Microcontroller Based IoT System Firmware Security: Case Studies

Chao Gao‡, Lan Luo†, Yue Zhang‡, Bryan Pearson†, Xinwen Fu†

‡Department of Computer Science, University of Massachusetts Lowell, MA, USA. Email: cgao@cs.uml.edu
†Department of Computer Science, University of Central Florida, FL, USA. Email: {lukachan,bpearson}@knights.ucf.edu, xinwenfu@ucf.edu
‡Department of Computer Science, Jinan University, Guangzhou, China. Email: zyueinfosec@gmail.com

Abstract—The Internet of Things (IoT) has attracted much interest recently from the industry given its flexibility, convenience and smartness. However, security issues and exploits have become amongst the most colossal concerns for IoT. This paper studies the security of Microcontroller (MCU) based IoT firmware. Given the varieties of MCUs and their running environments, we perform case studies to exploit the flaws behind contemporary firmware upgrade models. Specifically, we validate our attacks on a popular air quality sensor from PurpleAir. We also investigate a prototype of a secure firmware upgrade system on an ATmega1284P chip. To demonstrate the attack surface of the implemented countermeasure, we discuss the potential pitfalls identified through our own practice, since these pitfalls may occur during the implementation by other manufacturers.

Index Terms—MCU, Firmware Upgrade, Security, OTA.

I. INTRODUCTION

The Internet of Things (IoT) is a novel paradigm that has received a lot of attention, featuring a wide range of applications, such as smart home, smart city, smart healthcare and so forth. In the context of IoT, all devices are interconnected. In one application, a person may own many devices and control them from his smartphone, which offers unprecedented convenience to users. The market of IoT products is estimated to hit 25 billion by 2021 according to the forecasts from Gartner [1], and Forbes [2] research also shows that more than 51% of companies have launched IoT projects.

However, not all of these companies have been prepared for the IoT era due to various reasons. One of the biggest concerns is the rampant security issues in IoT products, which hinders the development of the IoT market. According to a recent report from SonicWall, there were 32.7 million attacks against IoT which occurred in 2018, while these attacks have been escalating throughout 2019 with an explosive growth [3]. The attack surface of IoT involves various dimensions, such as network attacks and firmware attacks. Among them, the firmware attacks are particularly challenging to address due to the heterogeneous hardware. Even for a specific type of attack, such as a buffer overflow attack, there is no general countermeasure that can be adopted to work unanimously across different platforms.

This paper studies attacks that may occur during a firmware upgrade and discusses the possible countermeasures. Specifically, instead of proposing a general firmware upgrade attack or countermeasure, which may not be realistic given the hardware capability, we have conducted a set of case studies to gain real world insight into the issues. We exploit the flawed design of several different products and investigate a defense that hinders these attacks. We also discuss the potential exploit during implementation. The motivation of this paper is to offer guidelines for implementing secure firmware updates, and to identify pitfalls during implementation for IoT vendors.

We first demonstrate how an attacker can break IoT security via a flawed firmware upgrade mechanism. In this case study, we use a popular air quality sensor from PurpleAir [4]. Air quality monitoring is an important application of IoT in the modern era, due to the pervasiveness of various environmental pollutants, such as dust and smoke. In [5], we have successfully exploited the communication protocol of the sensor device, breaking its data integrity. In this paper, we exploit the firmware of the sensor device. We systematically analyze the firmware upgrade mechanism and identify the fact that the firmware upgrade mechanism does not involve any authentication or encryption processes. We therefore deploy two types of attacks, a hardware attack and a remote attack. In the hardware attack, we assume that the attacker can physically access the air quality sensor and set up a connection to the MCU via its debugging UART (Universal Asynchronous Receiver/Transmitter) interface. This is a reasonable assumption, since an air quality sensor can be placed outside and free to touch by anyone. In this attack, the attacker can either flash a malicious firmware to achieve destructive purposes (e.g., injecting fabricated data into the air quality sensor network), or steal the sensitive information from the firmware. The sensitive stolen information can be the pre-configured password
of a Wi-Fi network, which may bring risks beyond the product itself. In the remote attack, the attacker can exploit the firmware upgrade mechanism remotely. The attacker can impersonate as a server, fabricating a malicious firmware and sending it to the sensor. The sensor device flashes the malicious firmware back without any verification process. The attacker can also impersonate as a sensor, requesting the newly firmware from the server and exploiting the firmware bugs.

