Security and Privacy in IoT Era

Orlando Arias, Kelvin Ly, and Yier Jin

Abstract Trends in miniaturization have resulted in an explosion of small, low power devices with network connectivity. Welcome to the era of Internet of Things (IoT), wearable devices, and automated home and industrial systems. These devices are loaded with sensors, collect information from their surroundings, process it, and relay it to remote locations for further analysis. Pervasive and seeminly harmless, this new breed of devices raise security and privacy concerns. In this chapter, we evaluate the security of these devices from an industry point of view, concentrating on the design flow, and catalogue the types of vulnerabilities we have found. We also present an in-depth evaluation of the Google Nest Thermostat, the Nike+ Fuelband SE Fitness Tracker, the Haier SmartCare home automation system, and the Itron Centron CL200 electric meter. We study and present an analysis of the effects of these compromised devices in an every day setting. We then finish by discussing design flow enhancements, with security mechanisms that can be efficiently added into a device in a comparative way.

1 Introduction

Totalling an estimated of 15 billion devices, there are roughly two connected devices per living human [1]. This is thanks to trends in this past decade, which show a drastic increase in the number of Internet of Things (IoT) and wearable devices in the market. This trend is expected to continue, with an estimate of 26 billion

Orlando Arias
University of Central Florida e-mail: oarias@knights.ucf.edu

Kelvin Ly
University of Central Florida e-mail: rangertime@knights.ucf.edu

Yier Jin
University of Central Florida e-mail: yier.jin@eecs.ucf.edu

connected devices by the year 2020, the majority of which being IoT and wearable devices [2].

IoT and wearable devices mainly consist of sensor nodes with the ability of transmitting data. Very little processing often takes place within this type of devices, relying on remote services or nodes to perform the computational workload. The information collected by these devices can range from a simple heartbeat, to temperature and humidity data, to energy consumption patterns, all while providing functionality such as health monitoring and home automation. Because of the type of information these devices gather and store, they become prime targets for attackers. Further, given their always-on network connectivity some of these devices exhibit, these devices can be targets for malware, increasing their potential for harmful usage.

Although some manufacturers are aware of the privacy and security implications in IoT and wearable devices, in most cases, security is either neglected, treated as an afterthought, or implemented incorrectly. The few devices that implement security mechanisms usually employ software level solutions, such as firmware signing and signed binaries. These are methods reminiscent of those used in regular computing [3–12]. These solutions, however, do not consider the difference in usage patterns between IoT, wearable, and industrial devices when compared to traditional computing systems. This has proven to be insufficient at times. Furthermore, concentrating on software-based security mechanisms often leaves the underlying hardware platform unintendedly vulnerable, allowing for new attack vectors.

In order to understand the security and privacy issues associated with current IoT, wearable and industrial devices, their design flow, and their implication, we categorize types of vulnerabilities we have encountered during our research. We also examine trends in manufacturing while providing a discussion of their effects in the final device. We then present four case studies: the Google Nest Learning Thermostat, the Nike+ Fuelband SE Fitness Tracker, the Haier SmartCare home automation system, and the Itron Centrol CL200 electric meter. These devices were chosen because of their popularity and importance to the industry. Furthermore, we believe that these devices are good representatives of their respective categories. We will provide a security evaluation of these devices and present their vulnerabilities, demonstrating how their software-based solutions were insufficient to fully protect the device.

2 Design Practices and Taxonomy of Vulnerabilities

Throughout our study of Internet of Things (IoT) devices, wearable devices and IoT devices, we have found common patterns in their design flow. Although these patterns simplify the design process for manufacturers, it also leaves room for security oversights. In this section, we discuss common design patterns we have encountered while also presenting their consequences. We then categorize these consequences into common security vulnerabilities that are found in these embedded devices.

2.1 Common Design Patterns

Time to market is an important metric for companies looking to introduce their products while remaining competitive. This usually results in a shortened research and development phase which brings about patters on the design. We now discuss some of these patterns.

Reliance on Vendor Designs. There are cases where the lack of familiarity with the hardware being used has led to over reliance on vendor designs. That is, products are directly based on a design or application solution a vendor has provided. Whereas for targeted applications this may be sufficient, when the only available designs are for general purpose computing devices or development boards, it may lead to the unintentional exposure of interfaces that are meant for debugging or reprogramming purposes.

For example, Texas Instruments provides the EVM430-F6779 kit [13]. This kit is a demonstration platform and development board for smart meter and related applications. It is based around an MSP430F6779 microcontroller and a peripheral set necessary to build a 3-phase electric meter. Texas Instruments provides documentation [13] on how to design a smart meter around this platform, however, provides no details on security. As a development board, this platform comes equipped with the necessary debug facilities meant for testing. If left in a production run, an attacker can easily leverage these interfaces to leak internal sensitive information or even install malicious firmware to control device operation.

Software Source Models. At firmware level, some of the higher end devices commonly utilize Linux-based software stacks. However, other open source projects such as FreeRTOS [14] are also popular choices. Other manufacturers opt for proprietary solutions, such as Wind River's vxWors [15] or Blackberry's QNX [16]. Smaller devices are often designed using a hardware vendor's toolkit, such as Texas Instruments' DriverLib [17]. The general idea is to utilize a pre-existing framework, saving time and development costs on the device.

Whether the software development model directly affects security is a hard question to answer. Open source software provides the attacker with the means to easily find vulnerabilities to utilize as an attack vector. However, under an open source model, a manufacturer does not have to rely on a vendor for security fixes. Closed source software requires extra effort for an attacker to reverse engineer, providing a layer of resistance against finding vulnerabilities. However, manufacturers need to rely on vendors once a vulnerability is found.

Weak or Bad Cryptographic Implementations. If a device is designed to be remotely updated, it must be able to verify the downloaded image for both integrity and authenticity. This usually involves a cryptographic algorithm, sometimes many. Cryptographically securing a product is a complicated task, as proven by the countless vulnerabilities found in software, not only because of the mathematics involved, but because of implementation errors [18–23]. Two of these vulnerabilities are of critical importance to our research as it shows how weakly implemented cryptographic systems can be bypassed, providing for a way to remotely attack the device. These exploits describe how an attacker can remotely compromise a Belkin WeMo

Home Automation device by exploiting the faulty usage of SSL, allowing remote firmware installation by spoofing a distribution server, or by spoofing SSL servers via arbitrary certificates.

Debug Interfaces on Production Runs. It is often cheaper to write images to flash chips when assembling the device, rather than purchasing preprogrammed parts. Furthermore, the device must be functionally tested before it leaves production. This implies that the circuit board must expose programming interfaces and test points for the different components present within. Although at times unlabeled, these often unpopulated interfaces are not removed after testing. An attacker can utilize them to inject his own code on the unit or alter their functional behavior. The software component may also fall prey to this issue, as compilers can generate binaries that include debugging symbols, expressing the constructs that generated a certain block of machine code. Leaving these debugging symbols in production runs aids an attacker in reconstructing the original sources, allowing for easier vulnerability detection.

Supply Chain Threats. Hardware Trojans also pose a serious threat to IoT security. These malicious modifications to integrated circuits can leak key data to an attacker, cause a device to operate outside specified parameters, or otherwise render the device inoperable. Hardware Trojans further pose the threat of not being detected by normal testing methodologies, requiring expensive specialized tests to detect them. For example, a malicious adversary could insert a hardware Trojan in a cryptographic IP core utilized in a System-on-Chip (SoC) used in an IoT device [24]. When triggered, this Trojan weakens the entropy of the random number generator used to generate keys. If these keys are used to encrypt sensitive data that is being transmitted by the device, the amount of computational effort required by the attacker to decrypt the data is severely reduced.

2.2 Security Threat Taxonomy

We now group common security vulnerabilities found in embedded devices. These categories range from software-based issues to hardware-based errors. We enumerate the types of vulnerabilities below and discuss their implications.

