environments. For example, consider someone who wants to develop a temperature measurement system using an Arduino-based IoT device. To do so, the person needs to know how to program applications written in the C language as well as be acquainted with hardware-related concepts regarding the measurement of sensors and the Arduino IoT device. Moreover, if the person wants to transfer their application to run on another IoT device such as a Raspberry Pi, this person will need to port the code into this new devices programming platform by rewriting the application in another programming language (e.g., Python). In addition to the cost of learning a broad range of

programming languages and platforms for different IoT devices, over-the-air programming Bluetooth and WiFi data transmission protocols require great attention and an in-depth knowledge of IoT devices. To address these issues, this thesis aims to design a software suite that facilitates the need to learn so many technologies: it helps to reduce the barrier of entry for secure IoT application development. The software suite solution, PROVIZ+, which is introduced in Chapter 4, is a secure IoT application development software suit that helps users to develop IoT applications without asking for any software or hardware experience. PROVIZ+ supports various IoT development boards including major brands such as Raspberry Pi and Arduino. Moreover, the software suite includes a domain-specific programming language, namely Panther language, which is introduced in Chapter 4, to unify the existing programming languages to simplify the development of IoT applications. In addition to the Panther language, PROVIZ+ has visual programming and script-based programming tools which enable users to program IoT devices either by dragging and dropping using the visual programming tool or writing code in the Panther language using the code-based programming tool. PROVIZ+ also supports automated Bluetooth, WiFi, and wired data transmission for Arduino and Raspberry Pi IoT devices. Besides, it provides an easy IoT application and firmware transfer mechanism to supported IoT devices. This firmware upload process can be achieved either over-the-air or through physical communication over the serial data cables. Since the firmware transfer is a crucial task for different IoT applications, we designed and implemented a novel over-the-air programming feature for PROVIZ+. Also, as existing over-the-air programming protocols for low-powered IoT devices do not include adequate security mechanisms to defend against different malicious attacks, we designed and implemented a secure over-the-air programming framework, named SOTA, which is introduced in Chapter 5.

## CHAPTER 2

### BACKGROUND INFORMATION

This chapter provides background information about Internet of Things and corresponding or related technologies that are used in the development of PROVIZ+ and the SOTA framework.

#### 2.1 Internet of Things (IoT)

The Internet of Things (IoT) is the network of physical devices that have actuators, sensors, and computation and communication units in order to exchange data with each other over a communication protocol. IoT applications can be created using several types of customized hardware, single board computers, and micro-controllers [PRO, CUB13]. Since Raspberry Pi and Arduino are selected as natively supported IoT devices in PROVIZ+, they are introduced in the following sub-sections.

#### Raspberry Pi

The Raspberry Pi is a single board computer (SBC) that has a system on chip (SOC) as the central processing unit with an integrated ARM central processing unit (CPU) and a graphics processing unit (GPU). Since it has a GPU and it can run a desktop environment smoothly, it can be used as a personal computer. In addition to being powerful, Raspberry Pi devices are credit card-sized computers, which means that they are portable and can easily be moved to another location. Pis have multiple purpose I/O pins which enable developers to create IoT sensor applications. Moreover, they support WiFi and Bluetooth connection protocols to support data transmission over wireless networks. Because Raspberry Pis have a powerful microprocessor, developers use them for IoT applications that require high

Table 2.1: Specifications of Raspberry Pi models

Device Name	CPU	Memory	Ethernet	Wireless	Bluetooth	OS
Raspberry Pi Zero	1GHz	512 MB DDR2	None	None	None	Linux
Raspberry Pi 2 Model B	0.9 GHz	1 GB DDR2	Yes	None	None	Linux
Raspberry Pi 3 Model B	1.2 GHz	1 GB DDR2	Yes	802.11n	Bluetooth 4.1	Linux
Raspberry Pi 3 B+	1.4 GHz	1 GB DDR2	Yes	802.11 b/g/n/ac	Bluetooth 4.2	Linux
Raspberry Pi A+	700 MHz	256 MB DDR2	Yes	Yes	Bluetooth 4.1	Linux
Raspberry Pi Zero WH	1 GHz	512 MB DDR2	Yes	802.11n	Bluetooth 4.1	Linux

computation power. In the scope of this thesis, we use Raspberry Pi 3 Model B shown in Figure 2.1 in performance evaluations of the PROVIZ+ software suite. The specifications of the different Raspberry Pi models can be seen in Table 2.1.



Figure 2.1: Raspberry Pi IoT device [RASa].

## Arduino

Arduino is an hardware and software development company that builds single-board computers and micro-controller kits. These kits are distributed as an open-source hardware with large open-source community support.

Because they do not have SOC, they mostly do not support embedded communication protocols unlike Raspberry Pis. Despite having limited capabilities, Arduino devices can perform sophisticated tasks by utilizing sensor and communication shields, which can be added to the board. These shields can be WiFi, Zigbee, or Bluetooth communication shields to support data transmission, and also they can

Table 2.2: Arduino IoT device specifications

Device Name	CPU	Memory	Ethernet	Wireless	Bluetooth	OS
Arduino YN	400 MHz & 16 MHz	64 MB DDR2	Yes	Yes	None	Linux
Arduino Uno	16 Mhz 8-Bit	2 Kb	None	None	None	Bare Machine
Arduino Mega 2560	16 MHz	8 KB	None	None	None	Bare Machine
Arduino Due	84 MHz	96 KB	None	None	None	Bare Machine
Arduino LilyPad USB	8 MHz	2.5 KB	None	None	None	Bare Machine
Arduino Zero	48 MHz	32 KB	None	None	None	Bare Machine

be sensor shields to increase the capabilities of Arduinos such as measuring temperature, pressure, or humidity. Arduino IoT devices have a low-level I/O operation support; they support Inter-Integrated Circuit (I2C) and Serial Peripheral Interface Bus (SPI) communication ports. Interestingly, a large majority of Arduino IoT device models have the same brand of micro-controller, Atmel [ATW]. Having the same micro-controller makes it possible to develop common solutions for many IoT products without needing to differentiate by the type of Arduino IoT devices. In addition to wide communication port support, they are also energy-efficient as they employ low power ATMEL micro-controllers. Table 2.2 shows the specifications of the popular Arduino IoT devices. In the scope of this thesis, we use the Arduino Mega 2560 IoT device that is shown in Figure 2.2.

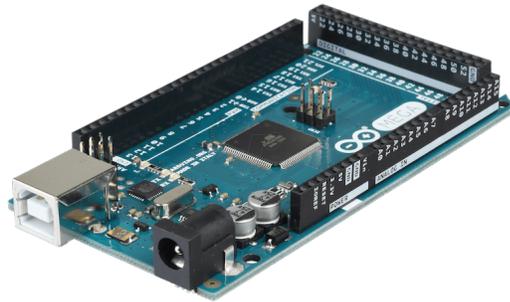


Figure 2.2: Arduino IoT device [ARD].

## 2.2 Over-the-air Programming

Over-the-air programming is a method of programming electronic devices remotely without any physical contact. It can be achieved using Bluetooth or WiFi communication protocols. These protocols carry firmware pieces for target micro-controllers or microprocessors for purposes of updating the firmware due to security concerns or to add new features to it. The majority of Arduino-based IoT devices contain Atmel micro-controllers which utilize the STK500 Bootloader [ATM] to communicate with the remote programmer during the firmware transfer. In the SOTA framework, the STK500 Bootloader was modified to make a secure over-the-air programming framework.

## 2.3 Automatic Code Generation Tools

Automatic code generation tools produce the source code of an application for various devices from templates. These templates carry necessary information that are taken from users to be utilized in the code generation of the target application. In the scope of this thesis, an automatic code generation tool was used to generate the source code for the IoT devices used in this project by only using a code generation template. As such, users only require filling a code generation template so as to generate the source code for Arduino and Raspberry Pi IoT devices. Specifically, the Apache Freemarker [APA] library was selected to implement the automatic code generation tool for the PROVIZ+ software suite.

## 2.4 Parser Generators

Parser generators produce the source code of a parser that are for reading, understanding, and executing binary files or text. Generated parsers are generally used in parsing structured text operations. In the scope of this thesis, a parser generator library is added to the PROVIZ+ software suite in order to parse the Panther language. In PROVIZ+, the generated parser first utilizes the script that is written in the Panther language, then fills the code template to produce the source code for a target device. For this, the ANTLR library [ANT] was selected as a parser generator tool to generate the parser of the Panther language in the PROVIZ+ software suite.

## CHAPTER 3

### RELATED WORK

In this chapter, the related work in visual IoT sensor application development tools are given and compared with PROVIZ+. We also provide the related work for the SOTA framework in the next section. Finally, we discuss other IoT programming platforms.

The PROVIZ+ software suite presented in this work is inspired by the previous software project also titled PROVIZ [RCUB16]. That project was intended to provide a framework to visualize and program wireless sensor networks (WSNs). It allowed the user to develop wireless sensor applications using a scripting language that could be easily written and reused by any developer [RCUB16]. Although built on the same concept, the new PROVIZ, PROVIZ+, has some crucial differences. These include:

- PROVIZ+ is built for IoT devices. The first generation of the PROVIZ project was created for WSNs and their sensor nodes such as the MICAz [CRO]. However, PROVIZ supports obsolete sensor devices, which are stale and deprecated. PROVIZ+ works smoothly with a broad range of popular IoT devices. It creates sensor applications by programming IoT boards such as the Arduino and Raspberry Pi devices.
- PROVIZ+ supports over-the-air programming, unlike PROVIZ.
- The PROVIZ+ software suite includes an Android App that enables users to monitor flexibly their IoT devices' sensor data.
- As opposed to PROVIZ, the graphical user interface of PROVIZ+ was designed with a step-by-step guide that prompts the user for the information

needed in a sequential manner. In this way, the user can understand what occurs on the back end as they program the device.

### **3.0.1 Visual IoT Sensor Application Development Tools**

PROVIZ+ is the first software of its kind to have committed itself to ease and simplicity across platforms for IoT devices. Some visual development tools for wireless sensor networks exist, including Viptos [CLZ06], SensorSim [PSS00], and WISDOM [VIE05]. Viptos is a graphical development tool that programs WSNs in TinyOS [TIN] using diagrams of TinyOS components. This framework integrates the development of WSN applications with hardware visualization to ease the user development process. WISDOM is a modular application development tool. Like PROVIZ+, it has the capacity to program many different sensor platforms within the same network. It uses a modular system to send different programs to different sensors in the network to build the most versatile network. Both these platforms achieve similar goals to PROVIZ+ in terms of heterogeneous networks, network deployment, and visual development tools. However, they are confined to the traditional WSN realm. PROVIZ+ is the only software to have also achieved the goal of secure programming capabilities of heterogeneous IoT networks. Furthermore, SensorSim aims to develop a simulation environment for sensor networks. The authors of it designed a micro sensor node model that covers radio, power, and battery features of the simulated sensor nodes. Also, SensorSim includes a scripting language that can simply develop virtual sensor nodes in a simulation environment. PROVIZ+ is different from SensorSim because it has the capability of developing physical IoT applications instead of having a simulation environment, and secure over-the-air programming.

### 3.0.2 Secure Over-the-Air Programming Frameworks

Secure and wireless code-dissemination has been the focus of several useful studies in the literature [KW05, DHM05, RSUB12]. Especially, given the resource-limited nature of IoT devices, energy-efficient code dissemination is vital. Since most of the sensor applications based on embedded systems are limited regarding energy source, computation power, and usable memory size, energy efficient code dissemination is one of the vital aims of PROVIZ+, and it is also a concern for these related studies, [KW05], [DHM05], [RSUB12]. In [ABB<sup>+</sup>12], the authors propose a new framework called SenSeOp for a selective and secure over-the-air programming protocol for WSNs. In their study, they used asymmetric encryption with Elliptic Curve Cryptography to protect the firmware against cybersecurity attacks. In addition to the asymmetric cryptography used in this work, [LK] uses hash functions instead of public key cryptography to provide a secure sensor network programming method. According to authors, the signature-based public key infrastructure (PKI) might produce an overhead for embedded devices and wireless sensor nodes. In this study, SHA or MD5 supported hash chains are used to provide security. Also, in [HST08], the work aims to offer over-the-air programming techniques using rateless codes. In [RL03], the authors aim to distribute the firmware wirelessly to wireless sensors by only sending the changed part of the firmware (i.e., delta) to sensor nodes. Another useful study [LGN06] classifies and compares network reprogramming protocols in terms of security, survivability, and performance metrics. Most of the attacks mentioned in this work [LGN06] are directly related to the packet routing-related attacks and the solution provided in the work also utilizes a symmetric key cryptography as in SOTA. In a different study [KW03], the authors present attacks on secure routing and its countermeasures in order to mitigate possible damage. Besides these studies, there is a more cloud-based IoT device programming work [NSV<sup>+</sup>13]. This

work introduces the PatRICIA [NSV<sup>+</sup>13] framework as a high level, end-to-end, cloud-based IoT programming framework. It has a data persistence layer as well as a cloud run-time layer. Since data persistence requires extensive memory and high input-output traffic, this approach is more suitable for advanced IoT devices that have more memory and computational power than low-cost Arduino-based IoT boards.

