This is an author-formatted work. The archival version for citation is:
R. Tsang, D. Joseph, A. Asmita, S. Salehi, N. Carreon, P. Mohapatra, and H. Homayoun, "FANDEMIC: Firmware Attack Construction and Deployment on Power Management Integrated Circuit and Impacts on IoT Applications," in Proceedings of *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 27-March 3, 2022.

FANDEMIC: Firmware Attack Construction and Deployment on Power Management Integrated Circuit and Impacts on IoT Applications

Ryan Tsang*, Doreen Joseph†, Asmita Asmita‡, Soheil Salehi§, Nadir Carreon¶, Prasant Mohapatra‖, and Houman Homayoun**
Department of Electrical and Computer Engineering, University of California Davis
Email:{*rchtsang,†dkjoseph,‡aasmita,§ssalehi,¶ncarreon,‖pmohapatra,**hhomayoun}@ucdavis.edu

Abstract

Supply chains have become a pillar of our economic world, and they have brought tremendous advantages to both enterprises and users. These networks consist of companies and suppliers with the goal of reducing costs and production time by branching various stages of the production process to third party foundries. Although the supply chain offers many advantages, it is also vulnerable to attacks at many different nodes along the pipeline. For Internet-of-Things (IoT) devices, this problem is exacerbated by firmware vulnerabilities, which influence the low-level control of the system hardware. Moreover, according to the National Vulnerability Database (NVD) the number of firmware vulnerabilities within IoT devices is rapidly increasing every year, making firmware vulnerabilities a cause for growing concern and magnifying the need to address emerging firmware vulnerabilities. In this paper we attempt to define and expand upon a class of firmware vulnerability that is characterized by the malicious configuration of power management integrated circuits (PMIC). We propose a firmware attack construction and deployment on power management IC (FANDEMIC), which consists of reverse engineering the binary running on bare-metal IoT devices made to function without operating systems, to identify the functions that interact with its PMIC. We demonstrate the possibility of directly altering the firmware binary to deliberately misconfigure the PMIC such that supply line voltages are altered, which could result in a variety of issues such as sensor data corruption, battery failure, and rapid aging and wear-out. We propose a workflow to reverse engineer the binary, using Ghidra and Python scripting, and provide two simple, but novel function matching algorithms. Furthermore, we highlight and discuss the potential consequences of PMIC attacks, which include battery degradation and failure, accelerated aging effects, and sensor data corruption. As a proof of concept we fully implement the proposed attack on real hardware using a bare-metal microcontroller and PMIC to demonstrate sensor data corruption. Finally, we discuss possible mitigation techniques, which include binary auditing and secure firmware updates.

I. Introduction

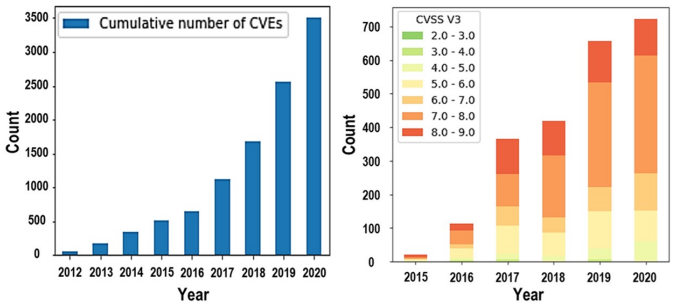
The increasing globalization of the Internet of Things (IoT) supply chain over the past several decades has resulted in major security concerns. As original equipment manufacturers (OEMs) further outsource production and assembly to external entities in order to keep production costs low and maintain a fast time-to-market, they have exposed themselves to potential attacks due to untrustworthy third party manufacturers [22]. These potential attacks include hardware trojans, design modifications, firmware modifications, and malware, all of which are difficult to detect by virtue of stealth design and the inherent difficulty of locating such alterations early in the supply chain.

In a recent exploit from December 2020, FireEye revealed a large scale software supply chain attack, now known as the SolarWinds hack, that was leveraged to compromise the U.S. Government and private companies like Microsoft and FireEye [7]. In this attack, adversaries breached SolarWinds' Orion IT management software and sent out malicious software updates to users, which were used to create back doors in user systems since as early as Spring 2020, thereby going undetected for months [7]. This attack has significant implications because the Orion platform is capable of sending firmware updates, and given that more than 33,000 of SolarWinds' clients use Orion [12], the high-impact nature of stealthy supply chain attacks similar to the SolarWinds hack cannot be understated, especially when considering device hardware.

Recently, the Department of Homeland Security (DHS) published a study where they highlight the need to advance security in the IoT ecosystem in order to provide a robust defense system against current and emerging threats [5]. The National Vulnerability Database (NVD) [25] report new firmware vulnerabilities being discovered daily, as illustrated in Fig. 1(a), and given the increase in the number of IoT devices [47] as well as the increase in the severity of the IoT firmware vulnerabilities over the years at an alarming rate [25], [40], as shown in Fig. 1(b), it is vital to expose firmware vulnerabilities and mitigate them before they become widespread problems and cause irreversible damage.