Board Level Exploitation. During manufacturing, test points and debug ports are added to devices in order to ensure their functionality before being shipped. This is necessary as it is part of quality assurance during production. Furthermore, it is often cheaper to perform in-board programming of the device rather than purchase pre-programmed chips. Unfortunately, leaving open test points and programming or debug ports on the circuit board provides an avenue for an attacker with physical access to probe the device and test its functionality. For example, exploits on the XBOX 360 allows an attacker to downgrade the system to a vulnerable kernel version through a timing attack [25] utilizing the on-board debug facilities.

Chip Level Exploitation. Commercial off-the-shelf (COTS) components are designed with general purpose usage in mind. This is specially the case with micro-

processors and microcontrollers. These devices offer commonly used functions and peripherals with the aim to make them as flexible as possible. As such, documentation on the operation of these devices is often public knowledge. As such, COTS components are not designed to contain a per-device root of trust internally embedded.

However, vendors such as Texas Instruments are capable of designing and fabricating customized parts for application-specific scenarios. These parts, such as some of their OMAP-based parts sport an on-die root of trust. Unfortunately, chip-level exploitation of integrated circuits defeats this kind of protection. Semi-invasive and invasive probing can reveal the secrets contained within the root of trust of the device. Modern technology facilitates the reverse engineering and leakage of sensitive information stored on-chip. For example, by "bumping" the internal memory on an Actel ProASIC3 FPGA, researchers were able to extract the stored AES key [26]. Furthermore, vendors such as Chipworks are capable of performing most reverse engineering tasks on a device [27].

Boot Process Vulnerabilities. Devices that, due to processor and system limitations, chainload an operating system may present security vulnerabilities. Chainloading refers to running sequentially larger pieces of software until the target software has been reached. This is done since devices do not usually have all of their hardware or software mechanisms initialized during boot. However, an attacker may leverage issues in the boot process of a device to inject a malicious payload. Any protection mechanism that is not active from the time of boot can be leveraged by an attacker to insert a malicious payload.

The boot sequence is one of the main targets of attack, as many of the high-level protection mechanisms are unable to be executed during the boot process. Since these mechanisms are not present, it leaves the system open for attack, which makes this a critical area to protect. For example, the attack on the iPhone's bootloader leads to a chain-of-trust exploit [28].

Implementation Errors. Encryption and hash functions are used in smart devices to secure passwords and other sensitive information, in addition to playing a key role in device communication and authentication. These functions are mathematically proven to be secure and robust, however side-channel attacks and information based cryptanalysis methods are threatening their integrity. In addition, improper implementations of these functions and the utilization of cryptographically weak encryption algorithms threaten the security of these devices. For example, the Sony PlayStation 3 firmware was downgraded due to a series of vulnerabilities in weak cryptographic applications [29, 30]. Interestingly, while the problems have been repeated in modern smart devices, the mitigation methods have already been proposed decades ago [31].

Software-level vulnerabilities in smart devices are similar to those in traditional embedded systems and general computing systems. Because smart device software stacks are often derived from the general computing domain, any software vulnerabilities found in the general computing area will also affect these devices. Therefore, software patches are required to update smart devices against known software-level attacks. Recent examples include a stack-based buffer overflow attack in glibc [32].

Methods to mitigate software exploitation attacks often follow those developed in general computing areas [33, 34]. However, as discussed in [35] that these solutions may not fit in smart devices due the resource constraints.

Remote Access Channels. Smart devices are often equipped with channels that allow for remote communication and debugging after manufacturing. These channels are also used for over-the-air (OTA) firmware upgrades. Though these channels are extremely useful, their implementations are not always secure. During development, manufacturers may leave in APIs which allow arbitrary command execution, or developers may not properly secure the communications channel. Through this attack vector, attacks may be able to remotely obtain the status of the device, or even control the device. A modern example of a backdoor in a remote channel is the Summer Baby Zoom WiFi camera, which has hardcoded credentials for administrator access [36].

3 Case Study 1: Smart Thermostat

Boot process hijacking invalidates software level protection schemes before they are properly installed and loaded. In this case, attackers try to break the normal boot process through the vulnerabilities within the chain-of-trust and install customized userland images or kernel modules. Malicious payloads can be inserted into the kernel modules and/or userland filesystems. One example of this type of attack is the compromise of the Google Nest Thermostat [37–39].

The Nest Thermostat is a smart device designed to control a standard heating, ventilation and air conditioning (HVAC) unit based on heuristics and learned behavior. Coupled with a WiFi module, the unit is able to connect to the user's home or office network and interface with the Nest Cloud, thereby allowing for remote control of the unit. The thermostat is divided into two main components, a backplate which interfaces with the HVAC unit and a front plate which presents the main user interface. The largest part count is found in the front plate of the thermostat, which is driven by a Texas Instruments Sitara AM3703 system-on-chip (SoC) [40], interfacing directly with a Micron ECC NAND flash memory module, a Samsung SDRAM memory module and a LCD screen. Figure 1 shows the device's internal components and the overall device configuration.

Upon normal powering on process, the Sitara AM3703 starts to execute the code in its internal ROM. This code initializes the most basic peripherals, including the General Purpose Memory Controller (GPMC). It then looks for the first stage bootloader, `x-loader`, and places it into SRAM. Once this operation finishes, the ROM code jumps into `x-loader`, which proceeds to initialize other peripherals and SDRAM. Afterwards, it copies the second stage bootloader, `u-boot`, into SDRAM and proceeds to execute it. At this point, `u-boot` initializes the remaining subsystems and executes the `uImage` in NAND flash with the configured environment. The system finishes booting from NAND flash as initialization scripts are executed and services are run, culminating with the loading of the Nest Thermostat propri-

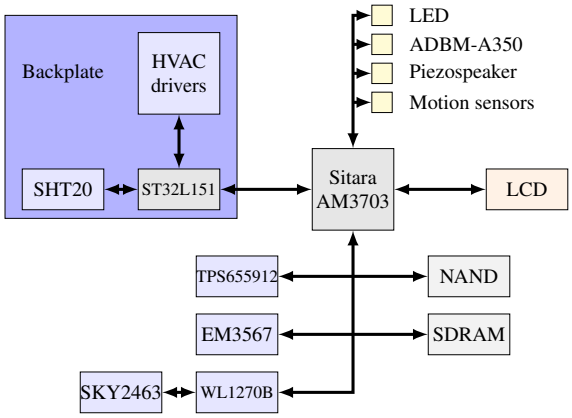

Fig. 1: Device map of the Nest Thermostat [39].

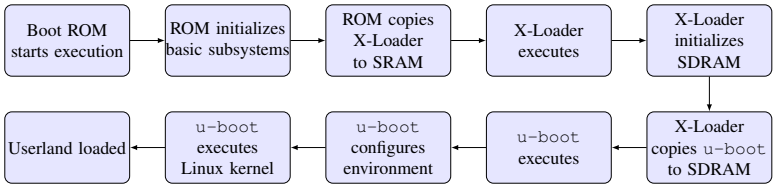

Fig. 2: Standard Nest Thermostat boot process.

etary software stack. Figure 2 shows the normal boot sequence of the device. The device boot configuration is set by six external pins, `sys_boot[5:0]`. After power-on reset, the value of these pins is latched into the `CONTROL.CONTROL_STATUS` register. Table 1 describes the boot selection process for a selected set of configurations.

sys_boot[5:0]	**First**	**Second**	**Third**	**Fourth**	**Fifth**
001101	XIP	USB	UART3	MMC1	
001110	XIPwait	DOC	USB	UART3	MMC1
001111	NAND	USB	UART3	MMC1	
101101	USB	UART3	MMC1	XIP	
101110	USB	UART3	MMC1	XIPwait	DOC
101111	USB	UART3	MMC1	NAND	

Table 1: Selected boot configurations

After performing basic initialization tasks, the on-chip ROM may jump into a connected execute in place (XIP) memory, if the `sys_boot` pins are configured as

such. This boot mode is executed as a blind jump to the external addressable memory as soon as it is available. Otherwise, the ROM constructs a boot device list to be searched for boot images and stores it in the first location of available scratchpad memory. The construction of this list depends on whether or not the device is booting from a power-on reset state. If the device is booting from a power-on reset, the boot configuration is read directly from the `sys_boot` pins and latched into the `CONTROL.CONTROL_STATUS` register. Otherwise, the ROM will look in the scratchpad area of SRAM for a valid boot configuration. If it finds one, it will utilize it, otherwise it will build one from "permanent devices" as configured in the `sys_boot` pins. Through this vulnerability, attackers can send a modified `x-loader` into the device, coupled with a custom `u-boot` crafted with an argument list to be passed to the on-board kernel. Arbitrary payloads can then be inserted into the device through the custom `u-boot` image [39].