Given the lack of security measures for protecting firmwares, efforts have been taken to counter attacks [6], [7], which enforce the authentication and encryption during the firmware upgrade. MITRE Cyber Academy [8] proposes requirements for designing a secure firmware upgrade system. We follow these requirements carefully and implement a prototype of the secure firmware upgrade system on an ATmega1284P chip. ATmega1284P is a high performance AVR-architecture based chip, featuring internal security features such as a lock-bit mechanism, which prevents hardware based cracking attacks. Essentially, the implemented secure firmware upgrade system contains four components, namely a secure bootloader, a firmware protection tool, a firmware upgrade tool, and a readme tool. The secure bootloader is internal to the chip, while the others are maintained by the vendors. We assume that an AES key is predefined before deployment and known only by the vendor tools and secure bootloader. Specifically, the secure bootloader is used to verify the integrity of the firmware and perform decryption with the pre-defined key. The secure bootloader also installs the firmware into the chip. The firmware protection tool is used to encrypt the firmware when the firmware is released. The firmware update tool works with the secure bootloader cooperatively to install the firmware. The readme tool is designed for vendors to extract the firmware out of the chip when unexpected errors occur on the chip after deployment, so that the vendors can perform firmware analysis to diagnose the issues.

We discuss the possible pitfalls that may occur during the implementation of a firmware upgrade system. We identify these pitfalls through our own practice, and we believe that these pitfalls may occur during the implementation of other applications. If these pitfalls are not carefully addressed before deployment, they may lead to grave consequences. For example, the mis-configuration of the lock-bit allows aforementioned attacks to deploy without any changes. The Clock Glitch Attack, which is caused by overclocking [9] of a microcontroller (MCU), may allow the encryption process to be bypassed. Other pitfalls, such as the leakage of the key and firmware verification bypass, are possible due to the flaws of implementation, breaking the firmware security in various ways. While we are aware that the attacks and pitfalls we demonstrated are just the tip of the iceberg, we hope that a more practical approach to these issues can educate vendors and help them to avoid similar pitfalls.

The main contributions of our paper can be summarized as follows:

1) We systematically analyze the firmware upgrade mechanisms of air quality sensor device from a known brand PurpleAir, and deploy attacks against the sensor. From our understanding, we are the first to launch a firmware exploit against PurpleAir.

2) We design and implement a secure firmware upgrade system on an ATmega1284P according to the requirements from MITRE Cyber Academy, which address the attacks that occur during the firmware upgrade. We try to provide reasonable guidelines to follow on design and implementation in regards to firmware upgrade security.

3) We point out the pitfalls of the secure firmware upgrade system. To notice, all these pitfalls are identified through our own practice. We try to help other vendors to avoid these pitfalls on their own implementation.

Roadmap: The rest of our paper is organized as follows. In Section II, we briefly introduce the primitives involved in our paper, including MCU, Over-The-Air (OTA) and so forth. In Section III, we demonstrate the attacks against air quality sensor from PrepleAir. Subsequently, we show the detailed design of a secure firmware upgrade system in Section IV. Afterwards, Related work is presented in Section V and we make a conclusion in Section VI.

II. BACKGROUND

In this section, we present IoT primitives which are used in our paper. Particularly, we first provide a brief introduction to Microcontrollers (MCUs), and then we show how MCUs initiate Internet connections. Afterwards, we introduce the Over-The-Air (OTA) mechanism.

A. Microcontrollers

A microcontroller is a small programmable processor designed for embedded systems. Compared with other programmable chips, MCUs are characterized with minimized power consumption, lower price, small size, but limited computation and memory resources. Although the hardware resources [10] are limited to MCUs, its excellent extensibility allows it to append other functional modules, e.g., external flash and a network interface. These features lead to a broad implementation of MCUs for lightweight IoT applications such as air quality sensor networks. As the fundamental hardware of the smart devices, MCU plays the role of “a smart brain”, in that it handles all the incoming system
tasks, processes data, makes decisions, and controls peripherals. Due to its restricted resources and relatively simple architecture, MCUs are often dedicated to one or more simple tasks instead of processing multiple complex tasks simultaneously.

A MCU chip, while device specific, fundamentally consists of a central processing unit (CPU), system clock, memory, and peripherals [11]. Based on the support of these hardwares, the firmware, which is also called application image, can be burned to the memory (internal or external) of the MCU and executed.