Unfortunately, the current supply chain model, where third party manufacturers and assemblers often have access to the firmware binary or even the source code for testing purposes, poses a significant threat. This is a major concern especially in resource constrained bare-metal IoT devices designed without operating systems, and can go unnoticed in the final stages of functional testing. In particular, IoT applications such as healthcare, public transportation, and environment monitoring pose significant security concerns due to the implications they have on human lives and their privacy.

In particular, we shift our focus to the increasing complexity of IoT devices. In order for a single device to accommodate more and more applications, a myriad of various components need to be incorporated into it, often requiring different power requirements. To handle this, system designers often make use of specially designed integrated circuits (ICs) called power management integrated circuits (PMICs) that manage power supply, power distribution, and battery health. It is critical for



(a) (b)

Fig. 1: (a) Number of CVEs related to IoT firmware from 2012 to 2020, and (b) Severity scores of these CVEs from 2015 to 2020 according to CVSS V3 reported in the NVD.

subsystem power requirements to be met, as even small deviations outside of the allowed ranges can result in a number of problems ranging from incorrect instruction execution to outright failure [15], [34].

For instance, over-voltage and under-voltage attacks can be deployed to disable protection circuits and cause incorrect instruction execution [34] using controlled transient effects. Given the critical role of the PMIC in modern IoT devices, there has been some consideration for targeting them in hardware attacks. Tencent Security demonstrated the possibility of exploiting PMIC charging controls to cause batteries to combust [30]. Herein, we expand upon the concept of PMIC attacks by proposing FANDEMIC, a firmware attack scheme that targets the power management IC (PMIC) of IoT devices running bare-metal systems. The reasoning for focusing on IoT devices that run on bare-metal firmware is that non bare-metal systems are equipped with operating systems capable to obfuscate the addresses needed by our current approach. An overview of the proposed FANDEMIC is illustrated in Fig. 2. To the best of our knowledge, this is the first time this vulnerability is exposed and the impacts are studied[1].

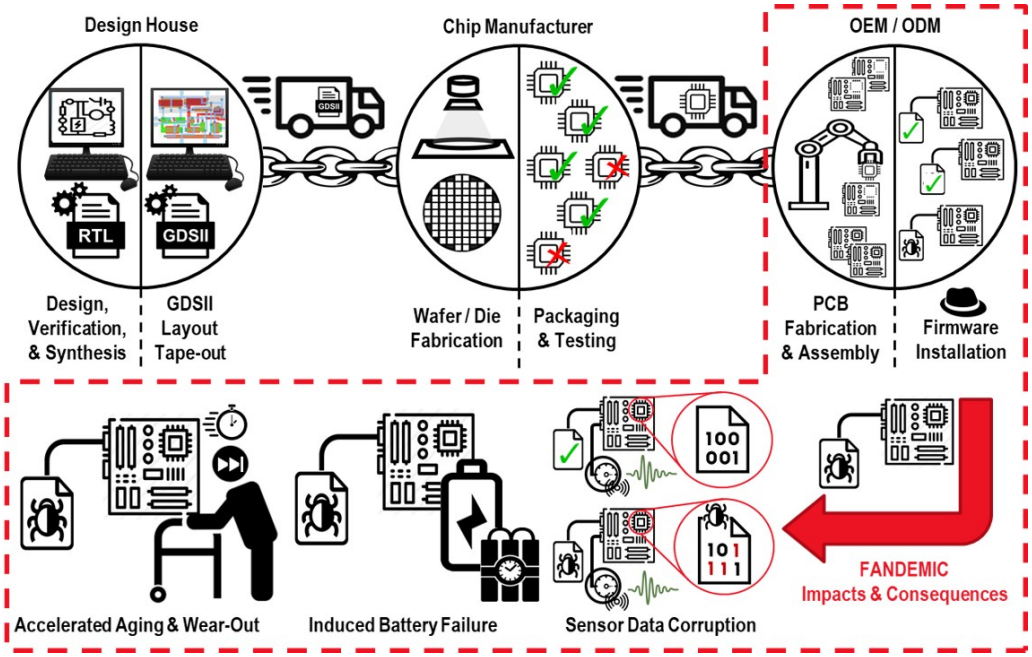


Fig. 2: Illustration of a simplified supply chain overview, in which we can see FANDEMIC would attack the supply chain at the last stage of production, which is the PCB fabrication & assembly, and firmware installation. FANDEMIC has the potential to cause accelerated aging and wear-out on logic circuits, induce battery failures, which could potentially harm users if the device itself is implantable (e.g., medical devices like pacemakers), and corrupt sensor data that could result in incorrect readings and potentially cause failures in the correct functionality of devices.

The contributions of our paper are as follows:

[1]We are in the process of reporting this vulnerability to the NVD and NIST

- Demonstrate a vulnerability in bare-metal IoT devices.
- Demonstrate how a firmware attack may change the operating voltage of a PMIC, disrupting the correct functionality of its host device.
- Propose a framework to reverse engineer a system's binary, construct, and deploy a payload using cross-correlation function matching and Ghidra tool.
- Propose mitigation techniques against the attack.