4 Case Study 2: Nike+ Fuelband

Architecture wise, wearable and medical devices resemble IoT devices, however, they tend to have much less computational power and limited communication interfaces. Nevertheless, these units perform as much if not more data collection than IoT devices do. Although closely related to IoT devices, security vulnerabilities on wearable devices can lead to safety concerns for users. A pacemaker with wireless capabilities was proven to be vulnerable and could be used to affect the health of the patient [41]. Information leaks from fitness devices owned by corporate executives could be used against them, causing the corporation's value to deteriorate on the market, severely affecting its performance.

Much like our work with the Nest Thermostat, we performed a similar analysis on medical and wearable devices, looking for possible hardware vulnerabilities which may be utilized against an unsuspecting user. In the following subsections, we introduce as a secondary case study our work with the Nike+ Fuelband, a wearable device with fitness monitoring capabilities.

4.1 High Level Overview

The Nike+ Fuelband is a low-power Bluetooth 4.0-enabled fitness wristband designed to measure daily physical activity, such as the amount of steps taken, sleep patterns and estimate the amount of calories burned (see Figure 3). This is done by means of reading data from the on-board 3-axis accelerometer, which is subsequently stored within the unit. By means of software provided by the manufacturer, the unit can communicate with a Windows or OS X based computer, as well as Android and iOS devices. The collected data can then be analyzed, tracked and shared with the Nike+ online community. Periodic synchronization with the device can

Fig. 3: Nike+ Fuelband SE fitness tracker (credit: Nike).

be achieved with the mobile applications and real-time feedback is performed with the on-board LED matrix display. The device is powered by two Lithium-polymer batteries, advertised to provide up to four days of continuous usage.

4.2 Device Security

The Nike+ Fuelband contains a Bluetooth interface which it uses to communicate with a smartphone. Some settings of the Fuelband can be configured through these means and information from the band can be sent back to the smartphone using this channel. Firmware updates, however, are performed by means of the Nike+ application on a Windows or OS X based personal computer. Most of the communications from the smartband are done through the smartphone or personal computer application. Upon boot, the firmware is checked against a checksum before it is run ensuring a valid image.

4.3 Device Descriptive Overview

The main processing unit in this device is the ST Microelectronics STM32L151QCH6 microcontroller. Built upon an ARM Cortex-M3 core, this microcontroller is described in greater detail in Section 4.4. An LIS3DH 3-axis MEMS accelerometer from the same manufacturer interfaces with the STM32 by means of a Silego SLG46300 programmable mixed signal array. The 120-LED matrix is driven by an AMS AS1130 driver, which simplifies some LED matrix related operations. Power management is provided by the ST Microelectronics RS12, which also facilitates communications over USB 2.0. Bluetooth communication is achieved by means of a Cambridge Silicon Radio CSR1010 Bluetooth Low Energy module. Figure 4 shows the device map of the unit.

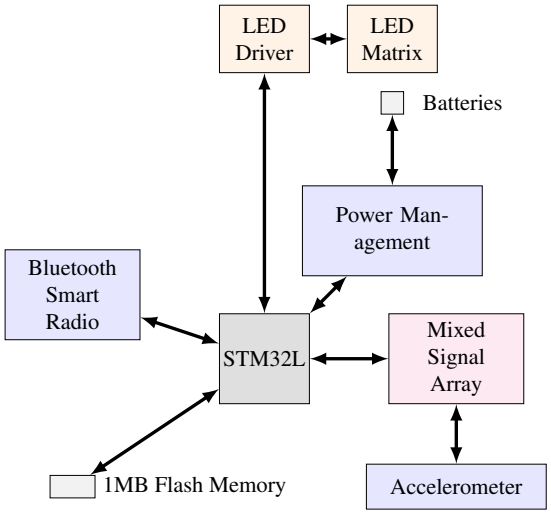

Fig. 4: Device map of the fuelband.

4.4 The STM32L151QCH6 - A Closer Look

The ST Microelectronics STM32L151QCH6 system on a chip (SoC), hereafter referred to as STM32, is an ultra-low-power platform offering a 12 channel DMA controller, 23 capacitive sensing channels and a CRC calculation unit. The SoC further includes a 96bit unique ID, a preprogrammed bootloader supporting both USB and USART programming, 116 fast input/output pins which are mappable to a 16 interrupt vector table. Storage wise, the STM32 in question offers 256KiB of flash storage with ECC support, 32KiB of SRAM, 8KiB of ECC supporting EEPROM and a 128B backup register. Included peripherals range from an LCD driver, to communication interfaces supporting USB 2.0, USART, SPI and I^2C [42].

The included ARM Cortex-M3 core supports both the Thumb and Thumb-2 instruction set architectures. Advanced low-power optimizations are achieved by means of multiple power and clock domains, architecture defined sleep modes and support for advanced low-power technologies such as State Retention Power Gating. A JTAG mechanism is provided by means of serial wire debug, which provides real-time access to system memory without halting the processor.

A simplified memory map of the STM32 is illustrated in Figure 5. The highlighted block of addresses in the figure are multiplexed between Flash or System Memory, depending on the status of the external `BOOT0` pin (see Section 4.5).

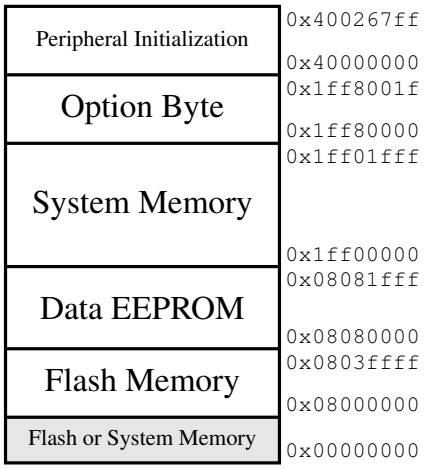

Fig. 5: Simplified memory map of the STM32L151QCH6.

4.5 Boot Process and Device Initialization

Upon device power on, the STM32 executes the code stored in its internal ROM, initializing the device's basic peripherals. Execution then continues from internal flash memory, which proceeds to finish device setup into a working model. Specific to the Nike+ Fuelband, this entails activation of the Bluetooth radio, mixed signal array and LED driver, along with the calibration of the accelerometer. At this point, the device is ready for regular usage.

The STM32, however, implements a secondary boot mode, which is triggered by holding the `BOOT0` pin to a logic 1 as the device starts. If started this way, the device initializes a basic set of peripherals and configures the USB subsystem. Then, if a USB cable is detected whilst being driven by the proper clock signal, the internal PLL reconfigures the system clock to 32MHz and the USB subsystem clock to 48MHz. The system proceeds to execute the DFU bootloader with USB interrupts enabled, as to allow for communication. Using this mechanism, the STM32 can be sent commands which allow for read and write operations to memory, changing memory protection modes and status retrieval.