Furthermore, the topic of secure over-the-air programming has become popular in the vehicular networks domain. In the [NL08] study, the authors used symmetric encryption to offer secure firmware updates over the air. Their solution aims to provide data integrity, authentication, confidentiality, and freshness. Also in this study [NSN08], the same authors proposed a solution for self-verification of downloaded firmware to detect any modification of firmware during the flashing and downloading phases. Finally, another study [ISR<sup>+</sup>11] proposes a secure firmware update protocol that can be used with not only hard-wired, but also over-the-air data transmission.

Our proposed solution in this thesis, the SOTA framework, is different from the aforementioned studies as firstly, it focuses on providing an over-the-air programming solution to IoT devices; secondly, it proposes an open-source configurable implementation of over-the-air programming framework to IoT devices that have low-power, tiny micro-controllers such as the Atmel chipsets; and thirdly, it provides a comprehensive security services, including confidentiality, authentication, and integrity.

### **3.0.3 IoT Platforms**

The development of IoT platforms has been popular among technology companies. And, today there are many commercial IoT platforms in the market [THI, AMA,

GOO, APP, IBM]. These commercial IoT platforms focus on the cloud-based sensor data analytics rather than programming IoT devices. They mostly receive sensor data from IoT devices and then process them to visualize on a screen. In addition to this, these platforms can only work with specific IoT devices and are closed-source. They do not provide a flexible platform for extensions by the developers nor users.

PROVIZ+ is different from the aforementioned platforms because of the following reasons. Firstly, PROVIZ+ primarily focuses on IoT device programming, and can program different IoT devices. Secondly, PROVIZ+ proposes a non-commercial open source and extendable software suite. Thirdly, PROVIZ+ has a secure over-the-air programming capability to program IoT devices remotely. Finally, PROVIZ+ contains a domain-specific programming language to easily program the IoT devices and applications, unifying the existing IoT programming languages for a more user-friendly development process.

## CHAPTER 4

### **PROVIZ+: A SIMPLIFIED SECURE PROGRAMMING PLATFORM FOR INTERNET OF THINGS DEVICES**

The PROVIZ+ software suite is an application development environment for IoT devices. PROVIZ+ is explained in detail in this chapter. The organization of this chapter is as follows: In section 4.1, we present introductory information about the PROVIZ+ software suite; then, we give details about our design and implementation of PROVIZ+.

#### **4.1 Introduction**

The use of IoT devices such as Arduinos and Raspberry Pis has increased in recent years [RW12, RASb]. These powerful single-board computers allow users to learn about the power of micro-computing while enabling them to digitally solve real-world problems [SPBS13].

The continuous increase in IoT devices has created a new ecosystem of novel platforms and languages [DEDP15, CBS<sup>+</sup>18]. This ecosystem creates a need for a framework that can facilitate the secure development of IoT applications and can unify a broad range of IoT application development environments and languages. Otherwise, those interested in developing applications for various IoT devices have to learn programming languages and how to setup several development environments.

To help the aforementioned circumstances, the PROVIZ+ software suite is introduced in this chapter. PROVIZ+ is a cross-platform, user-friendly secure sensor application development software suite. It allows for programming a variety of IoT single-board computers using novel visual-based or code-based programming tools.

The programming of IoT boards has traditionally required an extensive background on hardware and software components of the target device. This has resulted in a steep learning curve for the developers in this environment. For example, consider someone who wants to develop a temperature measurement system using an Arduino-based IoT device. To do so, the person needs to know how to program in the C language as well as be acquainted with hardware-related concepts regarding measurement sensors and the Arduino-based IoT device. Moreover, if the same person wants to transfer the application to run on another IoT device such as a Raspberry Pi, this person will need to port the code into this new devices programming platform by re-writing the application in another programming language (e.g., Python). In addition to the cost of learning a broad range of programming languages and platforms for different IoT devices, vital IoT device functions and operations such as over-the-air and serial programming, Bluetooth and WiFi data transmission protocols, and IoT device management require great attention and an in-depth knowledge of IoT devices. PROVIZ+ is designed with an easy all-in-one programming feature to mend this dissonance. This feature allows users to program any of these boards using only the Panther language.

The PROVIZ+ software suite's main contributions are listed as follows:

- PROVIZ+ offers support to different IoT devices platforms including Arduinos and Raspberry Pis. This extends to the broad range of sensors that these boards support (e.g., Adafruit BME280 12C [Sys]).
- PROVIZ+ provides an automatic firmware upload mechanism for a generated firmware to transfer to a target IoT device. It supports WiFi, Bluetooth, and universal serial bus communications.

- PROVIZ+ presents an easy-to-use graphical user interface that enables users of all knowledge and backgrounds to readily create a range of sensor applications from simple IoT applications to complex research experiments.
- PROVIZ+ protects the firmware during the over-the-air programming for IoT devices with a novel over-the-air programming framework, called SOTA that is introduced in Chapter 5.
- PROVIZ+ has an Android tablet application for location independent tracking of sensor data.
- PROVIZ+ includes a simple, comprehensive language that consolidates the sensor application development of all IoT boards to a single language.
- PROVIZ+ offers a built-in visual representation of the devices and sensors in network that can be easily exported as a JSON file to be shared with others.

## 4.2 Design and Implementation

PROVIZ+ is a secure application development platform for IoT devices, and it consists of four components: a main application, a Raspberry Pi client application, an Arduino client application, and a tablet application. All components, the relations between them, and which operating system they support can be seen in Figure 4.1.

The source code of the PROVIZ+ software suite was written in the JAVA and C programming languages, and the graphical user interface (GUI) was designed by utilizing JAVA FX library. For the PROVIZ+ Android tablet application, JAVA Android Development Kit was used to develop an Android tablet device compatible application. Moreover, PROVIZ+ has a native SOTA over-the-air programming framework support to make secure firmware transfer for tiny IoT devices and this

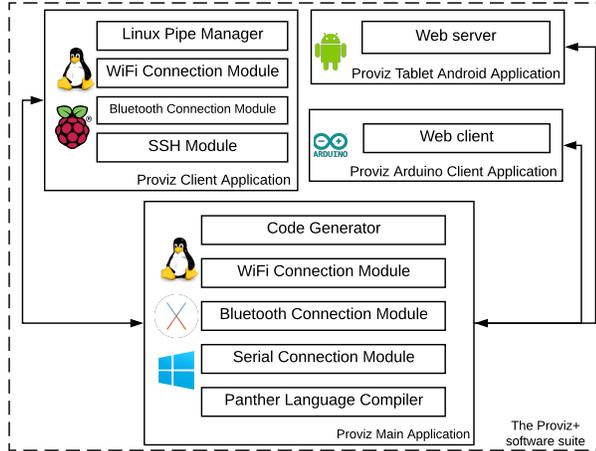


Figure 4.1: PROVIZ+ main components.

framework was implemented using the C language as well as the native Atmel micro-controller development library.

Since the PROVIZ+ project is mostly written in the JAVA programming language and runs on Java Virtual Machine (JVM), it supports three operating systems: Linux, Windows, and OS X. To create flexibility between components in the PROVIZ+ software suite, we designed it as a modular project. Future developers can quickly add new components to extend the capability of PROVIZ+.

Instead of developing PROVIZ+ as a command line software, we designed and implemented a GUI to provide a robust and user-friendly experience for all users, regardless of their technical background.

### 4.2.1 Main Application

The PROVIZ+ main application is a comprehensive software that supports two ways of programming sensor applications for IoT devices: a visual programming and a code-based programming tool. A user can program IoT devices using either

the visual programming tool, which can be seen in Figure 4.2 or the code-based programming tool that can be seen in Figure 4.3.

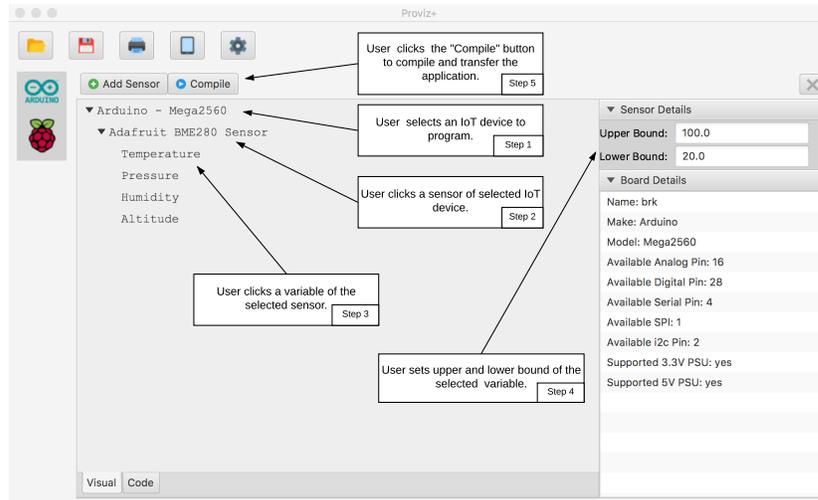


Figure 4.2: Visual-based programming tool.

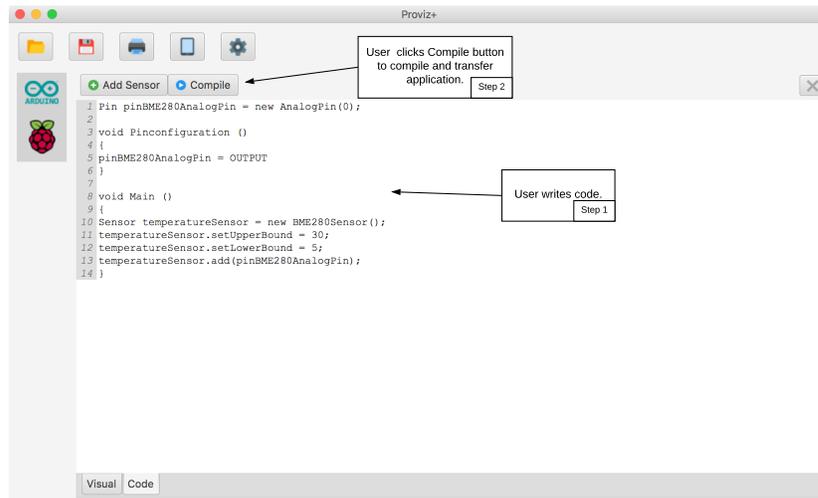


Figure 4.3: Code-based programming tool.

An example usage scenario for programming can be specified in Figure 4.4: The user drags and drops the IoT device logo from the board selection toolbar and then

makes a right click to get more options. These options are *Program*, *Configure*, and *Details*.