II. Background

Supply chains are constantly evolving, and they have reached the point where costs have been greatly minimized, but this comes with added risks (e.g., just-in-time supply chain). Additionally, supply chains are now distributed across different retailers to help reduce costs for both end users and businesses, but it also introduces more links in the network. Since "security is only as strong as the weakest link," in this section we give an overview of the supply chain system, its security challenges, and discuss possible attacks.

A. Power Management in IoT Devices

Nowadays, it is common for IoT devices to include a variety of sensors and components each with specific voltage needs to operate correctly [48], and PMICs have been widely used to manage the multiple voltages required. These ICs are solid state devices that control the flow and direction of electrical power, and common functionality includes DC-to-DC conversion, voltage scaling, and power-source selection. PMICs incorporate several functions (e.g., power conversion, under-voltage, etc.) into one chip, helping reduce the amount of space otherwise required and having better heat dissipation.

B. The Modern Supply Chain

Broadly speaking, there are six main stages in the supply chain: Design, Fabrication, Assembly, Distribution, Maintenance, and Disposal, [20] typically with testing stages associated with each of the manufacturing/production steps (Design, Fabrication, and Assembly, shown in Fig. 2). The design stage encompasses the design of the device: component selection, schematic design, netlisting, floorplanning, and all of the other steps that need to be taken before physical manufacturing can begin. The fabrication stage involves the processing of raw materials into the components that will be integrated into the final product. These components might be sensors, actuators, processors, printed circuit boards (PCBs), passive components, or even entire systems-on-chips (SoCs) that can come from a variety of different manufacturers, which might have supply chains of their own. The assembly stage is where all of these components are then integrated into what will become the final product. This is where components are soldered to the PCB, the firmware is loaded, and the device undergoes its final round of tests before being declared ready and shipped to its final destination. We consider assembly to be the final production stage, and the stage FANDEMIC would target, as shown in Fig. 2). Though it could be possible that the proposed malware would be vulnerable to detection if there was a testing stage after assembly, it would depend on the kind of testing performed. For example, logic verification tests likely won't be able to find it since they are only concerned with logic values. However, analog probing might, if test points are included in the design and the margins of error on supply voltages are tight. Following assembly, assuming the device has passed the final inspection, it enters the distribution stage in which finished products are distributed from foundries to retailers then to end users. The maintenance stage encompasses customer service, technical support, and repairs for the products the end user has received. Finally, the disposal stage refers to how devices that have reached their end-of-life are processed, recycled, or disposed of, though it is often disregarded by manufacturers and end users alike.

The electronics supply chain is driven by a "just-in-time" or "lean" manufacturing approach to production since this approach, when working smoothly, minimizes operating costs while maximizing production efficiency. Fundamentally, the idea behind the model is to reduce costs by minimizing inventory. Now if a component manufacturer is somehow unable to produce, then the entire process can get bottle-necked due to a lack of resources/inventory for the next stage, during which the company cannot sell its products [18]. This drives companies to turn to other manufacturers to get the parts they need to stabilize their supply of components [1]. This diversification of suppliers, combined with the widespread adoption of lean manufacturing practices, has opened up supply chains to malicious actors and untrusted foundries, who offer cheap services, but lack integrity.

C. Supply Chain Security

The primary goal of supply chain security is to provide trust and assurance for end devices, despite the presence of untrusted entities in the manufacturing process. [3], [20], [21], [39], [41]–[43], [46]. From a security engineering perspective, this boils down to defining system policies, implementing security mechanisms, specifying assurances, and providing incentives [29]. Practically speaking, it requires a deep understanding and defending against various forms of supply chain attacks. A supply chain attack is a malicious modification of a device that occurs at some point in the production stages of the supply chain. They can typically be classified as either a board-level attack, in which a physical modification is made, or a firmware attack, in which the device's firmware is altered. They can vary wildly in appearance, tactic, and severity, and can have many different possible goals, from stealing or tampering with data, to full-blown device failure.

D. Firmware Attacks

Firmware and hardware attacks are major threats to embedded systems. While hardware attacks require physical access to a device, and comparatively more effort than firmware attacks, the consequence of a hardware attack could be drastic. Unlike firmware vulnerabilities, that could be patched via software updates, patching of a hardware vulnerability would require re-manufacturing the hardware, costing a major loss to the vendor. On the other hand, firmware vulnerabilities are low hanging fruits that could be majorly targeted by attackers to create threatening exploits. FANDEMIC presents the exploitation of a firmware vulnerability to manipulate PMIC configuration and disrupt system operations.

Firmware interacts directly with hardware components and serves as a storage location for sensitive information. This low-level software component can become a single point of failure as it could lead to the compromise of an entire device. However, though firmware attacks can yield devastating consequences, research shows that firmware is commonly left unprotected. In a March 2021 report, Microsoft reported that 80% of 1,000 interviewed businesses experienced at least one firmware attack over the past two years, yet only about 30% allocated funds for firmware protection in their budgets [37]. This issue coupled with the challenge of verifying firmware integrity are important motivations for this work.