4.6 Attack Vector on the Nike+ Fuelband

Although the STM32 documentation states that the microprocessor contains the necessary capabilities to lock external reads and writes against the internal flash, thus isolating the device's firmware from the external world, this protection was not employed on the Nike+ Fuelband. As such, the contents of flash can be freely modified by an attacker with access to the device.

The Nike+ Fuelband contains a standard USB connector which is used for both device charging and synchronization. This connector can also be used to write new firmware onto the device, however, the necessary access to the `BOOT0` pin is not externally provided. As such, the device must be opened in order to trigger the alternate boot sequence. Further complicating the issue is the fact that the microcontroller is packaged as a Ball Grid Array (BGA) and thus no direct access to the `BOOT0` pin can be obtained. Traces on the circuit board must then be followed in order to encounter a test point indirectly exposing the pin in question.

After following this process, we were able to indirectly locate the `BOOT0` pin, which was subsequently driven a logic 1 state by means of a 100Ω resistor connected to V_{DD}. This allowed us to enter the alternate boot mechanism and exploit the lack of read and write protection on the device. By means of standard ST Microelectronics development tools, communication over USB with the STM32 was achieved and the device's firmware was obtained.

With the device's firmware in our hand, we set on to modify it. The simplest change is one of string replacement, that is, find a string in the program that gets displayed at some point and change it to something else. With the change made, the modified firmware was written to the device, only to find normal functionality had ceased to exist. Further testing demonstrated that this was caused by a failure to compute the proper CRC for the image. Since the image was modified, the check failed.

Closer examination of the disassembled firmware image demonstrated that it utilized the CRC engine within the STM32 microcontroller in order to verify itself as genuine by checking the result of the CRC computation against a stored value. This value was found within the image itself, and thus easily modifiable. With the proper checksum added, the modified firmware was sent to the device and proven to work.

5 Case Study 3: Haier SmartCare

Commercial IoT devices which directly target end-users are often designed with emphasis on device functionality. Security features are often added in an ad-hoc manner where remote attacks are treated as the main threats. Therefore, commercial IoT devices often suffer from hardware-level vulnerabilities [37] which may be remotely exploited. In order to demonstrate these security vulnerabilities and help designers/consumers better understand the design backdoors, the Haier SmartCare home automation system is selected as a case study in this paper.

5.1 High Level Overview

The Haier SmartCare is a smart device designed to control and read information from various sensors placed throughout a user's home which include a smoke de-

tector, a water leakage sensor, a sensor to check whether doors are open or closed, and a remote power switch. These sensors are connected through the ZigBee protocol. The primary function of this device is to allow the user to better monitor their homes when they are away and to get alerts based on sensor information.

Fig. 6: Haier SmartCare device (credit: Haier).

In order for users to connect to the device, they must first download a mobile application from the manufacturer's website. Next, they must connect the SmartCare to their network using an Ethernet connection. Following, they must connect their mobile device to the same local network as their SmartCare. Once it is connected, they must open the mobile application and create an account through the manufacturer's cloud service, which allows users to view their sensor data outside of their local network. Once this has been established, the users will be able interact with the sensors from their SmartCare through the mobile application.

5.2 Hardware Analysis

The first step in our vulnerability analysis was to analyze the components on the SmartCare's hardware platform. The main processing unit is a TI AM3352BZCZ60, which is a part of TI's Sitara line of processors. The processor contains an ARM Cortex A8 with NEON extensions. The processor also supports the use of operating systems such as Linux and Android.Upon analyzing the data sheet for the processor, we were able to locate traces for UART on the device. The SmartCare PCB is shown in Figure 7.

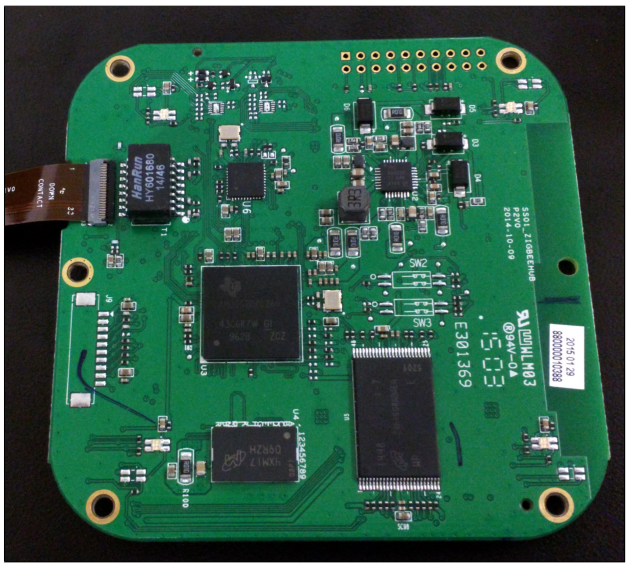

Fig. 7: SmartCare hardware platform.

By leveraging the UART connection, we are able to read serial data from the device. By setting the correct parameters in the terminal emulator and connecting a serial-to-USB device to the SmartCare, we were able to view its start up sequence. In the beginning of the boot process, the device prompted us as to whether we would like to stop the automatic boot sequence. Upon stopping the process we were dropped into a U-Boot shell. It is here where the we were able to modify specific boot parameters for the device, such as where to start reading from memory, and what the initial shell will be. By modifying the initial shell among other variables, attackers will be able to gain low-level access to the device. After modifying the parameters we initiated the boot process. Once the device had finished booting up, we were dropped into a rudimentary shell.

5.3 Into the Shell

After reading the boot output of the device, it was apparent that this device was running Linux. Being on a Linux device, it is necessary to know what kind of permissions we have, running `id` showed us that we were on the root account of the device. Looking through the BusyBox utility showed us that the device is capable of running a `telnet` server, allows for TFTP file transfer, and is able to fetch files from the web through `wget`.

Being on the root shell of the device also gave us the opportunity to look at the password hashes on the device, shown in Figure 8.

```
root@am335x-evm:/etc# cat /etc/shadow
root:RE0tDm4UmFgUo:16674:0:99999:7:::
```

Fig. 8: SmartCare hashed root password.

By referencing documentation on Linux shadow file structures, we were able to deduce that this device was using DES encryption on the password while also not using a salt. This means that the password is truncated to a maximum of eight characters then hashed. In order to obtain the root password for the device, the root password hash had to be cracked. The first attempt at cracking utilized a dictionary attack. In a dictionary attack, each password in the dictionary is hashed and subsequently checked against the hash in question. If the hashes match, then the password has been found, otherwise it will continue to check and hash each password in the list until it has reached the end. In this attack, a large word list containing approximately 32 million passwords was checked against.

Though 32 million passwords were checked against, none of them matched the root password of this device. The next option was a brute force attack, where every possible combination of characters is checked and hashed in order to find the root password. The total keyspace for a DES password using printable ASCII characters is $\sum_{i=0}^{8} 95^i$. This is a somewhat large keyspace, and may take hours or even days to go through every iteration on high performance hardware. Given that this method of attack is much more computationally intensive, we tried to optimize the cracking procedure leveraging high performance hardware with parallel processing capabilities. In our case study, we used two AMD R9 290 graphics cards to speed up the process

In our run, it took around five hours to get the root password. Since the root password for the device was known, the next course of action was to move onto another layer of attack. That is, we wanted to find out how we could attack other SmartCare devices using the secret learned from the device.

5.4 SmartCare Network Analysis

The new attack we tried to perform was a network-based remote attack. The first step in performing the network analysis was to scan the ports on the SmartCare to see if it is listening or transmitting on any of them. By performing a network scan we were able identify that the device may have had a `telnet` server running. Connecting to the device over telnet, we encountered a login prompt. Using the root credentials that were found earlier, we were able to get a root shell, which is shown in Figure 9.