B. Wireless Internet Connection

In the context of IoT, the wireless Internet connection is an essential element of IoT devices for connecting the small gadgets to much more powerful servers. The wireless Internet interfaces of MCU can be categorized as internal or external based on their integration method. The module could also be further categorized based on their functionality features. Some manufacturers have noticed the importance of an Internet connection, thus, integrate the network interfaces inside the MCU as its basic component. These integrated interfaces are usually chosen as WiFi or Bluetooth low energy (BLE), which are the most commonly used interfaces in IoT gadgets [12].

Even though many MCUs have integrated network interfaces internally, there is a considerable number of MCUs selected to provide the fundamental functions of system control and leave Internet tasks as optional. Owing to the extensibility of MCUs, a developer can still apply such MCUs to the IoT gadgets by adding their choice of external network interfaces. For instance, the ATWINC1500 add-on WiFi module from Microchip provides a WiFi connection to the microcontroller and the WL1831MOD module from Texas Instrument offers both WiFi and Bluetooth connections.

In recent years, a new type of connectivity technology, cellular low-power wide-area network (cellular LPWAN), came into IoT developers’ sights. This technology provides a low-cost solution to the Internet connection of IoT systems of which sensors distributed in a wide area periodically transmit small amounts of data through the network. Constructed based on the traditional Long-Term Evolution (LTE) standard for mobile device wireless communication, Narrowband IoT (NB-IoT) and Cat-M1 are two typical types of cellular technology for IoT devices. The service of NB-IoT and Cat-M1 are provided by the cellular providers such as AT&T and Verizon within the U.S.. By connecting to the nearby towers of the service provider, IoT devices can connect to the network using different bands from the standard LTE. The cellular IoT modules that provide NB-IoT and Cat-M1 can also be connected to the MCU externally to provide Internet connection.

It is worth noting that nowadays’ MCUs are well equipped with protocols such as Universal Asynchronous Receiver/Transmitter (UART) and Serial Peripheral Interface (SPI), providing convenient connections between the external Internet interfaces and MCUs. Lower level programming burdens are sufficiently hidden from end users, and external Internet interfaces can be controlled by pre-coordinated string-based commands.

C. Over-The-Air mechanism

OTA refers to the mechanism that devices can be updated by wirelessly downloading the newly released firmware from remote servers and reprogramming itself. Considering the convenience and stability of the protocol, the OTA mechanism fits for the distributed IoT devices well. Upgrading through secure OTA programming is in fact critical for improving the lifetime of these smart gadgets while also providing firmware protection [13]. Not only does OTA increases the functionality and scalability of the devices, but security vulnerabilities can be fixed even after the devices have been launched. In addition, costs for maintaining the distributed IoT applications are largely reduced by automatically updating remotely.

However, network vulnerabilities may exist in the data downloading process if the network protocol, e.g., WiFi or BLE, is not secure enough. For instance, an OTA task built atop the Hypertext Transfer Protocol (HTTP) sends the firmware to the IoT devices without a firmware authentication process or data encryption mechanism. Furthermore, a reprogrammable MCU may allow malware to intrude into the devices. Therefore, to ensure the scalability and security of the IoT devices, a secure OTA mechanism has to be implemented.

III. CASE STUDY - PURPLEAIR

In this section, we present a case study to demonstrate the grave impacts against a popular air quality sensor from PurpleAir. We give a brief introduction to the PurpleAir sensor device. Afterwards, we perform a firmware analysis, revealing the architecture of its firmware. Finally, we discuss the potential exploits against the sensor device.

A. PurpleAir Sensor Devices

Fig. 1 illustrates a suite of construction of the PurpleAir sensor device, including an air quality sensor, an MCU, and a power supply circuit board. The air quality sensor measures ambient mass (e.g. particle number concentrations of PM2.5, humidity) and transmits the measurement to the MCU. The MCU here is an ESP8266 [14] chip, featuring various functionality such as network connectivity and components communication for different peripherals. The power supply circuit
board is an integrated circuit board, where other components are soldered. As the term suggests, the power supply circuit board is used to power all components. Collectively, these components can monitor multiple environmental metrics and report the measurements to the cloud server. Moreover, in considering the firmware may need to be updated after deployment, PurpleAir offers an OTA mechanism, allowing the sensor device to download a newly released firmware from the cloud server and flash it locally.