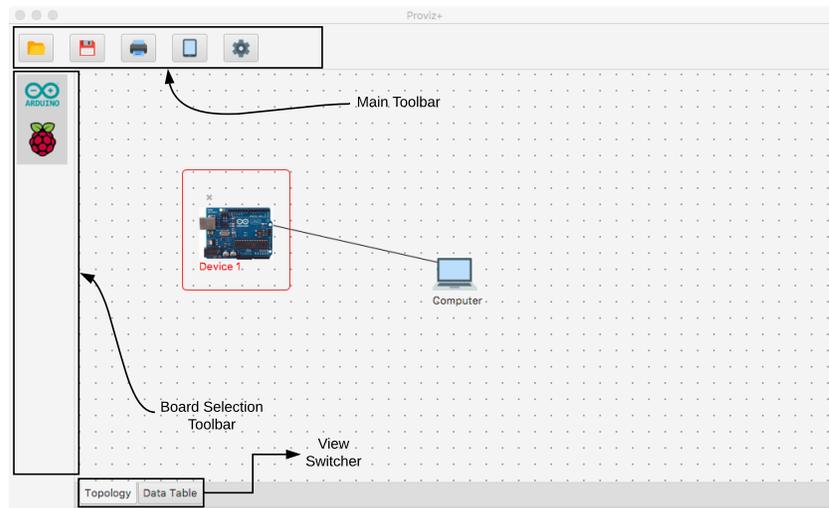


Figure 4.4: Explanation of the PROVIZ+ main application.

The user clicks *Configure* to configure the board then the IoT device properties window, seen in Figure 4.5a, is shown to the user for getting input about the selected IoT board. The user fills the required input fields, then clicks *Next* to navigate to the connection type selection screen, seen in Figure 4.5b. After selecting one of the options on the connection type selection screen, PROVIZ+ handles creating a connection between the IoT device and the main application. When the activation of the connection between the user's computer and the IoT device is done, PROVIZ+ shows the confirmation message screen to let user know about the result of the configuration process. This confirmation message screen can be seen in Figure 4.5c. A configuration of the IoT device involves selecting a connection type, board name, and board type.

After the configuration process, the user will be able to add sensors on to the IoT development board by clicking on the *Program* option in the device programming screen. PROVIZ+ does not let the user program any IoT device without doing

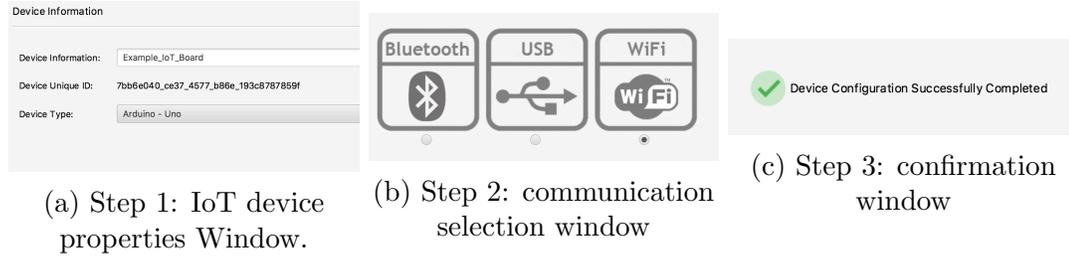


Figure 4.5: Steps of IoT device configuration task.

the configuration first. The visual programming tool will welcome the user as the default tool; however, the user can change to the code based programming tool by selecting the option from the view switcher in both programming windows. The user can add new sensors using the sensor add wizard to the selected IoT device. The method for adding a sensor to the IoT device can be followed in Figure 4.6.

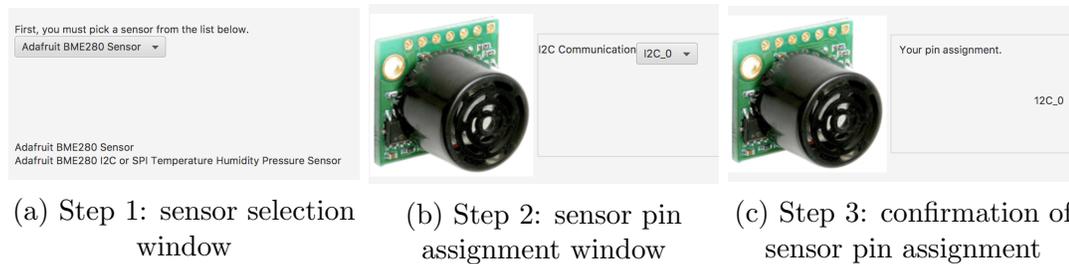


Figure 4.6: Steps of adding a sensor to IoT device in the visual-based programming tool.

Then, the user can change the upper bound and lower bound values related to the sensor or board properties in the visual programming tool, which can be seen in Figure 4.2. In addition to the visual programming tool, the user can create their own IoT applications using the code-based programming tool as seen in Figure 4.3. When the user clicks the compile button in the device programming window, PROVIZ+ creates the required source code of the IoT application. This code compiles to prepare the transfer over the firmware distribution. When the firmware distribution is over, it starts to accept a connection request from the programmed

IoT board to visualize the gathered data on the topology view or the data table view in Figure 4.4. In addition to the aforementioned usage scenario, the user can save the existing topology to continue later, transfer the topology to the PROVIZ+ Android application, or customize the user interface components to increase the usability of PROVIZ+.

As noted earlier, all graphical components were designed and implemented using the JAVA FX library. Actually, we first designed and implemented the user interfaces of PROVIZ+ using the JAVA Swing library; however, we realized that it had a compatibility issue with high-resolution computer screens. It was showing a GUI smaller than their actual size due to the high-resolution screen bug in the Java Swing library [JAV]. After that, we decided to replace it with JAVA FX.

As can be seen in Figure 4.1, the PROVIZ+ main application includes five main components: a code generator, a WiFi connection module, a Bluetooth connection module, a serial connection module, and lastly the Panther programming language compiler. In the following sub-sections, the main components will be explained in further detail.

### **Code Generator**

The main purpose of PROVIZ+ is programming IoT development boards with a broad range of various purpose sensors. To be able to develop secure IoT sensor applications, the source code of the application needs to be created by PROVIZ+. This generated source code varies depending on the device. To illustrate this, if the user chooses a Raspberry Pi as a target device, the source code can be written in Python or Java. On the other hand, if the user selects Arduino for the IoT application, the source code of the Arduino project should be written in the C language.

While we were designing the PROVIZ+ software architecture, we specified that instead of generating the user’s sensor application source code from scratch, producing it from a general template would be more efficient and easier to manage. When the user selects an icon for the IoT device from the board selection toolbar at the main PROVIZ+ screen, shown in Figure 4.4, it automatically creates a template class for it. That template class contains information about the selected board such as the board type, connection type, and how many sensors it will have. When a user adds new sensors to the board, the new sensors are attached to a sensor array that is in the board’s template class. This template class is used to generate the source of the sensor application, and this can be seen in code generation steps in Figure 4.7. The code generator accepts the board template to get the detailed information

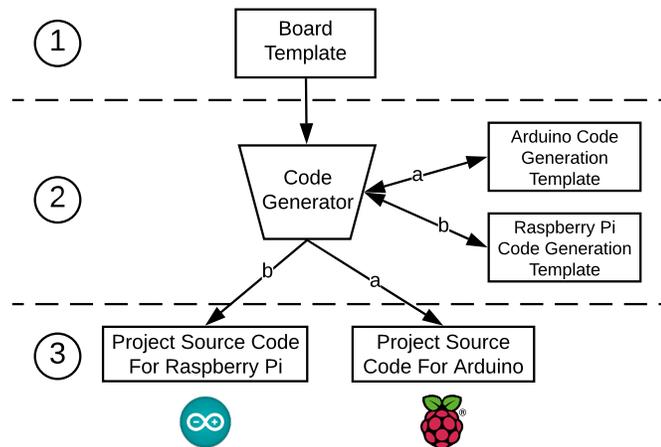


Figure 4.7: PROVIZ+ code generation process diagram.

about the target board. Afterwards, it gets the code generation template to fill the necessary blanks in source code templates to create an actively working source code. Currently, PROVIZ+ supports two IoT development boards: Arduino and Raspberry Pi; however, these supported IoT development boards can be extended

by adding new templates due to the modular structure of PROVIZ+. Note that the Apache Freemarker framework [APA] was used to develop code generator.

## **WiFi Connection Module**

PROVIZ+ supports WiFi data communication to transmit sensor data, and securely program the IoT devices over-the-air. The sensor data transmission is established using the TCP protocol stack for Arduino and Raspberry Pi devices. Even though the TCP connection is slower than UDP, TCP is selected as PROVIZ+'s native transmission protocol. The reason behind this is that any non-transmitted sensor data can cause catastrophic consequences and the TCP protocol assures that each communication packet arrives seamlessly to the destination unlike the UDP protocol. When a user finishes designing an IoT application and wants to transfer it to the target IoT board, the user transmits the firmware physically only once. This physical firmware transmission copies firmware using a USB device for Raspberry Pis and serial data transmission for Arduino devices. After that, the user will be able to update the device's firmware over-the-air instead of programming physically using the SOTA framework for Arduino devices and Secure File Transfer Protocol (SFTP) for Raspberry Pi devices. SFTP is used for the firmware distribution to Raspberry Pi devices to provide confidentiality of firmware. When the user starts over-the-air programming for Raspberry Pis, PROVIZ+ initiates SFTP connection to target Raspberry Pi development boards. This SFTP connection works as a passwordless File Transfer Protocol (FTP) by utilizing the public key cryptography authentication. A public key of the target device is transferred to the main PROVIZ+ application during the initial firmware transfer.

Arduino is a single board micro-controller, and it uses mostly Atmel micro-controllers. Normally, Atmel micro-controllers use STK500 Communication Proto-

col [ATM] to perform firmware update; however, STK500 Communication Protocol does not have enough security mechanism to protect the firmware during over-the-air programming. In the scope of this thesis, we also developed a novel secure over-the-air programming framework (SOTA) that is introduced in Chapter 5 for Atmel-based IoT devices. Since Arduino boards mostly use Atmel micro-controllers, we integrated the SOTA framework into the PROVIZ+ software suite in order to provide secure over-the-air capabilities for the Arduino devices. Specifically the *ArduinoFirmwareUpload* JAVA class was implemented in order to program Arduino devices remotely using the SOTA secure over-the-air programming framework.

Over-the-air programming and data aggregation occur over the same TCP connection to minimize interactions between the PROVIZ+ main application and Arduino. As, Arduino boards do not have WiFi connection capabilities, in this thesis, the ESP8266 WiFi module [ESP] was used to bring WiFi connection capabilities to Arduino. This WiFi module supports transparent data transmission over Universal Asynchronous Receiver Transmitter (UART), which is a vital feature for over-the-air programming. When the main application wants to update the firmware, it sends a reboot command to Arduino, then Arduino resets itself using the watchdog timer-based reset method. When Arduino resets itself, it runs the SOTA bootloader instead of the actual sensor application, and it receives the SOTA and STK500 Communication Protocol commands to replace the current firmware. In the transparent data communication protocol, ESP8266 works as a communication bridge, but ESP8266 does not change the context of a packet that is coming from a sender and it directly transmits to a recipient.

## **Bluetooth Connection Module**

PROVIZ+ uses the Bluetooth Classic connection during the firmware distribution and sensor data transmission to the PROVIZ+ main application. Raspberry Pi has a built-in Bluetooth adapter; however, Arduino does not have any Bluetooth module. Before sending firmware to Arduino or getting sensor data from the Arduino board, the selected board needs to be configured. This configuration includes the Bluetooth pairing process between the Bluetooth module and the user's computer. After the completion of the pairing operation, the user may send firmware updates to a device. The major difference between Bluetooth support and WiFi support is that Bluetooth does not need to physically program devices using serial data transmission first. For this, we use Bluefruit EZ-Link Bluetooth Shield [BLU] to provide the aforementioned Bluetooth benefits to the user because it supports over-the-air programming for Arduino devices. We developed the *BluetoothManager* JAVA class to handle all the necessary Bluetooth data transmission operations.

## **Serial Connection Module**

Serial Connection Module is the third option for programming IoT devices, which requires physical contact. When a user connects IoT boards with the USB cable to a computer, it creates device object in the */dev* folder for UNIX-based operating systems or the *COM* port object for Windows. This device or port objects can read incoming messages from the target IoT device and send new firmware to the target IoT device.