Since we were able to get a root shell over a local network, the next step was to see what kind of traffic this device generates. In order to analyze its network traffic,

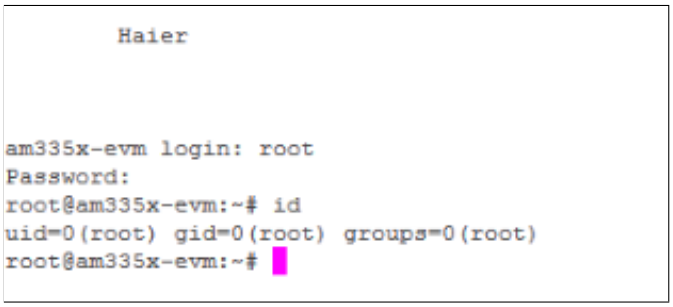

Fig. 9: SmartCare `Telnet` login prompt.

we had to perform a man-in-the-middle attack. This involved us using our computer as the gateway for the network the SmartCare was on. Through the gateway we were able to provide internet access. Using a packet sniffing program we were able to see what kind of traffic the device generates.

Once the network was up and running, we started the packet sniffer and looked at the network traffic. While most of the traffic going to and coming from the server was encrypted at the beginning, the device later fetched a firmware update over a plaintext HTTP connection, which is shown in Figure 10.

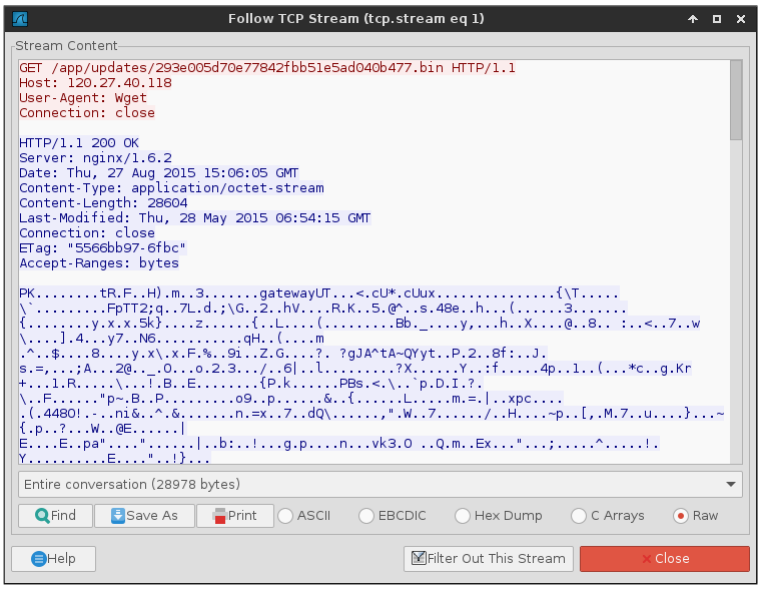

Fig. 10: SmartCare fetching update from manufacturer's server.

As we can see in Figure 10, the first line in red indicates the package it wants to receive, which in this case is the firmware update. The second line indicates where

it wants to get the firmware package from. The third line indicates the method it is using to receive the package, which in this case is `wget`. The blue section following shows the manufacturer's server's response to the firmware update fetch request, and subsequently the firmware image. Because the firmware update was fetched over a plaintext connection, and the SmartCare uses a standard utility to fetch the update, we decided to fetch the update ourselves. After fetching the update using `wget` and performing a file analysis on it, we were able to find that the firmware update was simply a ZIP archive.

Unzipping the archive allowed us to see the SmartCare's main binary along with `bash` scripts for updating the device and one of the SmartCare's main initialization scripts. Based on the initialization script, the device will set itself up, and then run the device's main binary. Knowing this information, the next step in our analysis was to see how the device handles firmware updates, which involves reverse engineering the SmartCare's binary.

5.5 SmartCare Binary Analysis

Using binary analysis software, we were able to search through the binary and see how it handles updates. The device utilizes the MQTT protocol in order to communicate securely with the manufacturer's server through an encrypted channel. MQTT is a Publisher/Subscriber protocol, where there is a broker which takes in information from publishers, and pushes the information to subscribers. The subscribers subscribe to topics, which are posted by the publishers. In our case, the SmartCare is a subscriber which communicates to the manufacturer's server to fetch the names of firmware updates, the correct hashes for the updates, commands from the user, and the current time. It also acts as a publisher, sending sensor information back to the manufacturer's server.

In terms of actually performing the firmware update, the device will fetch the package using the information gathered over MQTT. Once received, the device will run an MD5 checksum on the package and compare this hash to the hash provided by the manufacturer over MQTT. If both hashes match, the device will go through with the update. If the hashes do not match, the device will reboot, and start the entire process again. The whole verification mechanism is still under investigation for possible security vulnerabilities.

6 Case Study 4: Itron Centron CL200 Meter

Similar to commercial IoT devices, smart devices are also widely used in industrial applications. These devices, if compromised, may have a more serious impact than compromised commercial IoT devices. To better understand the security protections

in place for industrial IoT devices, we selected the Itron Centron smart meter as the other case study. Figure 11 shows the smart meter.

Fig. 11: Itron Centron CL200 smart meter (credit: Itron).

6.1 High Level Overview

The primary functionality of this device is to measure a customer's energy usage and report the collected information through an RF channel to a nearby meter reader or to a local substation. This information is then used to charge the customer for their energy usage, and may also be used to get statistics on community energy usage.

6.2 Hardware Analysis

Similar to our work on the home automation device, the first step in our analysis was to analyze the hardware platform of the smart meter. Inside of the device we were able to see a heavy-duty plastic cover, which guarded the main hardware platform. When looking at the hardware platform, we identified that it measures line voltage, measures reference voltages, checks the energy flow direction, energy pulse data, and checks the line frequency. Attached to the main hardware platform is a daughterboard, which is used when a company wants to implement functionality on the meter without having to replace the entire device.

In this case, the daughterboard is used to collect energy usage information along with tamper data and the ID of the board itself (see Figure 12). Located on the

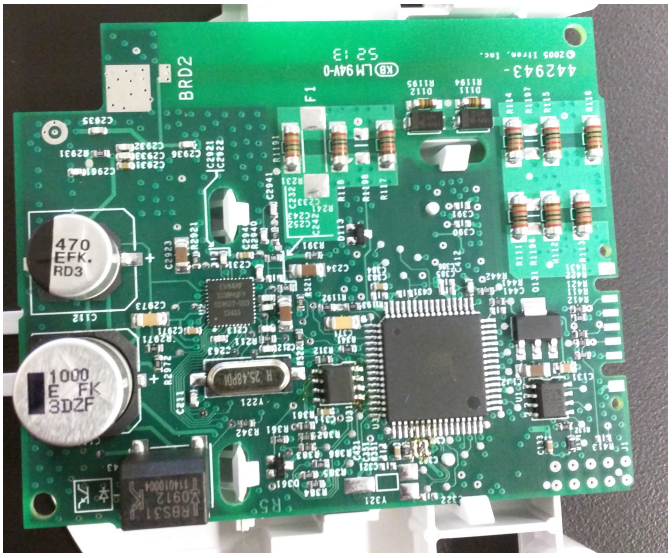

Fig. 12: Smart Meter CL200 daughterboard.

daughterboard is an ATMega microcontroller, a tamper sensor, and a 1 KB EEPROM. Through the microcontroller we were able to re-enable JTAG, and re-enable write access for on-chip memories.

6.3 Device ID Modification

For our analysis, our objective was to modify the smart meter ID in order for a meter reader to read the incorrect ID for the device. Upon further analysis, the ID was being stored in the external EEPROM. In order to figure out the ID of the meter, we had to read the ID on the meter itself, which is found on the front of the device underneath the grey cover. By analyzing the EEPROM dump, we were able to find where the ID was stored and change the ID to any arbitrary value.