Given the nature of the modern supply chain network, firmware can be tampered with at multiple points along the network. One of the dangers of the supply chain when it comes to firmware is that firmware can be compromised even before a device is deployed in the market. Consumers must trust that devices are legitimate and contain legitimate firmware when they purchase them, but that cannot be guaranteed by device vendors. A malicious insider could potentially tamper with firmware, thereby compromising the integrity of the final product before it is even on the market. There are multiple attack vectors attackers can leverage to compromise firmware in the supply chain. The main vectors relevant to this work are as follows: 1) Physical Access; 2) Counterfeit Devices; 3) Over the Air Updates; and 4) Third Party Software Libraries.

A common practice in software development is the use of software libraries to facilitate the code writing process. Such libraries are publicly available, and some have known vulnerabilities, which is is a major issue [4]. A serious disadvantage of the supply chain is the magnification of vulnerabilities as they spread across the technology landscape. Furthermore, the proliferation of IoT devices in the past few years has come with an increasing attack surface. Just recently, researchers at JSOF, a boutique cyber consultancy firm, discovered a set of 19 zero-day vulnerabilities in a common low-level TCP/IP software library [13]. Collectively known as Ripple20, the vulnerabilities affect ”hundreds of millions” of IoT devices from a plethora of vendors [13]. The software library developed by Treck Inc. has a large footprint, so the naming of the vulnerabilities correlates with the ripple effect of a single vulnerable component spreading outward in the supply chain across vendors, fields, companies, etc.

A major challenge with firmware attacks is the challenge of detecting firmware modifications. As long as a device functions as intended it can be very difficult to detect a compromise [4]. In addition, hardware may not contain any protection for firmware, especially considering the growth of low-cost devices which may neglect security for lower cost. Still, the presence of firmware protection does not mean a device is secure [27], [28], though it is better to have some protection than no protection at all. Thus, one of the main goals of this work is to modify firmware without altering the overall function of a device, thereby making the attack less likely to detect. The scenario *”without altering the overall function”* considers that the device’s core functionality is not affected in any direct way. While there are some changes, since the device should retain its original behavior, it becomes harder to determine whether something is wrong during subsequent testing. For example, in the context of hardware testing, a digital logic verification stage that tests the communications channels present on the device, issues with the device would not be detected, since it is power lines that are being affected. Hence, less likely to be detected. The aim is to demonstrate the threat posed by supply chain firmware attacks.

E. Reverse Engineering and Ghidra

A major component of the proposed attack is binary reverse engineering. In our attack model (Subsection III-A) we assume the attacker has access to the raw firmware binary, but not the source code. The attacker aims to alter the original binary and produce a malicious version of it, which is a difficult problem because a raw binary is harder to alter directly as small changes may cause a big impact, potentially breaking the entire program. In order to make these changes successfully, the attacker needs to reverse engineer the binary to acquire a deep understanding of it, which is required to carry out subtle alterations so the attack is not easily detected. Software reverse engineering is a highly active field, where a significant number of useful tools used to extract information from raw binaries exist. For this paper we make heavy use of Ghidra to facilitate reverse engineering as it is capable of binary decompilation, disassembly, and static analysis [6], [23].

III. Proposed FANDEMIC

Our proposed supply chain attack consists of reverse engineering the target system’s firmware binary to identify the functions that control the power management integrated circuit component and use them to slightly alter its operating voltage. Once these functions have been identified, the attacker modifies the binary (i.e., patches the binary), and deploys it on the supply chain where it is unwittingly flashed to the target. The attacker is motivated to target the PMIC because it is well-known that

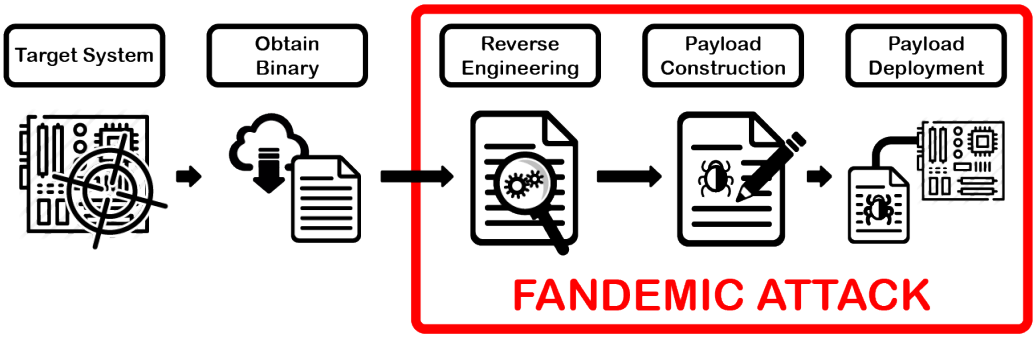


Fig. 3: Overview of the FANDEMIC attack flow. Before beginning the attack, the target device is chosen and original firmware is obtained. The FANDEMIC attack starts with reverse engineering the binary to determine PMIC configuration information. The next stage is firmware modification, and the final stage is firmware deployment.

operating devices at incorrect supply voltages can induce logic circuit wear-out, battery failure, and data corruption among other effects, all of which are uncommon effects that the attacker may want to exploit.