## **Panther Programming Language Compiler**

The Panther programming language is a domain-specific programming language that was designed for the PROVIZ+ software suite. Before the Panther programming

language, whenever a user wanted to program their IoT development board, they needed to learn a broad range of programming languages. For example, if the user wants to program the Arduino board for an IoT project, the user needs to learn the C programming language. For Raspberry Pi, the user needs to know various programming languages to implement IoT applications. This situation may be feasible for an expert developer; however, any person who does not have any programming experience has to spend a lot of time to learn all the necessary programming languages. With Panther language, learning only one programming language will be enough to design and implement complex IoT applications. We analyzed different programming languages to design an efficient and straightforward domain-specific programming language. The programming languages that are used to program IoT devices are multi-purpose programming languages, and they have unnecessary operations, functions, and libraries. We removed these unnecessary operators, functions, and libraries for the IoT application development to provide a simple IoT programming language. As you can see in Listing 4.8, if the user wants to program an IoT development board to implement a distance measurement application, all the user needs to write is 10 lines of code.

```
Pin distanceSensorPin = new AnalogPin(3);
void Pinconfiguration () {
    distanceSensorPin = OUTPUT;
}
void Main() {
    Sensor distanceSensor = new EzsonarSensor();
    distanceSensor.setUpperBound = 120;
    distanceSensor.setLowerBound = 20;
    distanceSensor.add(distanceSensorPin);
}
```

Figure 4.8: Sample distance measurement sensor application in the Panther language.

We can explain the structure of the Panther language by showing the sample distance measurement code in Listing 4.8. In the sample code, there are three major areas: a pin initialization, a pin data direction specification, and a main function area in order to make the code more manageable by the user. The pin initialization area is for declaring and defining pins of selected sensors. The user might give a variable name to the pins of the target sensor, then specify a type for the corresponding pins and their orders. Let us assume that we have a *MB1000 LV-MAXSONAR-EZ0* distance measurement sensor, and this sensor has a 5V power input, ground, TX, RX, analog, and pulse-width pins. The user wants to use this sensor in the IoT application and also connect that sensor through an analog port. In the first line, the user declares the third pin of a sensor for transmitting the analog signal to the target device. In lines between 2-4, the user specifies pin's data transmission method. After this specification, the next area is the main logic area. In this, the user claims which sensor will be used in the IoT application and sets upper and lower bounds of it, then the user binds the declared pins to the sensors. We implemented the Panther language using the ANTLR parser generator [ANT], and it automatically generates the necessary compiler files for the Panther language. Then, we imported it to the PROVIZ+ main application to be used in the code-based programming tool of PROVIZ+.

### **4.2.2 Proviz Client Applications**

In addition to the main application, the client applications were developed in order to bring PROVIZ+'s features to IoT devices. Each platform has unique characteristics. So, developing platform specific client applications is required for the supported devices. In the scope of PROVIZ+, we developed Arduino and Raspberry Pi client

applications using the C and JAVA languages, respectively. These applications are responsible for responding to over-the-air programming, sensor data aggregation, and communication with the other devices. The following subsections give the detailed information about the aforementioned client applications.

### **Proviz Arduino Client Application**

In order to provide the secure over-the-air feature, we developed the *ProvizWiFi* library for Arduino IoT devices. The ESP8266 WiFi shield requires commands to configure itself, connects to wireless networks, and so on. The user may prepare required communication commands without getting any help to initiate secure over-the-air programming or can use our proposed library to automatically create the required communication commands. In addition to this manual usage opportunity, our code generation module uses the library during the secure over-the-air programming for the Arduino IoT devices. We combined the ESP8266 communication and PROVIZ+ secure over-the-air commands to unify the required two libraries into a single library. Each Arduino sketch source code file contains two main functions: setup and loop. In the setup function, the serial communication between Arduino and ESP8266 are prepared; then, PROVIZ+'s identification data from the Arduino IoT board's EEPROM storage are loaded. A string that contains unique identification and the boolean data that indicate if the board is already programmed are stored in the EEPROM. After loading the data from EEPROM, the WiFi module is checked to know if it is reachable; then, it starts to connect to the access point and the SOTA Remote Programmer that is in the PROVIZ+ main application. When the user clicks on the firmware send button in the main application, the main application sends a reboot command to the target Arduino device to let Arduino load

its bootloader address space to run the SOTA bootloader. In the reset cycle, the watchdog timer of the micro-controller is used.

After putting the micro-controller into the SOTA bootloader mode, it sends commands from the SOTA Communication Protocol to distribute the new firmware to the target devices. In summary, our contributions include the development of a skeleton application that can remotely program the micro-controller of Arduino-based IoT devices. Since the secure over-the-air feature requires the WiFi data communication, we designed and implemented the *ProvizWiFi* library that makes the code generator module's task easy for developing the Arduino IoT applications with the secure over-the-air programming capability.

### **Proviz Raspberry Pi Client Application**

Raspberry Pi 3 Model B has WiFi and Bluetooth communication capabilities without requiring any external hardware. We aimed to implement the communication manager between the Raspberry Pi devices and the main application. With this in mind, the PROVIZ+ Raspberry Pi Client Application was developed. It has four main functionalities: managing an IoT application's life-cycle, forwarding the sensor values in JSON format to the main application, handling the WiFi and Bluetooth communication stacks, and receiving a new IoT application code from the main PROVIZ+ application. Specifically, the PROVIZ+ Raspberry Pi Client works with the Linux operating system seamlessly and it consists of the sensor application, the communication manager that handles communication and management tasks between software components, and a graphical user interface application.

The relation between the components of the client can be seen in Figure 4.9. The communication manager receives the aggregated sensor data from the sensor application; then, it forwards the data to the main PROVIZ+ application. The

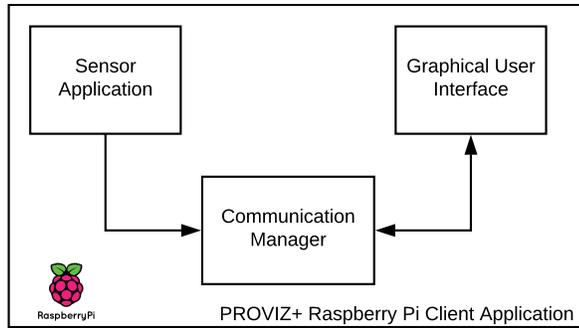


Figure 4.9: Components of the PROVIZ+ Raspberry Pi client application.

communication between the sensor application and the communication manager is done by utilizing *pipes* in Linux. Moreover, it has a GUI that has the start/stop button for the communication manager, and a debug panel to show the aggregated sensor value to the user. The main application sends a new firmware request to the communication manager for the over-the-air firmware update. The communication manager receives a new sensor application and resets the application life-cycle.

### 4.2.3 PROVIZ+ Android Tablet Application

In this thesis, we also developed the PROVIZ+ Android tablet application to extend the user experience in our ecosystem. Leaving only the PROVIZ+ main application as a data viewer is not applicable and practical because a user may have to be mobile. In order to provide another flexible platform, we chose the Android operating system and designed an Android tablet application. When a user designs the IoT application topology on the PROVIZ+ main application, the user can transfer that to the PROVIZ+ Android application. After that, the main application notifies existing IoT devices in the topology about the new data viewer application. Then, IoT devices in the topology start to send the aggregated sensor data to the Android

tablet application. The notification process of IoT devices in the topology can be followed in Figure 4.10.

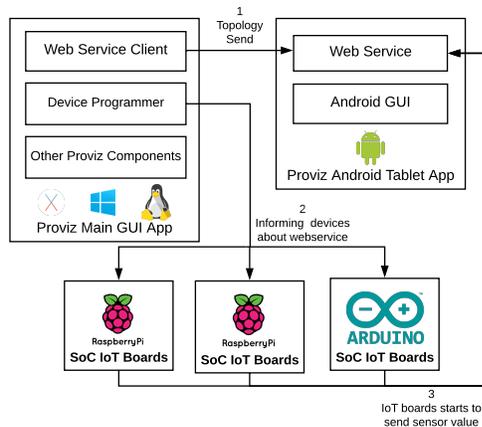


Figure 4.10: Notification of the PROVIZ+ Android tablet application to the existing IoT devices around.

The task of importing the topology and the usage of the tablet application occur as follows: First of all, the user clicks the tablet icon in the main toolbar in Figure 4.4, then this opens the topology importer wizard screen, and the user provides the IP address of the tablet. The topology importer wizard screen can be seen in Figure 4.11. After that, the system automatically enables the tablet application and it shows the topology on the tablet screen. The user might not know how to access the IP address of their tablet; hence, instead of leaving this task to the user, the PROVIZ+ Android application shows the IP address of their tablet on the screen, which is shown in Figure 4.12a. Then, the user enters the IP address into the IP address field of the topology importer screen of the main application. As soon as the connection is established with the PROVIZ+ main application, the PROVIZ+ Android application opens the topology view by loading the existing topology. The sample loaded topology on the tablet application screen is shown in Figure 4.12b. Whenever the user clicks on the board icon, a pop-up message window appears with

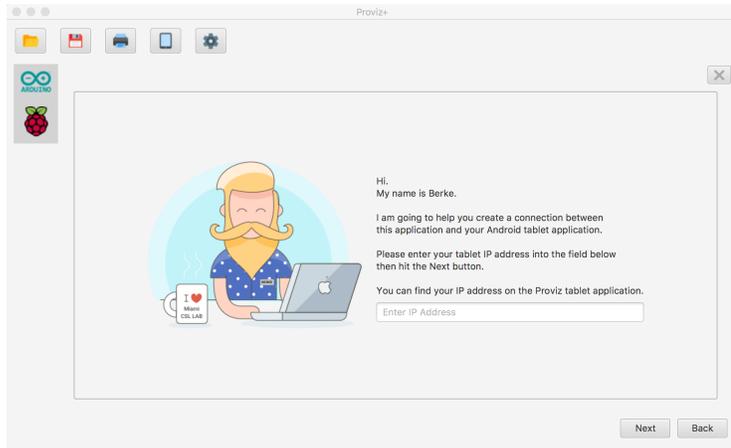


Figure 4.11: PROVIZ+ topology importer to tablet application tool.

the board information and the sensor values, which is shown seen in Figure 4.12c. In addition to the topology view, the user may monitor the IoT devices by checking them on the data table view in the PROVIZ+ Android tablet application. Here, IoT devices directly send their aggregated data to the tablet application as well as the main application. In doing so, even if the user closes the main application, the user can continue to use the tablet application. Since the entire system depends on the IP address, the user can configure a static IP address for the tablet application.

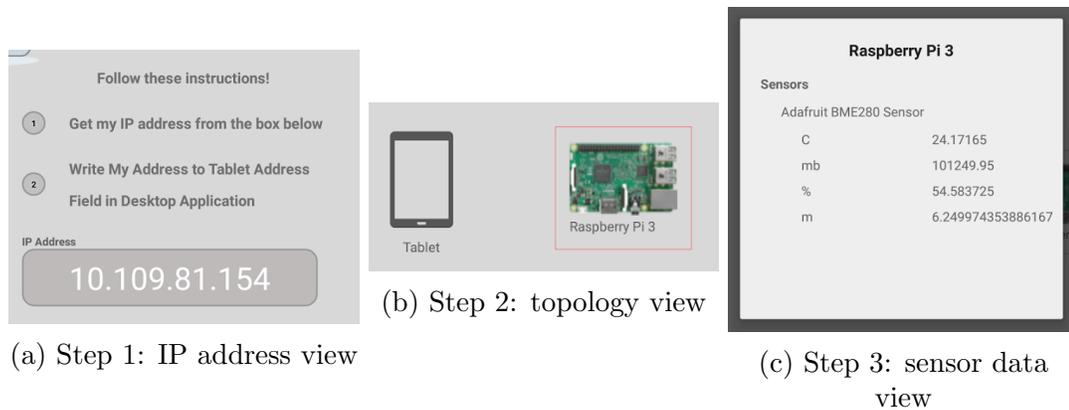


Figure 4.12: Screenshots of the PROVIZ+ Android tablet application.

## CHAPTER 5

### **SOTA: SECURE OVER-THE-AIR PROGRAMMING FRAMEWORK**

This chapter introduces a novel secure over-the-air programming framework, namely SOTA. SOTA is a part of the PROVIZ+ Software Suite to secure over-the-air programming of low-powered IoT devices by providing full data confidentiality, integration, and authentication. The rest of chapter is organized as follows: In section 5.1, we present introductory information about SOTA. Then, we propose our threat model and assumptions in the following section. The overall system design and implementation is given in section 5.3.