B. Analysis of the Firmware

In the following subsection, we elaborate on how to access the flash information of the sensor device for malicious purposes. To deploy an attack, an attacker must first analyze the firmware. To this end, an attacker may want to communicate with the chip and read the flash contents out of the chip. Recall that the sensor device integrates the ESP8266 as its MCU. The ESP8266 offers a universal asynchronous receiver-transmitter (UART) interface for data transmission. To enable the UART communication, we connect the sensor device to a debugging computer via a USB cable with a correct baud rate (115200). To initiate communication, we install a tool named esptool [15] onto our testing computer to receive the data sent from the sensor device. Esptool is a Python-based platform, which is designed to communicate with the ROM bootloader of Espressif Systems [16]. An example of Espressif System is the ESP family of MCUs, such as ESP8266 and ESP32 [17]. Esptool offers various shell-code scripts, allowing the manipulation of the Espressif System. Here, we list a few of these shellcodes that closely relate to our research.

1) “esptool.py -port PORT flash_id”. As shown in Fig. 2, this command can read the basic information of a firmware, such as the device MAC address, flash size, and manufacturer information. For clarity, the PORT argument always refers to the UART port number unless explicitly stated otherwise.

2) “esptool.py -port PORT -b 115200 read_flash 0 0x200000 flash_contents.bin”. As shown in Fig. 3, this command reads the flash content out of the chip. In our example, Esptool tries to read “0x200000” bytes which start from the address “0” of the flash memory and saves it onto the disk with a file name “flash_contents.bin”.

3) “esptool.py erase_flash”. This command erases the entire flash. That is, when this command executes, all bytes of the flash will be replaced by meaningless “0xFF” bytes. Similarly, “erase_region” is a command that can be used to erase a specific data section of the flash memory, by attaching the starting address and flash size, as shown in previous commands.

4) “esptool.py -p PORT write_flash 0x1000 my_app.bin”. This command is used to flash a binary file (my_app.bin in our case) onto the chip. The parameters are similar to the previous commands.

C. Architecture of PurpleAir Flash

Based on the analysis above, we now present the architecture of the flash from PurpleAir. The size of the flash is 2MB, while the last 1MB is meaningless padding data. According to the ESP8266 official documentation [18], the flash is usually split into four data sections, including two program images, an Electrically Erasable Programmable Read Only Memory (EEPROM) region, and a default data section. As shown in Fig.4, each program image contains two binary files, a boot.bin and a user.bin. The boot.bin is used to store boot data, such as some configuration that closely
relates to system booting. The `user.bin` is used to store program instructions, i.e., application logic. The two program images can be identical. We observe this case with the PurpleAir firmware. EEPROM is a type of non-volatile memory that provides persistent storage of data across reboots. In the case of PurpleAir, the WiFi SSID and password are stored in this region. Finally, the default data section contains a `esp_init_data_default.bin` file and a `blank.bin` file, which store default system parameters, such as Wi-Fi configurations other than SSID and password [19].

![PurpleAir Firmware Flash Map](image)

**Fig. 4.** PurpleAir Firmware Flash Map

### D. Exploits and Attacks

The ultimate goal of our attack is to gain control of the air quality sensor. To this end, two types of attacks will be demonstrated in this section, namely the physical attack and remote attack. In the physical attack, we assume that the adversary can physically access the sensor device, i.e., via a physical connection to the MCU. This assumption is reasonable since the air quality sensor can be placed outside and free to access. Furthermore, the sensor may be left unmonitored for extended lengths of time. However, in the remote attack, we do not assume that physical access is available to the adversary.

**Physical attack:** An adversary can deploy the physical attack by connecting the MCU to a computer. We list some potential exploits that can arise from this attack to demonstrate its grave consequences: (i) *Flashing a malicious firmware.* An attacker can program a malicious firmware and flash it back to the chip. The modified firmware can inject fabricated data into the air quality sensor network, which may misinform the public. (ii) *Stealing Wi-Fi credentials.* Due to the lack of authentication and encryption of the firmware, the flash contents are free to access. We can therefore extract the Wi-Fi credentials out from the flash, which causes damages beyond the air quality sensor network.

**Remote attack:** Recall that the PurpleAir offers an OTA mechanism, which allows the sensor device to update its firmware after deployment. In fact, when a PurpleAir sensor device reboots and connects to the cloud server, the sensor device will send a request to the cloud server and query the current firmware version. If the firmware is outdated, OTA will be performed to synchronize the firmware from the cloud server. This mechanism ensures the sensor is up-to-date, once a new firmware has been released. After carefully analyzing the PurpleAir OTA mechanism, we identify a severe vulnerability: the sensor does not authenticate the server, and the server does not authenticate the sensor. Moreover, the data exchanged during communication is in plaintext.