6.4 Demonstration

Now that we had modified the ID of the meter, we needed to read the ID of the meter remotely to demonstrate that a smart meter reader will pick up the wrong ID from a modified device. Utilizing a software-defined radio (SDR), we were able to run a TCP server on the SDR and connect it to another program which parses wireless information and displays the ID, the tamper bit status, and the energy usage for the

meter. Through the experimental platform, we were able to demonstrate that due to the lack of proper protection, one compromised smart meter can "represent" itself as any other smart meter. Figure 13 shows the SDR output in which two smart meters share the same ID but different power consumption values. At the bottom of the figure, there is a meter which identifies as the other, however its power consumption is different than those above it. Through this vector, energy theft becomes possible.

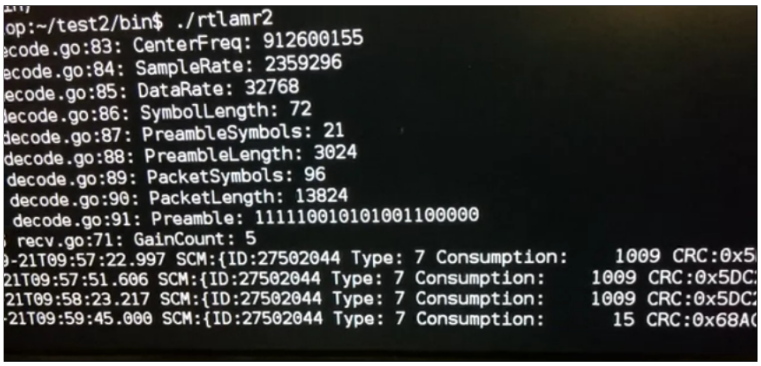

Fig. 13: Demonstration of the security vulnerability on the meter

7 Discussions

7.1 Security Impact to Network

A compromised IoT device can be utilized to further attack other units in an unsuspecting victim's network. Effects could range from simple backdoor injection to leaking user information and credentials to even causing physical harm to the user. As shown with the case of the Nest Thermostat, it can be used as a beachhead to other nodes within the network, allowing for discovery and attack of those nodes.

Furthermore, rogue services may be installed on the device, aiming to disrupt regular network operations. For instance, a rogue DHCP server may be utilized to inject DNS requests to a poisoned server which would return false information, allowing for traffic shaping. Address Resolution Protocol (ARP) based attacks are also possible, with the compromised device masquerading as the router, allowing for the capture and redirection of a target computer's network traffic.

Security issues with backdoored IoT devices are exacerbated by the fact that local network credentials need to be stored within the unit, thus becoming accessible to an attacker. Leveraging the extraction of network credentials allows for the introduction of extraneous devices into the local network, granting for new methods of exploitation against other nodes. In the case of the Nest Thermostat, the network

credentials are stored in regular text files, and even if these were encrypted, the algorithms necessary to obtain the clear text would necessarily be present on the device, granting the attacker the means to collect them.

7.2 Safety Concerns

Safety concerns arise when compromised IoT and wearable devices see on-field deployment. Due to the services these units provide, from communications to medical applications, a compromised device could then be used to cause physical harm to its user [41]. The Nest Thermostat could be employed to overstress the HVAC unit it is connected to, causing it to malfunction. Furthermore, all the information stored within the device can be utilized by the attacker to build a profile of the victim, aiding on the determination of a daily routine, the usage of which can result in facilitating the burglarizing of the victim's property.

7.3 Privacy Concerns

Almost all IoT and wearable devices, upon setup, will start collecting user information. For example, the Nest Thermostat will collect information such as the location of the thermostat, whether it is being used in a home or business, the postal code of the area and device information from the HVAC system to determine its capabilities. The on-board sensors on the thermostat will also collect temperature data, humidity and ambient light data, and by means of the onboard passive infrared sensor, whether somebody is moving in the room. Any direct temperature adjustments to the device are also recorded and utilized in algorithms to learn and compute comfort levels under different situations. Whenever the HVAC unit is activated, the thermostat will record the time and duration for which this happened. Using this information, the thermostat builds a profile for the users in order to help them feel comfortable whilst also providing energy savings. The Nike+ Fuelband will store the user's heartbeat and sleeping patterns, which can then be learned by the attacker. The information could potentially be used against the user, or against any entity the user is part of.

Although there are laws and standards defining data collection policies, some of these have proven to be ineffective and often antiquated, as demonstrated by information leaks from companies [43–45]. User information collected by the Nest Thermostat is stored within the unit and uploaded to the Nest Cloud. Local log files are sent to Nest as well and removed from the unit as to save space. System and software logs contain information such as the user's Zip code, device settings, HVAC settings and wiring configuration. Forensic analysis of the unit yields that the Nest Thermostat has code to prompt the user for information about their place of residence or office. Reports indicate that Nest plans to share this information with energy providers in order to aid with efficient power generation [46]. As for

the Nike+ Fuelband, the information collected and stored by the unit is then sent to a personal computer or mobile device, from where it can be publically shared with other users. Even if the information is not shared, an unauthorized third party still has access to the data from a compromised device and can use it for their own purposes. Although IoT manufacturers have gone through considerable efforts to ensure the secure transmission of this data, it is all for naught if it can be leaked at the source.

8 Related Work

Current IoT and wearable device literature often treats IoT from a network perspective or provide solutions that are inherently incompatible with the needs of a manufacturer. Few works have been published discussing the security of IoT devices themselves [47, 48]. In the ensuing sections, we summarize some of the previous work that has been presented in this area.

8.1 IoT Secure Protocols and Network Protection

An early survey about the IoT has shown that security and privacy are the main concerns that need to be addressed before IoT devices are widely adopted [49]. Proposed solutions for security rely on network protocols to ensure IoT security. Meanwhile, encrypted communication is treated as the effective solution for privacy protection. However, these proposed approaches do not consider the unique properties of IoT devices. The authors in [50] summarized all current security threats to the IoT network but these threat models are mostly derived from network security. They claim that hardware level attacks, such as differential power analysis (DPA) [51], are of high cost and therefore less harmful. Similarly, the authors in [4] treat IoT as an extremely interconnected network and list possible solutions to secure the IoT network including protocol and network security, data and privacy, identity management, trust and governance, fault tolerance, cryptography and protocols, identity and ownership, and privacy protection. All these methods try to regulate the communication between IoT devices under the assumption that all IoT devices are operating properly. The authors in [5] tried to solve IoT security through different IoT topologies: centralized architectures [6] and distributed architectures [7, 8]. Again, the network based solutions only emphasize high level structures without considering whether the available resources in IoT devices can afford these topologies.

Other research focuses on the secure communication between IoT nodes. For example, the authors in [9] focus on secure communication between IoT devices and present an Identity Authentication and Capability based Access Control (IACAC) model to protect IoT from man-in-the-middle, replay and denial of service (DoS) attacks. The authors in [10, 11] expand the definition of IoT to include four nodes

in a typical IoT network: person, intelligent object, technological ecosystem, and process. The authors claim that IoT security cannot be solved at a single-layer, but should require the analysis of the interactions between these nodes. A 2D version of the systemic approach was developed, which was expanded to a 3D version highlighting new functional plans of security [12].

Following this route, communication protocols were then developed to secure the interactions between IoT nodes such as 6LoWPAN [52] and Constrained Application Protocol (CoAP) [53]. The CoAP was constructed based on Datagram Transport Layer Security (DTLS) [54] and IPsec [55]. To counter the attacks at the transport layer, protocols were enhanced to use either HTTP/TLS or CoAP/DTLS by proposing a mapping between TLS and DTLS [56] or using secure tunneling on the transport layer [57]. However, these communication layer security analyses and protection methods ignore device level vulnerabilities and often impose unrealistic constraints on device deployment.