#### **5.1 Introduction**

A myriad of smart and interconnected IoT devices have become an integral part of our lives, scientific experiments, and military operations. The utilization of IoT devices can range from implementing a simple LED blink application to crucial applications such as monitoring military personnel’s heart rate activity. These applications require data transmission as well as remote programming capabilities to update vulnerable firmware with the latest secure firmware efficiently. This can be mostly done using a data transmission cable or over-the-air depending on usage and available resources. Since IoT devices usually have one or more low-power micro-controllers with limited computational capability, using efficient security systems to protect the IoT device’s firmware is vital while remotely programming these devices. One of the main usages of the over-the-air programming is firmware upgrade to extend the device’s capability or fixing a security vulnerability and privacy leakage. One recent incident in military is the leakage of sensitive location information of fitness tracker’s data [Ros]. In 2013, the US military gave wearable fitness trackers

to their personnel to track their activity in the scope of fighting with obesity. These wearable fitness trackers automatically share users' running routes when they are connected with a social network. This shared running routes creates heat maps that show military running routes, and it causes serious privacy concerns. To remedy this problem, an updated firmware for the wireless fitness tracker can be applied. Since a large amount of fitness trackers were distributed to military personnel, over-the-air firmware update is more feasible and efficient than physical firmware update. However, over-the-air firmware updates should be conducted with proper security precautions so as not to cause any other security concerns and should consider the resource-limited nature of the IoT devices. Indeed, secure over-the-air programming is crucial for any type of IoT application because without security, IoT devices can be reprogrammed by hackers and even firmware can be stolen by eavesdropping a live firmware distribution process. There are many further real examples of these incidents. One recent incident is the case of the Broadcom WiFi chip, which can be hijacked without any user interaction via its over-the-air programming capability [Kha]. Another example is the over-the-air update feature for modern vehicles. Many modern vehicle manufacturers have started to add over-the-air capabilities to keep their vehicles up to date without requiring any physical interaction to their vehicles. The vehicle manufacturers have implemented these features to eliminate the need for physical interaction with the vehicle. This feature can bring ease and convenience to the car owners; however, it opens the doors for potential vulnerabilities to hackers and malware. Hackers can exploit these vulnerabilities to upload ransomware. They can steal the vehicle firmware by eavesdropping during a vehicle firmware update and alter it. These types of attacks have already been seen in the real world. In 2015, the first major recall of vehicles happened due to cybersecurity concerns. Cybersecurity researchers discovered how to enter vehicle's system

by hacking Harman’s U-connect equipment on 2014 Jeep vehicles [Mea]. As these examples highlight, secure over the air programming is a vital topic. Especially, if the target environment is IoT, having limited memory, stack size, and computation power exacerbate the issues.

To address these concerns and provide a more efficient and robust solution for aforementioned problems in this thesis, a secure over-the-air programming framework called SOTA is proposed for Arduino-based IoT devices. Specifically, a symmetric encryption mechanism with AES 128 bit CBC is used to satisfy the firmware security requirements as well as providing authentication and data integrity services. The entire SOTA framework can be seen in Figure 5.1. SOTA is designed and implemented as a platform-independent, open-source, and efficient remote programmer which can simultaneously program multiple IoT devices. We also made SOTA available for the scientific community [REM] [CLI] [BOO].

**Contributions:** The main contributions are as follows:

- We designed a secure over the air programming communication protocol that supports confidentiality, integrity, and authentication for ATMEL 8 Bit microcontroller based IoT devices.
- We added a new authentication module to the original STK 500 Communication Protocol.
- We implemented over-the-air programming framework called SOTA, the boot-loader software, the over-the-air client software for ATMEL microcontrollers and the remote programmer software to program IoT devices remotely.
- We optimized existing symmetric encryption implementation with AES 128 Bit CBC algorithm in order to make it available for tiny IoT devices.



Figure 5.1: The SOTA system structure

## 5.2 Threat Model and Assumptions

In SOTA, our threat model is based on confidentiality, integrity, and authentication (CIA triad). Our solution creates end-to-end secure communication channels to transmit the firmware or any code updates to the IoT device using a symmetric encryption mechanism with AES-128 CBC algorithm. This model comes with a few assumptions. First, the private keys and initialization vectors are preloaded to the IoT devices in a secure fashion. Next, the attackers can try to eavesdrop ongoing communications between the IoT device and a benign user. Lastly, the denial-of-service attacks are outside of the scope of this work.

The following summarizes the security goals of the SOTA framework:

- Making data transmission secure and confidential: Some applications may require high level confidentiality when over-the-air-firmware updates occur. These firmware updates may contain crucial information which if compromised could pose a large threat.
- Adding authentication mechanism to eliminate any unauthorized modifications to the IoT device communications and providing data integrity to ensure that the data is not corrupted.
- Plain and tiny solution: Since many IoT devices have limited computational capabilities, lightweight operations are crucial for IoT devices. For this, in SOTA, we use Atmel ATmega2560 8 Bit micro-controller which has a 256 KB

program memory and 8 KB SRAM. If SRAM is not used efficiently and properly, possible memory leak error is inevitable. Hence, the SOTA framework should be designed with this constraint in mind. Another reason for creating plain and tiny solution is to consume less power. Applying cryptographic operations to IoT devices consume much power than regular operations, but the SOTA framework aims to mend this problem.

### 5.3 SOTA Architecture

In this section, we present the details of the architecture of the SOTA framework. As seen in Figure 5.2, SOTA consists of three major components: *IoT Device*, *IoT Device Communication Module*, and *Remote Programmer*. All of these components were designed with modularity and configurability in mind to provide the most flexible remote programming capability to the IoT devices. Also, as seen in the figure, similar components were designed for both the IoT Device and the Remote Programmer to keep the operations smooth. Finally, security (confidentiality, authentication, and integrity) was built in all of the operations. Note that the SOTA framework is designed with modular and configurable way so that any other devices or communication mechanisms can be easily integrated into the framework. In the following sub-sections, the components and important operations are articulated.

#### 5.3.1 Remote Programmer

The Remote Programmer is responsible to get the firmware from the user and send it to the IoT Device in a secure way over the air. It also supports a multi-device programming capability in a secure way by utilizing JAVA Thread library and the

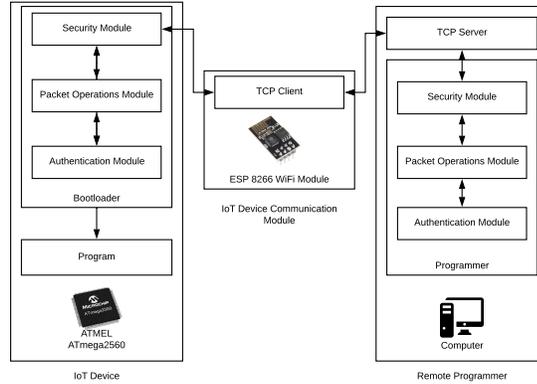


Figure 5.2: Major components of the SOTA Framework.

AES 128 bit CBC symmetric encryption algorithm [Bar16]. It is written in JAVA language to support different platforms (e.g., Linux, Mac OS, and Windows).

The Remote Programmer includes a *TCP Server* and the *Programmer* module to deliver the firmware to the target IoT Device reliably. In SOTA, the target IoT device is Arduino device with ATmega2560 micro-controller given their popularity. The TCP Server is an embedded TCP server to manage the data communication between the Remote Programmer and the IoT Device Communication Module. When the server gets new packets from the Communication Module, it directly passes them to the *Programmer* module without any modification on the packet. Also, whenever the server gets a packet from the Programmer, it directly transmits packets to the Communication Module. In SOTA, the Programmer includes the *Security Module* and the *SOTA Communication Protocol*, which is responsible for assembling the packet in the *SOTA Packet Operations Module* based on the STK500 Communication Protocol available for ATmega2560 micro-controllers. However, the default vendor-built STK500 Communication Protocol does not support any security. Hence, in SOTA's framework, confidentiality, integrity, and authentication services were also implemented to improve the security of the operations. Specifically, SOTA Packet Operations Module encapsulates STK500 Communication Protocol

packets encrypted in the Security Module with a packet starting point indicator byte (0x58) and two bytes that represent the size of the encrypted packet. For receiving commands from the IoT device, it parses the received SOTA Communication Protocol packet to extract IoT device’s encrypted STK500 communication protocol response packet. After finishing extraction of STK500 communication protocol response packet, it decrypts the received encrypted STK500 communication protocol response packet; then, it parses decrypted STK500 communication protocol response packet to extract commands of the STK500 communication protocol. The parsing operation is shown in Figure 5.3.

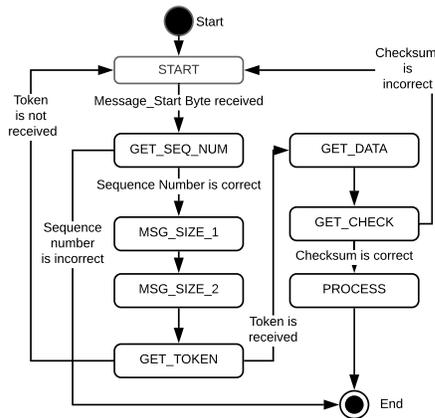


Figure 5.3: STK500 communication protocol parser.

The packet structure built upon the STK500 Communication Protocol packets is shown in Figure 5.4. As noted earlier, this new packet structure provides data integrity, confidentiality, and authentication.

The packet structure starts with 0x1B hexadecimal value to indicate the location of the starting point of the packet. Then, it continues with the sequence number field to prevent any replay attack. When the value of the sequence number field reaches the maximum value (0xFF), it zeros itself. The next two bytes represent the size of the communication commands in MSB byte order. After the size fields, 0x0E

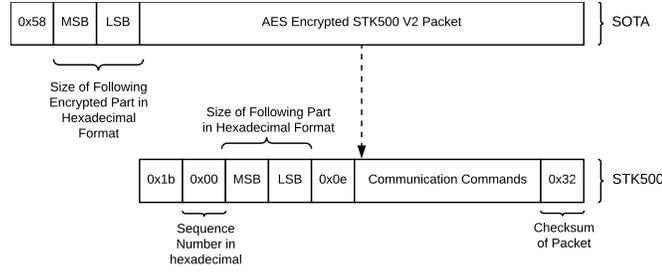


Figure 5.4: SOTA packet structure built upon the STK500 Protocol.

---

**Algorithm 1: The Remote Programmer and IoT Device programming flow.**

---

```

1 Accept request
2 Send reboot command to micro-controller
3 while reboot acknowledgment is not received do
4   | Send reboot command to micro-controller
5 Send authentication packet
6 while authentication acknowledgment is not received do
7   | Send authentication packet
8 Send synchronization packet to micro-controller
9 while synchronization acknowledgment is not received do
10  | Send synchronization packet to micro-controller
11 Send enabling program mode packet to micro-controller
12 while enabling program mode acknowledgment is not received do
13  | Send enabling program mode packet to micro-controller
14 Send load address packet to micro-controller
15 while load address acknowledgment is not received do
16  | Send load address packet to micro-controller
17 while firmware transfer is not finished do
18  | Send 256 KB firmware part to micro-controller
19  | while firmware sending acknowledgment do
20  |   | Send 256 KB firmware part to micro-controller
21 Send closing programming mode packet to micro-controller
22 while closing programming acknowledgement is not received do
23  | end closing programming mode packet to micro-controller

```

---

hexadecimal value comes into the next field as a fixed value and it is named as token. Token indicates following bytes are micro-controller operation commands such as enabling program mode or closing firmware update mode. After the token, communication commands are shown in the next fields until one byte-sized check-sum field. These communication bytes can be anything from the STK 500 Communication Protocol [ATM]. The final field is the checksum fields to provide data integrity between packet fields in each communication packet. The interaction between the Remote Programmer and the IoT Device flows are shown in Algorithm 1.