![The overview of remote attack](image)

**Fig. 5.** The overview of remote attack

The lack of authentication in the PurpleAir air quality network raises some grave concerns on their products. As shown in Fig. 5, an attacker can either pretend to be the cloud server or a rogue sensor device and achieve his malicious goals: (i) When the attacker impersonates the server, the attacker can trivially fabricate a malicious firmware and send it to the sensor device. The sensor device flashes the malicious firmware back without any verification process. Afterwards, the malicious firmware will execute on the sensor. (ii) When the attacker impersonates a sensor device, the attacker can obtain the new firmware by sending a query request to the cloud server, as shown in Fig 6. No encryption is adopted for the firmware either. We have confirmed these two attacks in our experiments.

![PurpleAir update response packet with new firmware attached](image)

**Fig. 6.** PurpleAir update response packet with new firmware attached.
IV. A POSSIBLE COUNTERMEASURE AND ITS PITFALLS

To counter attacks during the firmware upgrade, many efforts have been taken [6], [7], [20]. One effective solution involves mutual authentication and encryption during the firmware upgrade. In this regard, MITRE Cyber Academy [8] proposes requirements for designing a secure firmware upgrade system. However, the requirements do not involve any implementation details. For the sake of feasibility, we design and implement a secure firmware upgrade system according to the proposed requirements and discuss the potential pitfalls afterwards. The motivation here is to provide good guidelines to follow in order to implement secure firmware updates, and how to avoid pitfalls during the implementation.

A. Overview

To guarantee a secure firmware upgrade, encryption and authentication of the upgrade procedure must be incorporated and enforced. Specifically, the vendor would encrypt the firmware after release, while the decryption key would be inside each product. Hence, only the product can decrypt the firmware. Similarly, the product would verify the integrity of the firmware. To this end, the hash value of the target firmware would be encrypted and attached together with the encrypted firmware, so that an attacker cannot modify the ciphertext of the firmware and the integrity is guaranteed. Finally, to prevent the attacker from leveraging physical attacks to obtain sensitive data, such as the decryption key or the firmware, the debug interfaces, such as JTAG and SPI port, should be disabled or at least have limited accessibility (e.g. only the vendor itself can access the firmware). The lock bits should also be enabled if the chip of the product supported so. In this way, firmware analysis will fail.

As shown in Fig. 7, a secure firmware upgrade system consists of four components, including a secure bootloader, a firmware protection tool, a firmware update tool, and a manufacturer readback tool. During a firmware upgrade process, the secure bootloader is used to decrypt and verify the firmware. It also copies the decrypted firmware from external memory to the program memory. Then the device reboots to run the new firmware. The bootloader may also copy the encrypted firmware from the program memory to external memory when the debugging is needed. Notice that in this case, only the vendor itself has the accessibility of the debug interface. The firmware protection tool is used to encrypt the firmware. The firmware update tool assists the bootloader by copying the encrypted firmware into the flash memory. Finally, the readback tool is used for vendors debugging when the firmware has unexpected errors. Basically, the readback tool receives an encrypted firmware from the bootloader, so that the vendors can perform firmware analysis.

To notice, in the real world scenario, the firmware update tool can be replaced with the internal OTA mechanism introduced in section II. In this case, the bootloader obtains a firmware from the cloud server, then flashes it into the chip. The other functions of the bootloader and the other three tools will work without any changes. However, in some cases, the firmware update tool may be preferable—for instance, when the new firmware is too large to download without overwriting the previous firmware. For the sake of brevity, we will not reiterate details on MCU-based OTA, which were already discussed in section II.

B. Detailed Design

We now present a detailed design of a secure firmware upgrade system. As a demonstration, we implement the design on an ATmega1284P shown in Fig. 8. ATmega1284P is a high performance, low power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture, featuring 128KB ISP flash memory, 16KB SRAM, 4KB EEPROM, 32 General Purpose Working Registers, two UART interfaces, an SPI serial port, and a Joint Test Action Group (JTAG) test interface. JTAG is provided for on-chip debugging and programming. In regards to security, the ATmega1284P provides program locking mechanisms for software security. Specifically, there are three fuse bytes (low byte fuse, high byte fuse, and extended fuse) and a lock byte of the MCU which are used in the locking mechanism. The low byte fuse is used to deal with the clock related operations. The high byte fuse has several different settings, such as preserving or erasing EEPROM and the bootloader attribute settings. The extended fuse is used to set the brown-out detection trigger level, which refers to the voltage level setting. Locking mechanism prevents unauthorized read and write access to the flash memory and EEPROM.