A. Attack Model and Implementation

The objective of our proposed FANDEMIC attack framework is to produce unintended/abnormal system behavior by manipulating a device's power management integrated circuit (PMIC). In theory, the attack on the PMIC could be carried out via hardware or firmware attack. A hardware attack with lasting effects past the context of the supply chain would require hardware modifications with very few degrees of freedom and is substantially dependent on the targeted device. Instead, FANDEMIC focuses on firmware as its attack surface, due to the advantage of the attack surface being less physically invasive and having more flexibility during implementation. A detailed outline of the attack can be found in the appendix in the form of an attack goal tree, and a high-level overview of the attack flow can be seen on Fig. 3

1) Assumptions: To appropriately evaluate the proposed attack, several assumptions about the system were made.

a) Attacker is an entity within the device supply chain: A major assumption that we make is that this attack occurs at the supply chain level, where the attacker is likely to be an employee of a 3rd party foundry or OEM involved in the device's production. In particular, the attacker is at one of the end stages of the supply chain, where the device hardware is already assembled and ready for firmware. In such a scenario, the attacker does not have any control over a specific device, but instead the attack can be targeted to all produced devices. In this case, the goal of the attacker is to cause harm to the vendor itself (e.g., economic damage). Alternatively, the attack could target a single device by injecting the malicious firmware during an insecure firmware update. The fact that the attack occurs within the supply chain will become justification for several other assumptions we make for this attack.

b) PMIC is present on target device: Trivially, the attack targets a device's PMIC, so it is only applicable to devices on which a PMIC is present, which is the case in many IoT devices, as the PMIC is necessary to manage power for a variety of sensors and actuators commonly used in such devices.

c) PMIC is external to device processor: We assume that the PMIC is a separate chip from the main processor, this assumption is not strictly necessary depending on the circumstances. It is becoming common for processors to be integrated with peripherals onto a single chip, called a System-on-Chip (SoC). However, despite the recent shift towards SoCs, many OEMs continue to design devices in a modular fashion, so companies like MAXIM Integrated still heavily market PMICs as isolated chips. Furthermore, our approach does not necessarily preclude one from conducting a similar attack against an integrated processor-power management unit (CPU-PMU), as the integrated PMU still needs to be configured and programmed. Hence, though some assumptions might require alteration, the PMIC attack can be adapted to work against even integrated SoCs. However, that is not the focus of this work.

d) PMIC is digitally configurable via chip-to-chip communication protocol: This class of attack targets PMICs of considerable sophistication that are becoming more typical in consumer IoT devices like smartwatches. Such PMICs need to have an interface that the processor can use to set power configurations for the device. We assume that this interface is a standard chip-to-chip communication protocol like I2C or SPI. This assumption might be generalized to encompass any digital logic interface, but we constrain it as a demonstration.

e) PMIC and Processor's datasheets and reference manuals are publicly available: In order to reverse engineer and alter the firmware binary, the information pertinent to programming both the PMIC and processor is needed. This assumption holds in the vast majority of cases because any consumer-available component must have a datasheet for customer reference, as it is necessary for circuit designers to design the device in the first place.

f) IoT device is running on bare-metal: Our attack targets processors with bare-metal firmware, so we assume there is no higher-level operating system above hardware abstraction layers like development libraries and APIs. This sets our attack apart from typical IoT firmware attacks that aim to exploit OS vulnerabilities, though the PMIC attack can possibly be adapted to target the hardware abstraction layer that most operating systems are implemented on top of.

g) Malicious binary is the initial flash: Once a malicious payload is constructed, the attacker needs to be able to deploy it. In our model, the attack occurs at the end stages of the supply chain where the device will first receive its firmware. We assume that the attacker (the untrusted third party OEM) is responsible for the initial flash and has the means to flash the malicious binary instead of the intended original. This ensures that the payload can be deployed.

It is also possible to deploy the firmware after the initial flash in cases where:

1) No firmware integrity or source authentication checks are implemented in the system.
2) When integrity or authenticity checks are implemented, but vulnerable to exploit.

It must be noted that if the integrity and authenticity validation has a robust implementation, it becomes harder for the attacker to exploit the vulnerability.

h) Binary access has been achieved: Access to the original firmware is given. It may be due to the firmware being provided by the creator, or it was accessed due to a vulnerability in the system.

More assumptions may become necessary depending on the device and the approach taken to reverse engineer. In our implementation of the attack, for example, we will make several other assumptions (described in Section III-B2) with respect to the presence of manufacturer-provided APIs.

2) Attack Stages: In order to satisfy assumptions, we implicitly have an initial Target Selection Stage, during which hardware information is gathered about the target device and the original firmware binary is obtained. In particular, the target processor architecture and the target PMIC model needs to be determined. This initial stage also encompasses a collection of any hardware or software tools the attacker might need for the attack development (e.g., firmware libraries, development and evaluation kits, test devices, etc.).