### 5.3.2 IoT Device Communication Module

In SOTA, the IoT Device Communication Module supports the WiFi connection; however, it can be easily extended by adding new connection methods. Specifically, in SOTA ESP 8266 [SHI] Wi-Fi shield is used to connect the IoT device (i.e., ATmega2560) with the Remote Programmer. ESP 8266 is a Wi-Fi module that can be connected with not only Arduino devices, but also other IoT devices. It has 1 MB of flash memory and integrated TCP/IP protocol stack as well as IEEE 802.11 b/g/n support. ESP 8266 Wi-Fi module has the ESP 8266 Wi-Fi chip [SHI] as well as 1 MB flash memory as a communication buffer. It supports two modes of communications: transparent and normal. In the transparent mode, the Wi-Fi module gets data from a sender and then transmits to the receiver without modifying communication packets; however, in the normal mode, the Wi-Fi module puts packet header that gives information about the sender and communication type. Since this packet header is not useful and inefficient during the firmware distribution, we configured the ESP 8266 Wi-Fi module to use it in the transparent mode. This makes it possible to establish transparent UART communication between the IoT Device ATmega2560 and the ESP 8266 Wi-Fi module. This communication module directly communicates between the IoT Device and the Remote Programmer and it acts as a communication bridge. It has its own TCP stack and it can serve as a TCP client so the IoT Device and the bootloader inside the device does not have to deal with the TCP protocol and this situation makes IoT device perform its tasks without sacrificing any system resources for the Wi-Fi access support.

### 5.3.3 IoT Device

IoT applications can be created using several types of customized hardware and configurations as well as IoT development devices [PRO, CUB13]. In SOTA, we consider Atmel-based Arduino IoT devices for the development platform. Note that Arduino [IoT] is a very popular IoT development device. It has a huge open-source community support. It supports a broad range of sensors and modules to develop complex IoT applications and has highly customizable features. Furthermore, a large majority of Arduino models have the same brand micro-controller, Atmel [ATW]. Having the same micro-controller makes it possible to develop common solutions for many IoT products without needing to differentiate by the type of the Arduino device. As noted earlier, In SOTA, the Arduino Mega 2560 is selected as an IoT device to utilize these benefits, but the SOTA framework can be applied to other IoT development devices.

The IoT Device in SOTA includes two sub-components in its architecture: *Bootloader* and *Program Space* to load and run the firmware. The Bootloader is the first loading part when the IoT Device turns on. If it does not get any communication protocol commands from the remote programmer, it basically loads the IoT application software from the Program Space. If received commands comply with the communication protocol rules and are in command set of the communication protocol, the Bootloader does special operations on the device such as firmware update, erasing flash memory, modifying hardware signature, and so on.

The Program Space stores machine instructions of the IoT application. In other words, the developer's code or the distributed firmware is stored in the Program Space. The Program Space should contain rebooting mechanism to reach Bootloader by the Remote Programmer. When a user wants to reprogram the IoT Device, s/he will need to reboot the IoT Device to reach Bootloader. The rebooting

process enables accessing Bootloader from the Remote Programmer. In the SOTA framework, we developed a skeleton code code [CLI] to bring smooth self-rebooting capability into any IoT application software. It has a watchdog supported reboot mechanism; the watchdog is basically a native Atmel micro-controller function and it requires *avr/wdt.h* library to be compiled without any errors. The watchdog is a hardware timer that resets the micro-controller unless it receives a watchdog reset signal from the user's code. The aforementioned micro-controller reset function that we implemented can be seen in Listing 5.5. The user needs to implement the user's IoT application code into the given skeleton code [CLI].

```
void triggerWDTReset ()
{
    wifi .informServerForReset ();
    wdt .enable (WDTO_15MS);
}
```

Figure 5.5: Watchdog supported micro-controller reset function.

Moreover, similar to the Remote Programmer, the Bootloader has the SOTA Communication Protocol, Packet Operations, Authentication, and Security Modules to handle the firmware packets that are sent from the Programmer. The Communication Protocol gets decrypted packets from the Security Module or it sends packets to the Security Module in order to put encrypted packet back to the Remote Programmer.

### 5.3.4 Provision of Security

As noted earlier, one of the novel components of the SOTA framework is the provision of security via confidentiality, integrity, and authentication for the over-the-air programming.

In an IoT development environment, there are many limitations that stem from the small amount of memory size and limited computational resources. In our case, Atmega2560 comes with 8 Bit microprocessor with 256 KB self-programmable flash, 8 KB SRAM, and 4 KB EEPROM [MIC] which are smaller than the resources of regular desktop computing environments. These limitations make advanced encryption algorithms not viable choices with Atmega2560-based IoT devices without applying any optimization. Implementing encryption algorithms on this resource-limited IoT environment requires a great deal of attention and optimization of the microcontroller's resources. In our design, we considered the Security Module to work with the limited resources in the IoT device.

Hence, SOTA runs encryption and decryption process by utilizing a symmetric encryption algorithm with AES 128 Bit CBC algorithm, which needs relatively less advanced microprocessor and can operate well with the limited RAM and CPU. For this, in the Security Module, a minimized version of AES-128 bit symmetric-key encryption library for the Atmel Mega 2560 microcontroller [AES] was optimized by placing arrays in the flash memory of the micro-controller instead of placing RAM due to memory size concerns and used in order to satisfy the security expectations with the limited resources. The minimized version of AES-128bit encryption reduces the amount of stack size used and occupies a small amount of microprocessor's time.

Moreover, authentication in SOTA is provided to eliminate any unauthorized modifications to the firmware on the IoT Device. With authentication, it is guaranteed that the Remote Programmer can not do anything on the IoT Device before authorization. We develop the authentication mechanism in the scope of SOTA, and the authentication is constructed between pairs by applying the same process in both ways of the communication. In the remote programmer's authentication, the Remote Programmer applies authentication procedure to IoT devices to gain

the target device's trust. After finishing the remote programmer's authentication, IoT devices follow the same the authentication procedure to assure that both are legitimate. The Remote Programmer and Bootloader have the same embedded 4 byte authentication token and 4 byte secret key to use during the authentication procedure that consists of two parts. Both keys are physically preloaded to pairs. This authentication procedure that is initiated by the Remote Programmer to authenticate future communications with the IoT device runs as follows: The Remote Programmer puts the authentication token next to a 4 byte number randomly generated number; then, it encrypts and sends the authentication packet to the IoT device to process it. When the the device receives the authentication packet from the Remote Programmer, it decrypts to extract the authentication token and a random generated number. After the completion of the decryption, it splits them into two different parts: the authentication token and four byte random number. The authentication token informs the IoT device that the Remote Programmer is legitimate and then the device starts to trust the source. After finishing it, the second part of the authentication procedure is started. The steps of the second part of the authentication procedure can be seen in Figure 5.6.

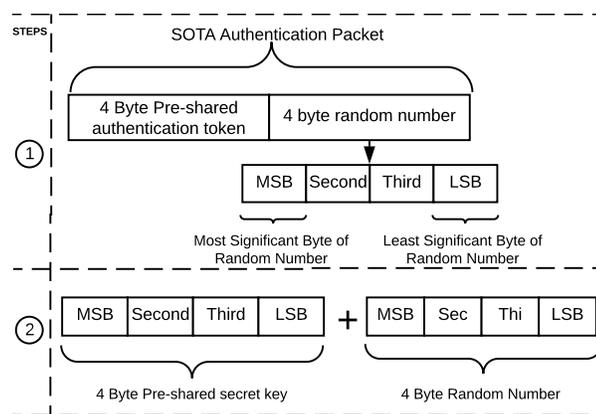


Figure 5.6: Steps of the authentication process.

When Bootloader passes the first part of the authentication mechanism, it starts to parse the next four byte number into a new 4 byte sized byte array (Step 1). Then, it adds to 4 byte secret key that is loaded during the first physical firmware upload process, to generate a new number (Step 2). Since the same algorithm is known by the Remote Programmer, the device generates the same number as well. The IoT Device sends this new generated number in a response packet to the Remote Programmer. Meanwhile, the Remote Programmer applies the same steps to find the new 4 byte random number to compare it with device's generated random number as shown in Figure 5.6. Once the Remote Programmer sees that the received number matches with the Remote Programmer generated number, it completes the remote programmer's part of authentication mechanism; then, it starts same procedure again from the IoT device's perspective. After finishing the authentication mechanism, the IoT device and the Remote Programmer trust each other to accept further packets. When the authentication between the remote programmer and the IoT device are finished successfully, Bootloader starts to accept other commands of the STK500 Communication Protocol to make changes on the IoT Device. After the establishment of authentication between the device and the Remote Programmer, both ends are synced and the programming of the device is possible.

## CHAPTER 6

### PERFORMANCE EVALUATION

In this chapter, we present the performance evaluations of the PROVIZ+ software suite and the SOTA framework. Section 6.1 introduces the testbed and experiment methodologies; and the subsequent sections examine the performance in further detail with different metrics.

#### 6.1 Testbed and Methodology

Since PROVIZ+ has different software modules, a multi-dimensional performance analysis is optimal and needed for the evaluation of PROVIZ+. The experiments are designed to measure the programming time of IoT devices and average system resource utilization. For both the main and client applications, the resource utilizations were monitored in real-time using the native resource monitor of each application’s respective operating system.

The main application was run on an Apple Macbook Pro using Oracle JRE 9.0.4 [JVM]. Then, we used Raspberry Pi 3 Model B and Arduino Mega 2560 in experiments of the PROVIZ+ Raspberry Pi and Arduino client applications. Moreover, Samsung Galaxy Tab A tablet was used to measure the performance of the PROVIZ+ tablet application. These devices’ specifications can be seen in Table 6.1.

Table 6.1: Specification of devices that were used in the experiments.

Device Name	CPU	Memory(RAM)	Disk Speed (write)
Apple Macbook Pro	2.4 GHz Intel i5	8 Gb	254.4 MB/s
Raspberry Pi 3 Model B	1.2 GHz Quad-Core ARM Cortex A53	1 GB	20 MB/s
Arduino Mega 2560	ATMEL Mega 2560	8 Kb SRAM	instant
Samsung Galaxy Tab A tablet	1.2 GHz Quad-Core, Qualcomm APQ 8016	1.5GB RAM,	52 MB/s

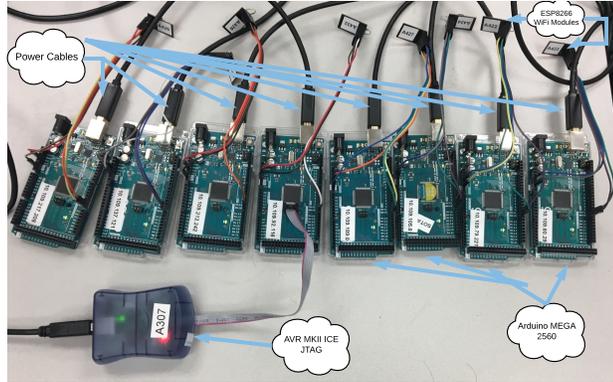


Figure 6.1: The SOTA Framework performance analysis testbed.

As noted, in addition to the PROVIZ+ performance evaluation, we conducted experiments for the SOTA framework, and designed a testbed to evaluate the framework accurately. As noted earlier, although the SOTA framework is designed with flexibility to work with other devices, it is built utilizing ATmega2560-based IoT Devices. The testbed to evaluate the SOTA framework is shown in Figure 6.1. Specifically, in the SOTA evaluation, we investigated four different experimental setups to analyze the performance of the SOTA framework. These experimental setups can be seen in Figure 6.2.

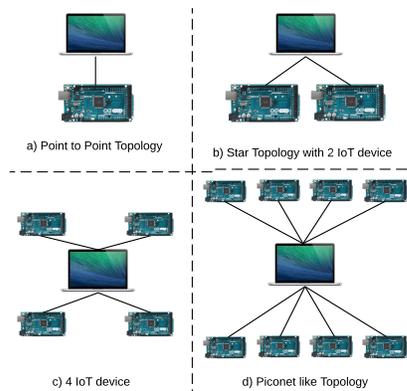


Figure 6.2: Experimental setups: 4 different topologies studied.

The first experimental setup, Topology-A, is a point-to-point network topology with one IoT device. However, the point-to-point topology is not a common network

topology for IoT devices and applications; so, in order to have a more realistic experimental setup, we added the star topology with two (Topology-B), four (Topology-C), and eight (Topology-D) different IoT devices into our experimental set. Topology-D with 8 devices was inspired from the Piconet structure of the Bluetooth protocol. Although Topology-D involves 8 devices, the testbed can be extended with more IoT devices. In performance evaluations, we aimed to investigate the behaviour and the performance of the SOTA framework in different settings.