1) Secure Bootloader: In our current implementation, the secure bootloader can work in three modes:
firmware loading mode, firmware booting mode, and readback mode. During firmware loading mode, the bootloader receives an encrypted firmware from external memory via the firmware update tool, and verifies the integrity of it, which prevents the modification of the encrypted firmware. Once verification succeeds, the secure bootloader decrypts the firmware and copies it into the flash program memory. Afterward, the bootloader enters firmware booting mode and boots the existing image in the flash program memory. The bootloader may also be in the readback mode. When the bootloader is in this mode, the vendor can communicate with it and the bootloader will copy and encrypt the firmware out of the chip. This may occur when the chip has some unexpected errors and a firmware analysis have to be done for diagnosis. In this case, the bootloader requires the readback tool to provide a username and password for authentication, which avoids the unauthorized access. These credentials were predefined and stored in the EEPROM by the vendor. Since we enable the hardware-based lock bit of the ATmega1248P, an attacker cannot compromise the firmware or the sensitive credentials via physical attacks.

2) Firmware Protection Tool: The firmware protection tool is used to generate a secure firmware, which contains five phases: (i) We generate an AES encryption key and configure the firmware protection tool to have access to the key. Before deployment, the AES key shall also be stored in the chip where the firmware will be installed. This can be done by installing an initial version of the firmware, where the key is hard-coded. (ii) The firmware protection tool encrypts the intended firmware with the AES key generated above. (iii) The firmware protection tool feeds the encrypted firmware into a hash function and computes the hash value. We denote the hash value as the ID of the firmware, since it uniquely refers to the firmware. (iv) The firmware protection tool feeds the firmware ID and other basic information of the firmware (e.g. the size, version number) into the hash function. The generated hash value here is used as the firmware checksum for the verification step. We then encrypted the hash checksum with the pre-defined key. The encrypted checksum will be a part of the secure firmware, so that the integrity of the firmware is also guaranteed. (v) The firmware protection tool uses the previous steps to generate a secure firmware. The secure firmware can be split into three parts as shown in Fig. 9: the header, the encrypted firmware and a release message. The header contains the firmware size, firmware version number, firmware ID, encrypted checksum. The release message provides a basic description of the secure firmware.

3) Firmware Update Tool: The firmware update tool assists the bootloader by copying the firmware into the flash memory, which contains two phases: (i) The firmware update tool sends the header of the secure firmware to the secure bootloader and waits for the response. In this phase, the bootloader is in the firmware loading mode and will perform the basic verification of the firmware, as discussed earlier. (ii) When the verification passes through, the firmware update tool sends the encrypted firmware to the bootloader and waits for the response. After the bootloader successfully receives the entire encrypted firmware, the firmware update tool then goes back to sleep.

4) Firmware Readback Tool: The readback tool communicates with the secure firmware via the UART port after deployment, allowing the vendors to obtain an encrypted firmware for analysis. To avoid unauthorized access to firmware, the secure bootloader will require a username and password which was predefined. Moreover, all data transferred between the readback tool and bootloader is encrypted by the predefined AES key.

C. Potential Pitfalls

1) Misconfiguration: Misconfiguration of the firmware may lead to severe consequences. A common misconfiguration is forgetting to enable the fuse or lock bits, which prevents hardware-based cracking after deployment. Enabling the fuse can hinder an attacker from accessing the memory via the JTAG/SPI port, while it may fail when an attacker deploys High Voltage Parallel Programming (HVPP). In this case, attackers can access and dump the firmware and EEPROM memory by using the Atmel Studio software [21] via the
AVR Dragon board. Recall that sensitive information is stored in EEPROM memory, such as decryption keys and credentials. Only by configuring the lock bits can this information be safe.

2) **Clock Glitch Attack**: For most integrated circuits (ICs), such as an MCU in our case, clock signals are used to synchronize the various register nodes in an integrated circuit. The maximum clock frequency of a chip is the frequency when the clock signal reaches each register node correctly and the instructions are fully executed, which refers to overclocking. Overclocking can improve CPU performance to some extent. However, it is not universally true due to hardware limitations. Extra heat may be generated in this case, which may hinder the performance as well. Therefore, an IC may not work properly when overclocking occurs, which may also prevent the clock signal from reaching each node accurately.