The attack consist of 3 main stages:

1) Firmware Reverse Engineering: program behavior is determined from the firmware binary
2) Binary Rewriting and Payload Construction: malicious binary is constructed by modifying original firmware
3) Payload Deployment: malicious binary is uploaded to device

These stages will be further detailed in following sections.

3) Attack Platform and Setup: To implement this attack, the Nordic nRF52832 microcontroller (MCU) and MAX20303 power management IC (PMIC) were selected. The configuration of the PMIC is controlled by the MCU via an I2C bus. We test our reverse engineering methods on three sample binaries created within our group that have different purposes: communication with the PMIC, data collection from an MPX2010 pressure sensor, and a smartwatch application. As mentioned, we make use of Ghidra for reverse engineering, as well as a number of custom scripts and tools written in Python.

4) PMIC Configuration: The configuration process of the PMIC plays a major role in determining how to go about reverse engineering. Since the configuration process is defined by the PMIC manufacturer, there will be a standardized method for interfacing with the unit, that is likely to yield certain patterns in the firmware like I2C reads and writes. For the MAX20303 PMIC used in our implementation, in order to configure the PMIC, the processor must first send a command identification value to the PMIC. It tells the PMIC what function, like the buck or boost converter, will be configured. Then the processor sends 4 pairs of values that specify the address of the register to write to, as well as the value to write. After finishing the transmission, the PMIC responds with an ACK bit that confirms the success of the transmission, and the received values will be written to its internal registers. Because this process is well defined, the firmware author must follow this convention and is therefore likely to hard-code configuration values with some spatial locality that will be reflected in the binary, which can be exploited during reverse engineering. I2C library functions must also be used, which can be leveraged to narrow down the possible areas of the binary where configuration happens.

B. Reverse Engineering

Once access to the firmware binary is achieved, the next step is to reverse engineer it. Per the objective of FANDEMIC, we need to reverse the binary to identify the functions responsible for performing data transactions with the PMIC chip. The identification of such functions would help trace the exact registers, instructions, and addresses where the binary can be modified to change PMIC behavior. To achieve this, we attempt a variety of methods including static analysis, function matching, and symbolic execution, ultimately succeeding in our implementation using function matching. Note that there many ways in which one can go about reverse engineering a given binary for the information we are looking for, not all of which will work. As the saying goes, ”there is no silver bullet.” In this work we highlight the method that worked for us, binary function matching, which we believe to be flexible enough to be adapted to other platforms, despite its assumptions. Section III-B3 outlines additional methods for function matching that did not work in our test cases, but we believe have potential with further effort or in other scenarios.

1) Static Analysis: In static analysis, the structure of the binary is examined without runtime information. While there are many tools available that can automate portions of static analysis, there are too many variables to consider for all stages of analysis to be fully automated, because of this a considerable manual effort is needed as well. Moreover, since the attacker's goal is to write a malicious version of the firmware, it is in their best interest not to rely too heavily on automated analysis, as the tools can make assumptions that might be invalid. Nonetheless, to the extent possible, analysis tools should be used where convenient.

To perform static analysis, we make use of Ghidra. Ghidra is unable to do out-of-the-box analysis of the binary without additional information like the instruction set architecture (ISA) and device memory map, which can be obtained from the processor datasheet. The datasheet contains useful information like the address of the reset handler, address of flash memory, addresses corresponding to peripherals and corresponding register configurations, which can also be given to Ghidra to enable more detailed manual analysis. Once the necessary information is provided, Ghidra performs some basic, semi-automated static analysis, disassembling the binary, locating functions and basic blocks, and generating cross references and control flow graphs. The disassembly and cross references are crucial for the manual static analysis involved in reconstructing the control flow and identifying instruction patterns.

At this point, it is useful to identify the main function's location, as it initiates all operations in bare metal firmwares. The details of how this is done will vary depending on the microcontroller's architecture, but since all microcontrollers must be able to execute the main function the steps will be similar to what is done on the Cortex-M4 architecture, which we used in our implementation. The first step is to locate the Reset handler in the binary. According to the Cortex-M4 Generic User Guide, the Reset Handler address is the second entry of the Interrupt Vector Table, which is located at the start of the binary by default [19]. As the Reset Handler is implemented by the compiler, we compile a blank firmware binary to analyze what the Reset Handler does by inspecting the objdump of the ELF file. The Reset Handler itself calls another built-in function: `mainCRTStartup` which does device initialization, followed by invoking the user main function. As this function is implemented by the compiler, it is generated algorithmically, and thus can be relied upon to locate the address of the main function.

Having identified the main function, the next step is to identify the function calls responsible for communication with the PMIC. The function call graph is generated in Ghidra for visualization. In the call graph, we can observed what the main function (i.e, FUN00002a6c) is internally leading to, and further analyzed.