8.2 Hardware Based Protection

Besides network level protection, researchers from the industry have also tried to develop highly secure processor/SoC architectures for IoT protection. ARM TrustZone is an industry landmark in providing a basis of trust for various applications such as secure payment, digital rights management (DRM), enterprise and web-based services. TrustZone technology provides infrastructure foundations that allow a SoC designer to choose from a range of components that can perform specific functions within the security environment [58]. Intel proposed the concept of enclaves recently [59, 60]. An enclave contains software code, data, and a stack that are protected by hardware enforced access control policies. Samsung KNOX has also been developed with protection in mind [61]. KNOX provides a safe execution environment in a KNOX-enabled device where the userland is verified and a KNOX container holds sensitive data, such as corporate contacts and e-mails in a cellphone. If the device is deemed to be compromised by altering the bootloader, an e-fuse is blown inside the SoC driving the unit, thus branding it as untrusted. However, these hardware-based secure architectures are developed with passive protection in mind, whereas they do not detect and mitigate hardware and software level attacks. Samsung KNOX is possibly an exception to this, however, it remains to be proven whether or not it is possible to bypass any checks to the e-fuse protection in the bootloader. TrustZone environments have been proven to be compromised as shown in [62–64] by exploiting bugs in the software stack. Furthermore, these solutions do not transfer well to low power embedded units. For example, at the time of writing, Samsung KNOX is only available in select Android-based cellular phones and tablets.

9 Device Security Enhancement

9.1 Security Solutions Common to IoT and Wearable Devices

Verifying the firmware at update time is a step towards securing IoT devices, however, this is often done by the on-board software. As with the Nest Thermostat and the Nike+ Fuelband, the on-board software is trusted to be authentic. The implementation of this check, however, must be sound. For example, schemes that utilize random numbers must ensure the usage of a cryptographically secure random number generator, any used cryptographic certificates must be validated by a trusted Certificate Authority [22]. A weakly implemented cryptographic algorithm is no better than a lack of a cryptographic algorithm.

However, as we have demonstrated with our case studies, it is insufficient to authenticate an update image. The software stack must also be authenticated before it can reliably determine if an update is valid or not. With the devices compromised, we are free to bypass any checks on the update image, thus rendering the protection mechanism ineffective. A proper chain of trust in the hardware infrastructure of the device can aid the process of determining an authentic software stack [65].

The attack in both the Nest Thermostat and the Nike+ Fuelband could have been avoided had a proper chain of trust been implemented. Inherently, this needs the type of hardware support which is not available in either the Sitara AM 3703 used in the Nest Thermostat or the STM32 microcontroller used in the Nike+ Fuelband.

The exposure of debug interfaces in these devices further presents a risk. These are often left as residues from development prototypes or as testpoints used during manufacturing. These debug interfaces can also serve as the means to service IoT or wearable devices on the field, as to ease repairs. As such, we can see why they may be needed. However, these interfaces must be protected against attackers. For example, FRAM devices in the MSP430 lines provide means to both secure JTAG access and to protect certain memory segments from access using a built in IP Encapsulation Module [66]. Other microcontrollers and microprocessors offer the same kind of functionality, implementing means to restrict access to its debug units. As such, manufacturers are able to still expose these interfaces for testing purposes and lock them before they are deployed. Ideally, however, any debug interfaces should be removed from production runs or have proper protections.

9.2 Specific Solutions for IoT and Wearable Devices

Often, IoT devices provide a full operating system in which binaries are loaded into an userland. This simplifies the interface to the hardware and provides high level Application Programming Interfaces (APIs). The Nest Thermostat, for example, employs an embedded Linux stack which is used to launch the proprietary Nest application which relays commands to the backplate of the unit and controls the

communications channels. As we demonstrated in our case study, binaries can be injected into the filesystem of the unit and executed in devices that utilize this model. As such, extra protection must be added to devices that load binaries into a userland. A possible approach is to only load and execute cryptographically signed binaries. This requires the kernel to have a custom loader that verifies these binaries as they are prepared for execution. If the signature verification fails, then the binary is not run and the device is set into a failsafe mode, notifying the user of possible tampering.

In devices whose architecture is self contained, that is, microcontroller based systems, it becomes necessary to secure all update channels. External reprogrammability of the microcontroller and any debug interfaces it may feature must be disabled. The microcontroller must also be programmed before being placed in the circuit board, as to avoid adding unnecessary interfaces which could expose functionality.

9.3 Overhead of Security Solutions

There is usually a certain degree of overhead associated with any protection mechanism. Cryptography necessarily adds computational overhead to any protection scheme that utilizes it. It may be reasonable to expect then that any device which utilizes encryption or any other cryptographic function to require binaries with functions to include the necessary checks and have higher memory and CPU requirements in order to perform better. However, current industry solutions include parts which are capable of accelerating these processes, much like the microcontroller utilized in the Nike+ Fuelband which can accelerate CRC32 computations [42]. This reduces the software overhead needed to perform these checks, but slightly increases the area and power consumption of these parts. It should be noted, however that for most parts, power can be gated to the SoC subsystems that are not being utilized, thus reducing power consumption in the device.

10 Conclusion

As our case studies demonstrated, a non-secure hardware platform will inevitably lead to a non-secure software stack. A vulnerability in the design of the unit can result in its compromise. Furthermore, without being able to authenticate the running software, it can not be trusted to make decisions about its own validity. Due to the short time to market engineers are given to finish a product, we believe that most of the current IoT and wearable devices suffer from similar issues. Software protection becomes ineffective if the hardware is vulnerable to attack. This raises safety and privacy issues with users, is their information safe?

Moving forward, we will continue to probe other IoT devices for security, with the goal of finding vulnerabilities in their hardware. Ultimately, this will lead us to

a better understanding of design issues and how to correct them. We will attempt to build prototypes of smart devices that utilize our proposed chain of trust to test for their viability and ability to prevent malicious attacks.

References

1. D. Evans, "The internet of things - how the next evolution of the internet is changing everything," White Paper. Cisco Internet Business Solutions Group (IBSG), 2011.
2. P. Middleton, P. Kjeldsen, and J. Tully, "Forecast: The internet of things, worldwide, 2013," Gartner, 2013.
3. D. Welch and S. Lathrop, "Wireless security threat taxonomy," in Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society, pp. 76–83, 2003.
4. R. Roman, P. Najera, and J. Lopez, "Securing the internet of things," Computer, vol. 44, no. 9, pp. 51–58, 2011.
5. R. Roman, J. Zhou, and J. Lopez, "On the features and challenges of security and privacy in distributed internet of things," Computer Networks, vol. 57, no. 10, pp. 2266–2279, 2013.
6. A. Williams, "How the internet of things helps us understand radiation levels," 2011. [Online]. http://readwrite.com/2011/04/01/ow-the-internet-of-things-help.
7. D. Viehland and F. Zhao, "The future of personal area networks in a ubiquitous computing world," International Journal of Advanced Pervasive and Ubiquitous Computing (IJAPUC), vol. 2, no. 2, pp. 30–44, 2010.
8. H. Schaffers, N. Komninos, M. Pallot, B. Trousse, M. Nilsson, and A. Oliveira, "Smart cities and the future internet: Towards cooperation frameworks for open innovation," in The Future Internet, vol. 6656 of Lecture Notes in Computer Science, pp. 431–446, Springer Berlin Heidelberg, 2011.
9. P. N. Mahalle, B. Anggorojati, N. R. Prasad, and R. Prasad, "Identify authentication and capability based access control (IACAC) for the internet of things," Journal of Cyber Security and Mobility, vol. 1, pp. 309–348, 2013.
10. Y. Challal, Internet of Things Security: towards a cognitive and systemic approach. PhD thesis, 2012.
11. A. Riahi, Y. Challal, E. Natalizio, Z. Chtourou, and A. Bouabdallah, "A systemic approach for IoT security," in 2013 IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 351–355, 2013.
12. A. Riahi, E. Natalizio, Y. Challal, N. Mitton, and A. Iera, "A systemic and cognitive approach for IoT security," in 2014 International Conference on Computing, Networking and Communications (ICNC), pp. 183–188, 2014.
13. EVM430-F6779 - 3 Phase Electronic Watt-Hour EVM for Metering, [Online]. http://www.ti.com/tool/EVM430-F6779.
14. "Freertos reference manual: Api functions and configuration options," tech. rep., Real Time Engineers Limited, 2009.
15. A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of vxworks, linux, rtai and xenomai in a hard real-time application," in Real-Time Conference, 2007 15th IEEE-NPSS, pp. 1–5, 2007.
16. "Qnx operating systems." http://www.qnx.com/products/neutrino-rtos/index.html, 1982-2014.
17. MSP Driver Library, [Online]. http://www.ti.com/tool/mspdriverlib.
18. "CVE-2014-0160." Common Vulnerabilities and Exposures [Online]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.
19. "CVE-2014-2783." Common Vulnerabilities and Exposures [Online]. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2783.
20. "CVE-2014-2001." Common Vulnerabilities and Exposures [Online]. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-2001.