A clock glitch is a short period of time when the system clock frequency suddenly increases. An attacker can intentionally create a clock glitch by applying a high voltage. When a clock glitch occurs, the MCU may not be able to execute the current instruction correctly, which causes the MCU to skip the current instruction. Worse still, an attacker can perform this attack to bypass the protection of encryption. To this end, an attacker may deploy the attack when the bootloader tries to load the decryption key from the EEPROM, which may lead the decryption key fail signal to load. By default, the failure of loading the decryption key will result in an all-zero byte array to be used as the decryption key. The attacker can then encrypt his malicious firmware with an all-zero byte array, and send the encrypted malicious firmware to the chip via the update tool. The malicious firmware will be verified and installed with, since the decryption key used here is the all-zero byte array. In such a way, the integrity protection can then be bypassed. To counter this, the straightforward solution is to verify the encryption key each time before performing the encryption, and abort the encryption process when the key is an all-zero byte array.

3) **The Leakage of the Key**: An attacker may obtain the encryption key in various ways due to the carelessness of vendors or the flaws existing in tools. For example, the key can be hard-coded in the firmware protection tool. In this case, an attacker can perform the reverse engineering process and extract the key from the binary file. Even if the key is hard-coded in the encrypted firmware, another potential pitfall involves using an insecure encryption algorithm, which may be subjected to cryptanalysis. Under such circumstances, an attacker can break the encryption of the firmware and obtain the hard-coded key that is stored in the EEPROM. Once the attacker obtains the key, various other attacks can be deployed, such as fabricating a malicious firmware. As a solution, the vendors should keep their encryption keys in a safe location and update it whenever needed.

4) **Bypassing the Firmware Verification**: Recall that firmware verification is involved in the update process. If the programming implementation is flawed, the attacker may skip the verification process and flash malicious firmware with no changes. Below we demonstrate a design flaw that may occur during the firmware upgrade. Basically, when the bootloader enters the firmware loading mode, the firmware is transferred page-by-page from the firmware update tool to the chip. That is, the chip first receives the data from the update tool and caches it into a buffer. The buffer will hold the data until it reaches to the size of one page, i.e., 256 bytes. Then, the entire page is written into the flash memory of the chip. This process repeats over and over until the entire firmware has been written to the chip. To notify the transferring task is completed, the firmware update tool will send a termination message to the chip. However, if an attacker intentionally terminates the upgrade process, this termination message will not be sent. In this case, if the program is designed to perform the firmware verification only when it receives a termination message, then verification will not be performed, since the program never receives this termination message. One patch for this type of attack is to enforce the secure bootloader to perform the verification process in all conditions. If the verification fails or the program terminates prematurely, the secure bootloader will reboot to the previous firmware.

5) **Flaws existing in Readback tools**: Recall that the bootloader requires user credentials (i.e., username and password) when the readback tool tries to communicate with it. However, if the user credentials are in plaintext, the entire system may still be in danger. In this case, the attacker may obtain the credentials and communicate with the bootloader to, for example, create a carefully-crafted message packet and exploit any vulnerabilities of the bootloader, such as a buffer overflow attack. This might allow the attacker to control the flow of the program. There are various ways for attack obtain the user credentials when encryption is not adopted. For example, a malware in the computer of the vendor may steal such sensitive information. To counter such an attack, the vendors must encrypt all of these user credentials and update them frequently.

V. **RELATED WORK**

The firmware attacks and the corresponding defense measures have quite deservedly received much recent attention due to the explosive growth of IoT products. In this section, we will review some attacks on a few mainstream chips as well as their countermeasures.

The Harvard-Architecture device is one of the most popular chips widely used all over the world. It was...
originally believed to defend against code injection attacks, since the data region and program region were physically separated. However, Francillon et al. [22] demonstrated a remote code injection attack against the Harvard-Architecture-based Wireless Sensor Network. A fake stack (containing malware) is injected into the victim’s data memory, and then a specially-crafted packet is sent to poke the vulnerabilities existing in the bootloader, leading the bootloader to copy the fake stack from data memory to program memory. In such a way, the malware is injected permanently, which allows the attacker to gain full control of the victim sensor. More recently, Krzysztof Cabaj et.al [23] showed that the Harvard architecture MCU is prone to code reuse attacks, which is a variant of the code injection attack. Their main contribution is a demonstration of the attack on the Arduino family of devices that use the Webduino web server, which is one of the most popular options for IoT devices. To counter code injection attacks running on sensor nodes, Tan [24] proposed and implemented a remote attestation protocol. Compared with other attestation protocols which may be subject to unauthorized alterations, they run their protocol on a tamper-resistant Trusted Platform Module (i.e. an Atmel AT97SC3203S chip), where all unauthorized alterations will fail. Sergio et al. [25] propose a software-based defense for Harvard architectures such as AVR MCUs, which thwart code reuse attacks. To this end, they randomize the code memory of MCUs. The main contribution of this work is proposing a software-based implementation rather than hardware, which targets endpoint users rather than manufacturers.