Since Ghidra's cross references work in both directions, we are able to determine where these function calls are by locating the function definition in the binary. By assuming that these function calls are made to library functions provided by the device manufacturer, we can locate these definitions using binary source code function matching techniques.

2) Function Matching: We are able to determine configuration information using binary similarity-based function matching if we make certain assumptions about the target firmware: (1) the PMIC's configuration information is hard-coded in the firmware, (2) the PMIC is configured using a chip-to-chip communication protocol like I2C or SPI, (3) firmware relies on manufacturer-provided libraries for chip-to-chip communication, (4) these libraries are open source, (5) an attacker can compile libraries on any available OS and with any possible optimization. Note that although we have now accumulated a number of assumptions, most of these represent the common case in embedded system development and should not disqualify our methods.

With these assumptions, we created a representative firmware binary for the same target processor and forced the inclusion of the chip communication library functions at compile time. Since we created this firmware ourselves, we can view the unstripped ELF file with objdump and use the symbol information to locate critical functions that implement data transfers. As an example, on the nRF52, I2C communications are initiated with one of two library functions: *nrfx_twi_xfer* or *nrfx_twim_xfer*, depending on the way the firmware is configured. After obtaining the offset and length of the function definition, we then extract the function binary for comparison against the target firmware. To carry out this comparison, we perform two simple matching algorithms based on cross-correlation to locate the function in the firmware's binary. Note that this is only possible if the compiled binary in the firmware is considerably similar to the binary that we compiled ourselves. We assert that the 5th assumption stated above will ensure a reasonable likelihood that this is the case, as the firmware is likely to have been compiled by an accessible OS and optimization level. There may be many possible permutations of compilation platforms and flags, but these can be explored fully via grid search provided sufficient time. The offset of the peak value of the cross correlation between the firmware binary and function binary, treated as signals, shows the location of the function in the firmware binary. Once this information is obtained, the function can be located in Ghidra and cross references to the function address can be used as starting points to manually back-trace to the hard-coded configuration values.

We have implemented 2 function matching algorithms in Python that use binary cross correlation and instruction cross correlation respectively. In the former, we treat the binaries as non-return-to-zero (NRZ) signals, converting 0's and 1's to -1's and +1's respectively. In this case, the correlation is defined as

$$Corr(f, g)[n] = \sum f[k]g[k + 8n]$$

where f ang g are NRZ signals and the resulting correlation is an array. This formulation can be thought of as a rolling dot product. Note that since we are working with byte-aligned firmware binaries, the shift value is $8n$ in order to keep the resulting correlation byte-aligned. If the lengths of the signals are mismatched, the shorter one is zero-padded so that it will not interfere with the mathematics of the correlation. By design, high values in the cross correlation will denote regions of high similarity with the searched-for function [35], with a peak value equivalent to the length of the shorter signal if present verbatim in the longer signal. Thus, if the peak value of the cross correlation is some high fraction of the length of the function binary, we can reasonably conclude that the function is probably defined in the firmware with an offset proportional to the argmax of the cross correlation.

In the *instruction-based implementation*, the same principle is applied, but with disassembled instruction mnemonics instead. The specific implementation presented in this paper might not work in some cases, but serves as a demonstration. In this method we use strings rather than signals, and redefine the inner product as the sum of element-wise comparisons, in which the comparison yields +1 if the strings are the same, -1 if they are different, and 0 if either string is empty.

$$Comp(s_1, s_2) = \begin{cases} +1 & \text{if } s_1 = s_2 \\ -1 & \text{if } s_1 \neq s_2 \\ 0 & \text{if } s_1 \text{ or } s_2 \text{ is empty} \end{cases}$$

$$Corr(f, g)[n] = \sum Comp(s_1[k], s_2[k + n])$$

The shift n refers to the instruction offset. Since we are mostly working with Thumb 2 instructions, for convenience, we assume that all instructions are 2 bytes long, and n correspond to 2-byte shifts. Hence any peak index given by n would actually correspond the byte offset $2n$. We are aware that there are inaccuracies introduced, because this assumption is technically invalid when applied over the entirety of the firmware, however, when most of the firmware consists of Thumb 2 instructions, because of the mathematics of cross correlation, the inaccuracies of the blanket application of 2-byte translation will not be significant, as the summation over all the matches should still result in a peak because of self-correlation when the function is present in the target firmware. The argument for using this formulation of cross correlation is that different compilers might optimize operands differently (for instance r7 instead of r8) while the instruction opcodes might be the same. In our implementation, since only operation mnemonics (which correspond directly to opcodes) are considered, differences in operands will not cause the correlation to drop as it would with cross correlation of binaries directly.

Note that neither of these methods for function matching have been tested exhaustively and only are crude attempts at function matching. Function binary and source code matching is an active area of research with many more advanced techniques than those presented here. Sophisticated techniques can make use of neural networks [45], control flow graph and call graph analysis, as well as better techniques for computing binary similarity than cross-correlation [14], [17]. Nonetheless, the aforementioned methods implemented here will work provided the specified assumptions are satisfied as is the case in our examples and have the advantage of being simple to use and understand.