## **6.2 Results**

In this section, we conducted experiments on the PROVIZ+ software suite and we discuss the results to present the performance of PROVIZ+. After performance evaluation of PROVIZ+, we continued our experiments by conducting the performance evaluation for our secure over-the-air programming framework. In our evaluations, we focused on three different analyses. In the first one, we analyzed the completion time of tasks. In the second analysis, the CPU, memory, and storage performance of PROVIZ+ and SOTA were examined; and in the third, the security analysis was performed. In these analyses, quantitative and qualitative data are presented to support the functionality and reliability of PROVIZ+ and SOTA.

### **6.2.1 Time Analysis**

As noted earlier, most of PROVIZ+'s software components were developed on top of the JAVA run-time environment (JVM) to make PROVIZ+ available for a broad range of platforms and operating systems. And, PROVIZ+ has some components that may require some native hardware support such as Bluetooth and WiFi communication as well as the secure over-the-air programming features. Implementing

the native support functions in the JAVA language can be difficult if performance is essential, and it could generate undesired latency. These features may create additional overhead to PROVIZ+. Therefore, we used multi-process computation so as to keep the PROVIZ+ software suite steady and quick to avoid any additional overhead. Table 6.2 shows the completion time of tasks.

Table 6.2: The PROVIZ+ main application task completion time statistics.

Task	Average Time (ms)	$\sigma$	CI (95%)
PROVIZ+ Main Application Opening	2455.91	439.77	145.69
Topology Transfer to Android App	198.17	103.59	34.32
Bluetooth OTA for Arduino Mega 2560	49473.14	13821.01	4747.68
WiFi OTA for Raspberry Pi 3	1827.17	575.42	197.66

As noted earlier, the performance evaluation of the PROVIZ+ main application was measured using an Apple Macbook computer which had 2.4 GHz i5 processor, 8 GB RAM, and 256 GB SSD flash storage. The experiment values that are in Table 6.2 were obtained by taking a mean of 35 experiment runs. Then, the variance and confidence interval (% 95) of the experiment values were calculated to present a distribution of experiment values. Since PROVIZ+ has native function calls for the Bluetooth and WiFi communication, the native function calls affect the performance of PROVIZ+ negatively. The values in the Table 6.2 could be lower if a more efficient JAVA Run-time Environment is selected.

In addition to the task completion performance analysis of PROVIZ+, we performed the same analysis for the secure over-the-air programming states in the SOTA framework. Analyzing a micro-controllers in the simulation environments create inaccurate results for real life scenarios. Furthermore, network availability may generate delays for the communication packets. In this thesis, in order to remedy aforementioned concerns, we implemented the Remote Programmer, the client application, and the bootloader on real IoT devices and selected our university ac-

cess point to get the average packet delay. The time analysis of the SOTA framework can be seen in Table 6.3.

Table 6.3: The SOTA Framework experiment results.

Topology	Value Type	Authentication	Closing Firmware Mode	Reboot Procedure	Sending Firmware Packet	Synchronization	Overall OTA
A	Time ( $\mu$ )	3099.11	2097.51	5383.11	46274.11	2184	84466.54
	$\sigma$	999.66	594.33	55.74	13125.83	52.96	5143.68
	$SE_{\bar{x}}$	171.44	100.46	9.55	2218.67	9.08	869.44
	CI (95%)	336.01	133.16	19.42	2767.89	18.45	1704.10
B	Time ( $\mu$ )	3326.24	2168.97	5471.98	48026	2177.85	88375.48
	$\sigma$	1011.89	89.85	584.91	749.78	69.49	8604.07
	$SE_{\bar{x}}$	121.81	10.81	70.41	90.26	8.36	1475.58
	CI (95%)	238.76	21.58	140.47	180.07	16.68	2892.15
C	Time ( $\mu$ )	3614.15	2159.68	5518.46	48054.02	2188.71	94290.85
	$\sigma$	1042.65	70.54	768.89	597.06	64.08	11216.99
	$SE_{\bar{x}}$	88.43	5.98	65.21	50.64	5.43	1923.69
	CI (95%)	173.33	11.82	128.94	100.12	10.74	3770.44
D	Time ( $\mu$ )	4423.72	2155.61	5611.63	47992.41	2182.89	115329.65
	$\sigma$	993.61	55.82	974.04	505.04	69.14	27258.99
	$SE_{\bar{x}}$	59.48	3.34	58.31	30.23	4.13	4674.87
	CI (95%)	116.59	6.57	114.79	59.52	8.14	9162.76

Accordingly, we ran the SOTA framework to program the Arduino-based IoT devices over-the-air 35 times and traced all the steps of the over-the-air programming process to get their completion time from the Remote Programmer’s perspective. Because the Arduino-based IoT device does not have enough computational resources to track itself while over-the-air is running, we did not track anything on the device side. The time analysis of the secure over-the-air programming consists of six major parts: authentication, closing firmware mode, reboot procedure, synchronization, sending firmware packets’ part, and finally overall over-the-air completion time. The results are presented in Table 6.3 to show the performance of the SOTA framework. Since we implemented a parallel over-the-air programming process by utilizing the JAVA thread library, our experiment results do not grow linearly due to the multi-threaded programming. As observed in Table 6.3, differences between values for different topologies are not high, and the confidence interval is relatively high in the first topology, thereafter continually decreasing. The confidence interval is continuously decreasing with more devices in the experiment except the overall

over-the-air completion time. The closing firmware task includes two variable assignments in the bootloader and therefore can give a fast response to the Remote Programmer without creating any visibly big difference among the different experiment topologies. This minor difference can be seen in Table 6.3. Besides these experiments, we traced the overall over-the-air completion time. These measurements started with the running time of the over-the-air application and terminated itself as soon as it got the closing program mode acknowledgment from the last IoT device in the experimental setups. The results of these experiments can also be seen in Table 6.3. As seen in the table, there is a non-linear increment among different experiment topologies. As expected, as the number of devices in the topology increase, so does the overall completion time. Moreover, the confidence interval changes according to the number of total devices in the experiments because each device contributes extra overhead to the total over-the-air completion time.

### 6.2.2 CPU, Memory, and Storage Analysis

In addition to measuring task completion time, we monitored the PROVIZ+ software suite and the SOTA framework to provide system usage and requirements information about them. As noted earlier, we used devices that are in Table 6.1 in CPU, memory, and storage analysis for components of PROVIZ+ software suite. Table 6.4 shows the average system resource utilization of the PROVIZ+ software suite.

Table 6.4: The PROVIZ+ Software Suite resource usage statistics.

Application Name	CPU	RAM	File Size
The main application	15.3%	235 MB (2.83%)	18.9 MB
The Arduino client application	-	911 Byte (11%)	10942 Byte (4%)
The Raspberry Pi client application	3%	54.3 MB (5.3%)	58.3 MB
The Android tablet application	5%	41 MB (4%)	28.3 MB

Table 6.4 indicates that the main application uses CPU time percentage more than the usage percentage of RAM. Since we provide real-time communication with IoT devices, the remote programmer runs software threads to manage the data transmission. The main application contains two web servers for importing the topology to the Android tablet application and aggregating the sensor data from IoT devices. Therefore, having two separate web servers generates 15% CPU usage in the main application. The client applications use less than 15% RAM, which indicates that IoT devices can run another application alongside the PROVIZ+ client applications. The Android tablet application runs software threads for each IoT device in the topology like the main application, which leads to optimal system resource usage for the Android tablet application.

Another contribution of this thesis, the SOTA framework, works on an Atmel-based IoT device with a tiny micro-controller which has limited memory and computational power. Table 6.5 shows memory footprint on Arduino-based IoT devices. Table 6.5: Memory Occupation of the STK500 Bootloader vs. the SOTA Bootloader.

Bootloader Type	Program Size (byte)	Data Size (byte)
Original STK500 Protocol	2080 (0.8% full)	6 (0.1% full)
SOTA Bootloader	5666 (2.2%full)	1591 (19.4%full)

As seen in Table 6.5, the program size difference is very negligible; however, the data occupation of the new SOTA framework is higher than the original STK500 Bootloader. The AES algorithm and authentication mechanisms need to allocate arrays to run the security necessary operations, and this increases the data sizes of the SOTA Bootloader.

### 6.2.3 Security Analysis

Since one of our main contributions is providing security in the IoT applications that are developed by the PROVIZ+ software suite, we focused on security during the development of PROVIZ+ and analyzed the software suite regarding the security concerns. In the main application of PROVIZ+, we generated a universally unique identifier for each IoT devices in the user's topology. These devices accept commands from the source that carries a device's universally unique identifier and simply ignore packets from any other sources. Moreover, we used standard Bluetooth security precautions to make data transmission and over-the-air programming secure. Bluetooth communication protocol runs data encryption to hide the content of ongoing packets. Furthermore, in WiFi connection of the PROVIZ+ Raspberry Pi client application, passwordless secure file copy and secure shell (SSH) were used during the firmware upgrade tasks. In addition to the security mechanism of the over-the-air feature of PROVIZ+ Raspberry Pi client application, Secure Hypertext Transfer Protocol (SHTTP) with a self-signed Secure Sockets Layer (SSL) certificate is used to communicate with the main application without exposing the data in clear. The same protocol was used in communication between the PROVIZ+ main application and the PROVIZ+ Android tablet application.

We also designed and developed a security mechanism for the SOTA framework to protect firmware over-the-air. An attacker can eavesdrop programming packets and get firmware data, and this firmware may be compromised or altered. A compromised firmware may lead to serious consequences such as showing inaccurate data in vital IoT applications. Hence, SOTA uses a symmetric encryption with AES-128 bit CBC to provide confidentiality and integrity for the data packets. We assume keys are preloaded. Note that there is no known attack for AES-encrypted data without the private key. Indeed, according to NIST, AES encryption is acceptable with 128

bit key through 2030 and beyond [Bar16]. It can be applicable to unclassified data and SECRET level data in government communications [oST17, CSS17]. In addition to this, AES encryption complies with ISO/IEC 18033 security standard. Given the resource-limited nature of the Atmel-based IoT devices, AES is a viable solution to provide security to the over-the-air-programming of the IoT devices. Hence, in SOTA, we primarily provide confidentiality by encrypting ongoing communication traffic with an AES symmetric algorithm. We also put additional random bytes at the end of communication packet before encryption in order to provide full confidentiality. Another important concept that we provide in the SOTA framework is integrity, and we provide it with our sequence number and check-sum field that is in the packet structure of encrypted the STK 500 communication protocol, and authentication with the mutual firmware update authorization mechanism. Since we have limited resources in IoT devices, we have to choose unauthenticated AES CBC symmetric encryption instead of authenticated AES symmetric encryption algorithm; however, we reinforced our protocol by developing and implementing a new authentication mechanism. To remedy a possible replay attack, we designed the authentication mechanism as a two way authorization process based on a randomly generated authentication number, an authentication token, and a preloaded secret key. Even if an attacker wants to use a previously recorded session packets to inhibit updated secure firmware, the authentication mechanism does not allow the use of any previous session's over-the-air programming packets thanks to this random authentication number control.

## CHAPTER 7

### CONCLUSION

In this chapter, we conclude the thesis regarding the PROVIZ+ software suite. We also discuss benefits of PROVIZ+ in Section 7.1. Then, any potential future work is presented in Section 7.2 to improve this thesis.

#### 7.1 Benefits of the PROVIZ+ Software Suite

PROVIZ+ contains several software modules: a Bluetooth, a WiFi, and a serial communication modules, the Panther language compiler, and the code generator. PROVIZ+ can program IoT devices to develop sophisticated IoT applications without requiring any hardware or software developmental experience using either the code-based programming tool or the visual programming tool. In the code-based programming, a user can write a code in the Panther language to program IoT devices. Using the Panther language helps a user by saving time from learning various programming language for different IoT devices. Moreover, the user can develop IoT applications using the visual programming module instead of the code-based programming module. In doing so, the users do not need to write code in the Panther language, they can simply select the target board and the sensors by dragging and dropping onto the topology canvas. PROVIZ+ has an Android tablet application support, so the user can download the PROVIZ+ Android tablet client to track aggregated sensor data from the current topology. Even if the user closes the PROVIZ+ main application, the Android tablet client application can independently get data from the sensors. Finally, the PROVIZ+ software suite includes the secure over-the-air programming framework to provide programming flexibility to the user, and this framework has the following four contributions: programming the

Atmel ATmega2560 micro-controllers over-the-air, providing data confidentiality, integrity, and authentication in the over-the-air firmware programming, cross-platform support for the remote programmer, and the SOTA bootloader for micro-controllers in IoT devices.