Shoei Nashimoto et al. [26] showed how buffer overflow attacks, together with multiple fault code injection attacks, can control the program flow of MCUs. They demonstrated their attacks on an AVR ATmega163 microcontroller and a 32-bit ARM Cortex-M0+ microcontroller. To counter the control flow attacks of software, Francillon et al. [27] proposed a control flow enforcement. The control flow attacks often involve the manipulation of the return stack. In this paper, the authors store the return stack in protected hardware rather than the typical location. They implemented the solution on an AVR MCU and showed that overhead was minimal.

Sergio et al. [28] implemented a proof-of-concept malware named ArduWorm that can compromise Arduino Yun, a popular IoT platform, by exploiting a memory corruption vulnerability existing in the victims Bridge library. Their insight is showing that the current design of the bridge library lacks proper access control and authentication. The exploit uses code reuse attacks, allowing malware to establish a backdoor and spread through neighbor nodes. The work also proposed possible remedies to mitigate the problem. John et al. [29] addressed two crucial issues in terms of creating stack-safe embedded software: security and efficiency. To address the security problem, they perform a whole program security analysis based on a context-sensitive abstract interpretation of machine codes. To address the efficiency issue, they adopt goal-directed global functions to reduce the stack memory requirements.

It can be observed all the above works address the firmware security but not firmware upgrade security. We now review the attacks related to the firmware upgrade attacks. Zachry Basnight et al. [30] examine how an attacker can modify the firmware on a Programmable Logic Controller (PLC) [31]. PLCs are a type of embedded device which provide a software-driven interface for users to modify the functionality of the device on a user level. Their main contribution is that they propose a firmware analysis methodology to attack such a type of device. A proof-of-concept attack on an Allen-Bradley Controllogix L61 PLC was also performed to validate the flexibility of their methodology. However, they do not propose any countermeasures. The effort that is closely related to ours is the research by Ang Cui et al. [32], who also poke the design flaws of the upgrade mechanism of the embedded devices, and demonstrate that an attacker can inject malicious firmware and take control of the embedded devices trivially. The examples used in their paper are the Laser-Jet printer. However, the example used in our paper is an air quality sensor. The compromised air quality sensor may inject fake data into the server, which may misinform the public and even mislead policy makers. Compared with a compromised Laser-Jet printer, which may only waste printing resources and cause frustration to users, the consequences of a compromised air quality sensor are much more devastating. Additionally, in this paper, the authors discuss possible countermeasures for the upgrade mechanism of the embedded devices, but do not propose the design details and implementation criteria. One major contribution of our paper is that we demonstrate how to implement a defense to firmware upgrade attacks, and we discuss the possible pitfalls. Checkoway et al. [33] show that the CD-based firmware update mechanism of automobiles can also be susceptible to remote compromise, and they offer possible remedies for mitigation. Jong-Hyouk Lee et al. [34] proposed a secure update system based on blockchain [35]. Their observation is that a blockchain system can provide an alternative software solution for data integrity and tamper resistance.

VI. CONCLUSION

This paper investigates the security issues which exist in the firmware upgrade process. We demonstrate that the firmware upgrade mechanism of the PurpleAir air quality sensor device is subject to our physical hardware attack and remote attack. To counter the discovered
attacks, we also investigate a plausible defense mechanism on an ATmega1284P chip, which follows the requirements proposed by the MITRE Cyber Academy. We show that firmware upgrade attacks are possible due to the flaws encountered in the design and implementation flaws of the countermeasures. This work and our survey of recent advances of MCUs in [36] will help vendors understand the importance of the secure firmware upgrade mechanism for MCU based IoT systems and avoid the pitfalls of this mechanism.

ACKNOWLEDGEMENT

We thank MITRE Cyber Academy for the ATmega1284P platform and development environment.

REFERENCES