21. "CVE-2013-7373." Common Vulnerabilities and Exposures [Online]. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-7373.
22. "CVE-2013-6951." Common Vulnerabilities and Exposures [Online]. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6951.
23. "CVE-2013-6950." Common Vulnerabilities and Exposures [Online]. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6950.
24. G. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in Cryptographic Hardware and Embedded Systems - CHES 2013, vol. 8086 of Lecture Notes in Computer Science, pp. 197–214, 2013.
25. "Xbox 360 timing attack," 2007. [Online]. http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack.
26. S. Skorobogatov, "Fault attacks on secure chips: from glitch to flash," in Design and Security of Cryptographic Algorithms and Devices (ECRYPT II), 2011.
27. http://www.chipworks.com/.
28. "Apple iphone bootloader attack," 2008. [Online]. http://rdist.root.org/2008/03/17/apple-iphone-bootloader-attack/.
29. bushing, marcan, segher, and sven, "Console hacking 2010: Ps3 epic fail," in 27th Chaos Communication Congress, 2010. [Online]. https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.
30. R. Lemos, "Sony left passwords, code-signing keys virtually unprotected," eWeek, 2014. [Online]. http://www.eweek.com/security/sony-left-passwords-code-signing-keys-virtually-unprotected.html.
31. B. Schneier, "Cryptographic design vulnerabilities," Computer, vol. 31, no. 9, pp. 29–33, 1998.
32. "Critical security flaw: glibc stack-based buffer overflow in getaddrinfo() (cve-2015-7547)," 2015. [Online]. https://access.redhat.com/articles/2161461.
33. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in Usenix Security, vol. 98, pp. 63–78, 1998.
34. C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in Proceedings of the 12th conference on USENIX Security Symposium, vol. 12, pp. 91–104, 2003.
35. O. Arias, J. Wurm, K. Hoang, and Y. Jin, "Privacy and security in internet of things and wearable devices," Multi-Scale Computing Systems, IEEE Transactions on, vol. 1, no. 2, pp. 99–109, 2015.
36. B. Fowler, "Some top baby monitors lack basic security features, report finds," 2015. [Online]. http://www.nbcnewyork.com/news/local/Baby-Monitor-Security-Research-324169831.html.
37. G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart Nest Thermostat: A smart spy in your home," in Black Hat USA, 2014.
38. R. Potter and Y. Jin, "Don't touch that dial: How smart thermostats have made us vulnerable," in RSA Conference, 2015.
39. O. Arias, J. Wurm, K. Hoang, and Y. Jin, "Privacy and security in internet of things and wearable devices," IEEE Transactions on Multi-Scale Computing Systems, vol. 1, no. 2, pp. 99–109, 2015.
40. Texas Instruments, "AM3715, AM3703 Sitara ARM Microprocessor," 2011.
41. D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in IEEE Symposium on Security and Privacy (SP), pp. 129–142, 2008.
42. ST Microelectronics, "STM32L15xQC, STM32L15xRC-A, STM32L15xVC-A, STM32L15xZC Ultra-low-power 32b MCU ARM-based Cortex-M3, 256KB Flash 32KB SRAM, 8KB EEPROM, LCD, USB, ADC, DAC," no. 026119 Rev 5, 2015.
43. M. BARBARO and T. Z. Jr., "A face is exposed for aol searcher no. 4417749," The New York Times, 2006. [Online]. http://query.nytimes.com/gst/abstract.html?res=9E0CE3DD1F3FF93AA3575BC0A9609C8B63.

44. I. Reynolds and C. Fujioka, "Update 2-sony removes data posted by hackers, delays playstation restart," Reuters, 2011. [Online]. http://www.reuters.com/article/2011/05/07/sony-idUSL3E7G701T20110507.
45. Z. Whittaker, "Amazon's zappos in massive data breach 24 million affected," ZDNet, 2012. [Online]. http://www.zdnet.com/article/amazons-zappos-in-massive-data-breach-24-million-affected/.
46. M. Mombrea, "Google's real plan behind the purchase of the nest thermostat," 2014. [Online]. http://www.itworld.com/consumerization-it/416110/googles-plan-rake-cash-nest-thermostat.
47. J. H. Ziegeldorf, O. G. Morchon, and K. Wehrle, "Privacy in the internet of things: threats and challenges," Security and Communication Networks, vol. 7, no. 12, pp. 2728–2742, 2014.
48. A. D. Thierer, "The internet of things and wearable technology: Addressing privacy and security concerns without derailing innovation," Rich. JL & Tech., vol. 21, pp. 6–15, 2015.
49. L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," Computer Networks, vol. 54, no. 15, pp. 2787–2805, 2010.
50. S. Babar, P. Mahalle, A. Stango, N. Prasad, and R. Prasad, "Proposed security model and threat taxonomy for the internet of things (IoT)," in Recent Trends in Network Security and Applications, vol. 89 of Communications in Computer and Information Science, pp. 420–429, Springer Berlin Heidelberg, 2010.
51. P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in Advances in Cryptology – CRYPTO'99, pp. 789–789, 1999.
52. G. Mulligan, "The 6lowpan architecture," in Proceedings of the 4th Workshop on Embedded Networked Sensors, EmNets '07, pp. 78–82, 2007.
53. Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained application protocol (coap), draft-ietf-core-coap-13," in The Internet Engineering Task Force (IETF), 2012.
54. E. Rescorla and N. Modadugu, "Datagram transport layer security," RFC 4347, 2006.
55. S. Kent and K. Seo, "Security architecture for the internet protocol," RFC 4301, 2005.
56. M. Brachmann, S. L. Keoh, O. Morchon, and S. Kumar, "End-to-end transport security in the ip-based internet of things," in 21st International Conference on Computer Communications and Networks (ICCCN), pp. 1–5, 2012.
57. R. Seggelmann, SCTP: Strategies to Secure End-To-End Communication. PhD thesis, University of Duisburg-Essen, 2012.
58. ARM, "Building a secure system using trustzone technology," ARM Limited, 2009.
59. F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instruction ans software model for isolated execution," in Hardware and Architectural Support for Security and Privacy, 2013.
60. I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," in The 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP), 2013.
61. Samsung, "Samsung KNOX: Mobile Enterprise Security," 2015.
62. N. Keltner and C. Holmes, "Here be dragons: A bedtime tale for sleepless nights," in RedCon, 2014.
63. D. Rosenberg, "Reflections on trusting trustzone," in BlackHat USA, 2014.
64. T. Wei and Y. Zhang, "To swipe or not to swipe: A challenge for your fingers," in RSA Conference, 2015.
65. W. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," in Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on, pp. 65–71, 1997.
66. Texas Instruments, "MSP430 Programming Via the JTAG Interface," 2015.