## 7.2 Conclusions and Future Work

A proliferation of IoT devices has brought the need for a single developmental platform for programming IoT devices. Traditionally, a user has to learn various IoT application development environments, programming languages, and fundamentals of hardware design. However, with an IoT application development environment like PROVIZ+, the user can develop sophisticated IoT applications without having experience in software and hardware development. In addition to the development of IoT devices, the secure over-the-air programming requires experience in the central processing units (CPU) and the security of IoT devices. Since most of IoT devices and applications are limited in terms of energy sources, computational power, and usable memory, energy-efficient secure over-the-air code dissemination is vital in IoT devices.

In summary, in this thesis, we propose the PROVIZ+ software suite that consists of the main application, the Arduino and Raspberry Pi client applications, the Android Tablet application, and the secure over-the-air programming framework for IoT devices. The SOTA framework, is designed and implemented as a platform-independent, open-source, and efficient over-the-air programming framework for programming multiple IoT devices simultaneously, which is suitable for the Atmel-based IoT devices. We also evaluated the performance of the PROVIZ+ software suite and SOTA. Our experiments revealed that PROVIZ+ can be used in

the development of IoT application without any security and usability concerns, and the SOTA framework yields minimum overhead for an over-the-air programming of IoT devices and can be efficiently utilized while providing confidentiality, integrity, and authentication.

For future work, we are planning to conduct our experiments on more devices and also to implement new features for recovering the over-the-air programming from any process failure. Furthermore, PROVIZ+ can be improved by adding new IoT development boards and sensors to its library. Since it supports sensor and board addition to the PROVIZ+ software suite, the user can add their desired sensor and this database can be shared with other users in order to eliminate double addition problems in the library. Also, PROVIZ+ is a significant tool for people who do not have any programming or hardware experience because it supports a visual programming tool that is easy to use for everyone. For example, K12 students can benefit from it. The entire PROVIZ+ project can be re-designed for high school students to teach them about IoT devices, development boards, and sensors. The students may be able to design their IoT application using the PROVIZ+ software suite. Finally, the Bluetooth over-the-air programming can also be considered as a new feature for the Raspberry Pi IoT devices.

## BIBLIOGRAPHY

- [ABB<sup>+</sup>12] N. Aschenbruck, J. Bauer, J. Bieling, A. Bothe, and M. Schwamborn. Selective and secure over-the-air programming for wireless sensor networks. In *21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, 2012.
- [AES] Tiny aes-128 bit library. <https://github.com/kokke/tiny-AES128-C>. Accessed: 04-18-2018.
- [AMA] Cloud iot core. <https://aws.amazon.com/iot-core/>. Accessed: 06-05-2018.
- [ANT] Antlr powerful parser generator. <http://www.antlr.org>. Accessed: 01-05-2018.
- [APA] What is apache freemarker? <https://freemarker.apache.org/>. Accessed: 05-31-2018.
- [APP] Homekit - apple developer. <https://developer.apple.com/homekit/>. Accessed: 06-05-2018.
- [ARD] Arduino mega tutorial pinout schematics. <https://www.circuitstoday.com/arduino-mega-pinout-schematics>. Accessed: 06-05-2018.
- [ATM] ATMEL. Avr stk500 datasheet. <http://www.atmel.com/images/doc2591.pdf>. Accessed: 05-05-2018.
- [ATW] Atmel. <http://www.atmel.com>. Accessed: 04-04-2018.
- [Bar16] Elaine Barker. Sp 800-57 part 1 rev. 4, recommendation for key management, part 1: General. *NIST special publication*, 800:57, 2016.
- [BLU] Bluefruit ez-link - bluetooth serial link arduino programmer. <https://www.adafruit.com/product/1588>. Accessed: 03-05-2018.
- [BOO] Sota bootloader for atmel micro-controllers. <https://github.com/cslfiu/SOTA-ATMEL-Bootloader-Application>. Accessed: 05-30-2018.

- [CBS<sup>+</sup>18] Z Berkay Celik, Leonardo Babun, Amit K Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. *arXiv preprint arXiv:1802.08307*, 2018.
- [CGOF13] Pablo Carreño, Francisco Gutierrez, Sergio F. Ochoa, and Giancarlo Fortino. Using human-centric wireless sensor networks to support personal security. In Mukaddim Pathan, Guiyi Wei, and Giancarlo Fortino, editors, *Internet and Distributed Computing Systems*, pages 51–64, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [CLI] The sota client skeleton code. <https://github.com/cslfiu/SOTA-Skeleton-Code-for-Client>. Accessed: 05-30-2018.
- [CLZ06] Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos: A graphical development simulation environment for tinyos-based wireless sensor networks. Technical Report 2, EECS Department, University of California, Berkeley, The address of the publisher, 7 2006.
- [CRO] CROSSBOW. Micaz. [http://www.cmt-gmbh.de/Produkte/WirelessSensorNetworks/Datenblaetter/MICAz\\_Kit\\_Datasheet.pdf](http://www.cmt-gmbh.de/Produkte/WirelessSensorNetworks/Datenblaetter/MICAz_Kit_Datasheet.pdf). Accessed: 04-04-2018.
- [CSS17] *National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information*. CNSS Policy No. 15, Fact Sheet No. 1. National Institute of Standards and Technology, 2017.
- [CUB13] Ramalingam K Chandrasekar, A Selcuk Uluagac, and Raheem Beyah. Proviz: An integrated visualization and programming framework for wsns. In *Local Computer Networks Workshops (LCN Workshops), 2013 IEEE 38th Conference on*, pages 146–149. IEEE, 2013.
- [DEDP15] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller. A survey of commercial frameworks for the internet of things. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, Sept 2015.
- [DHM05] J. Deng, R. Han, and S. Mishra. Secure code distribution in dynamically programmable wireless sensor networks ; cu-cs-1000-05. Technical report, University of Colorado, 2005.

- [ESP] Esp8266 soc wifi module. <https://www.sparkfun.com/products/13678>. Accessed: 03-05-2018.
- [GOO] Cloud iot core. <https://cloud.google.com/iot-core/>. Accessed: 06-05-2018.
- [HST08] Andrew Hagedorn, David Starobinski, and Ari Trachtenberg. Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks, IPSN '08*, pages 457–466. IEEE Computer Society, 2008.
- [IBM] Watson iot platform. <https://www.ibm.com/cloud/watson-iot-platform>. Accessed: 06-05-2018.
- [IoT] Arduino. <https://www.arduino.cc/en/Guide/Introduction>. Accessed: 04-05-2018.
- [ISR+11] Muhammad Sabir Idrees, Hendrik Schweppe, Yves Roudier, Marko Wolf, Dirk Scheuermann, and Olaf Henniger. *Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates*, pages 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [JAV] Java swing library bug. <https://bugs.openjdk.java.net/browse/JDK-8187367>. Accessed: 04-12-2018.
- [JVM] Overview of jdk 9 and jre 9 installation. <https://docs.oracle.com/javase/9/install/overview-jdk-9-and-jre-9-installation.htm> JSJIG-GUID-8677A77F-231A-40F7-98B9-1FD0B48C346A. Accessed: 05-31-2018.
- [Kha] Swati Khandelwal. Millions of smartphones using broadcom wi-fi chip can be hacked over-the-air. <http://thehackernews.com/2017/04/broadcom-wifi-hack.html>. Accessed: 04-21-2018.
- [KW03] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: attacks and countermeasures. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*, pages 113–127, May 2003.
- [KW05] S. S. Kulkarni and Limin Wang. Mnp: Multihop network reprogramming service for sensor networks. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 7–16, June 2005.

- [LGN06] Patrick E. Lanigan, Rajeev Gandhi, and Priya Narasimhan. Disseminating code updates in sensor networks: Survey of protocols and security issues. In *International Journal of Engineering Research & Technology (IJERT)*, 2006.
- [LK] Sokjoon Lee and S. Korea. Hash-based secure sensor network programming method without public key cryptography.
- [Mea] Lucas Mearian. Update: Chrysler recalls 1.4m vehicles after jeep hack. <http://www.computerworld.com/article/2952186/mobile-security/chrysler-recalls-14m-vehicles-after-jeep-hack.html>. Accessed: 05-05-2018.
- [MIC] Atmel mega 2560. [http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561\\_datasheet.pdf](http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf). Accessed: 04-04-2018.
- [MSPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.
- [NL08] D. K. Nilsson and U. E. Larson. Secure firmware updates over the air in intelligent vehicles. In *ICC Workshops - 2008 IEEE International Conference on Communications Workshops*, pages 380–384, May 2008.
- [NSN08] D. K. Nilsson, L. Sun, and T. Nakajima. A framework for self-verification of firmware updates over the air in vehicle ecus. In *2008 IEEE Globecom Workshops*, pages 1–5, Nov 2008.
- [NSV<sup>+</sup>13] S. Nastic, S. Sehic, M. Vgler, H. L. Truong, and S. Dustdar. Patricia – a novel programming model for iot applications on cloud platforms. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 53–60, Dec 2013.
- [oST17] National Institute of Standards and Technology. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. Federal Information Processing Standards Publication 197. National Institute of Standards and Technology, 2017.
- [PRO] Proviz+ project website. <https://www.proviz-project.net>. Accessed: 05-30-2018.

- [PSS00] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim: A simulation framework for sensor networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '00, pages 104–111, New York, NY, USA, 2000. ACM.
- [RASa] Raspberry pi 3 model b. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>. Accessed: 06-05-2018.
- [RASb] This \$35 computer just passed a major sales milestone. <http://fortune.com/2016/09/08/raspberry-pi-10-million/>. Accessed: 06-05-2018.
- [RCUB16] Shruthi Ravichandran, Ramalingam K. Chandrasekar, A. Selcuk Uluagac, and Raheem Beyah. A simple visualization and programming framework for wireless sensor networks: Proviz. *Ad Hoc Networks*, 53:1–16, 2016.
- [REM] The remote programmer source code. <https://github.com/cslfiu/SOTA-The-Remote-Programmer>. Accessed: 05-30-2018.
- [RL03] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 60–67. ACM, 2003.
- [Ros] Matthew Rosenberg. Strava fitness app can reveal military sites, analysts say. <https://www.nytimes.com/2018/01/29/world/middleeast/strava-heat-map.html>. Accessed: 05-31-2018.
- [RSUB12] KC Ramalingam, Venkatachalam Subramanian, A Selcuk Uluagac, and Raheem Beyah. Simage: Secure and link-quality cognizant image distribution for wireless sensor networks. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 616–621. IEEE, 2012.
- [RW12] Matt Richardson and Shawn Wallace. *Getting started with raspberry PI.* " O'Reilly Media, Inc.", 2012.
- [SHI] Espressif 8266. <https://espressif.com/en/products/hardware/esp8266ex/overview>. Accessed: 04-04-2018.

- [SPBS13] Jaroslav Sobota, Roman Pil, Pavel Balda, and Milo Schlegel. Raspberry pi and arduino boards in control education. *IFAC Proceedings Volumes*, 46(17):7 – 12, 2013. 10th IFAC Symposium Advances in Control Education.
- [Sys] Adafruit Learning Systems. Adafruit bme280 humidity+barometric pressure+temperature sensor breakout. <http://www.mouser.com/ds/2/737/adafruit-bme280-humidity-barometric-pressure-tempe-740823.pdf>. Accessed: 05-12-2018.
- [THI] Thingworx platform product brief. <https://www.ptc.com/en/resources/iot/product-brief/thingworx-platform>. Accessed: 06-05-2018.
- [TIN] Tinyos documentation wiki. [http://tinyos.stanford.edu/tinyos-wiki/index.php/Main\\_Page](http://tinyos.stanford.edu/tinyos-wiki/index.php/Main_Page). Accessed: 06-05-2018.
- [VIE05] F.L.M. VIEIRA. Wisdom: A visual development framework for multi-platform wireless sensor networks. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference*. IEEE, 2005.