We employed these methods on 2 of our firmware examples (sensor reading and smartwatch) and found the function locations at 0x21AC and 0xC6E4 for the sensor firmware and smartwatch firmware respectively. The results can be seen in Figure 4, where the peaks represent the function of interest.

Once the binary function definitions have been located in the firmware, we return to Ghidra and directly analyze their offsets. As we have already located the main function, Ghidra is able to disassemble the functions of interest and generated cross references to calls from the main. We then go through these cross references one by one and manually backtrace the calling parameters for values that might be used to configure the PMIC. These values should be documented in the PMIC's datasheet due to having some significance for configuration, and search functions can be used to help locate them. Note again that this process is applicable only if the values are actually hard-coded in the binary and not obfuscated, though we believe this to be true in many cases. Once done with this process, and having found the configuration values to modify, the attack moves into the next stage: binary rewriting and payload construction.

3) Alternative Approaches:

a) Cross Referencing Peripherals: Besides the previously mentioned approach, an alternative approach is to cross reference memory-mapped peripheral addresses directly with Ghidra. Referring to the processor memory map can help locate functions responsible for peripheral control. In bare metal firmware, the processor interfaces with peripherals by reading and writing to a set of memory addresses associated with a given peripheral. These memory locations are specified in the processor datasheet and can be loaded into Ghidra for cross referencing [32]. Loading the memory map to Ghidra can be performed manually or automated via SVD-loader [36] or other Python or Java script.

Once the memory map is loaded into Ghidra, we examine all cross references made to memory addresses associated with the communication peripheral used to configure the PMIC. In our proof of concept, we target the MAX20303 PMIC chip, which the MCU configures via I2C. Thus, we examine the set of addresses associated with the Two-Wire Interface (TWI)

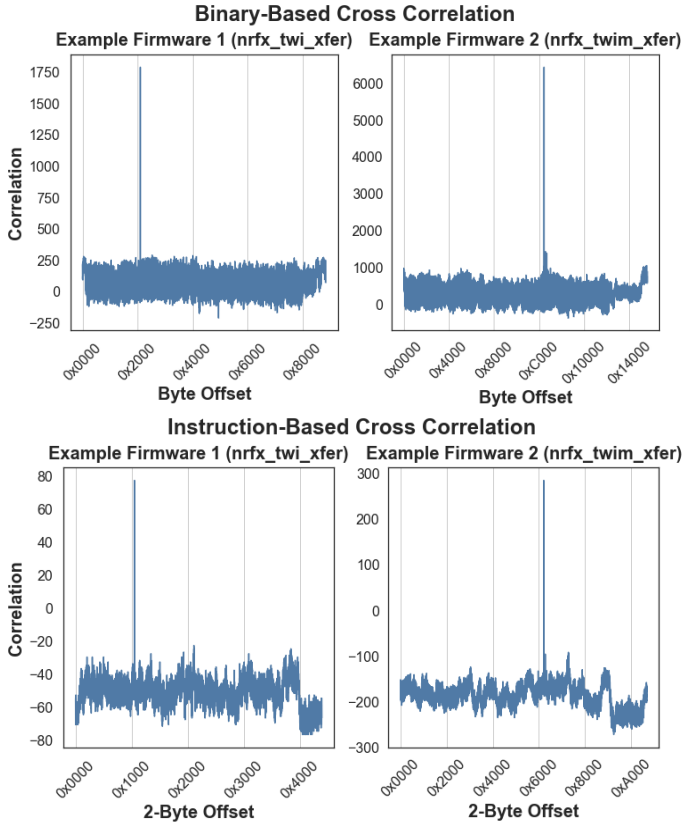


Fig. 4: Cross Correlations of Function Binaries and Instructions for 2 firmware examples using different library functions for I2C communication. The correct byte offset should be 0x21AC and 0xC6E4 in firmware 1 and firmware 2 respetively.

peripheral that the nRF52 uses to implement I2C transfers. Referring to the datasheet, we find that the TWI peripherals are mapped to 4 KiB memory blocks with base addresses at `0x40003000` and `0x40004000`.

Once these addresses are loaded to the memory map, Ghidra automatically generates cross references from the previously disassembled instructions if an instruction directly references one of the mapped addresses. However, if the peripheral addresses are not directly referenced— for instance if the address is accessed via a pointer stored on stack, or offset from a register— then Ghidra will not be able to locate peripheral functions and another method must be employed. Another potential issue is that simply locating the function that interfaces with the peripheral may not reveal the configuration information, which was the case in our implementation.

Though we were able to locate the function that accessed the peripheral addresses, this function turned out to be an Interrupt Handler invoked asynchronously at runtime. Because of this, it becomes prohibitively difficult to backtrace parameters statically. Though library functions were used to initiate I2C transfer and would have conceivably been invoked from the main function, these API calls relied on the interrupt handler to actually interact with the peripherals. Hence, we were able to locate this handler, but were unable to locate the API calls that passed the PMIC configuration information. While we were unsuccessful with this method, we have included it in our attack description nonetheless as it may have proven effective in other scenarios. Our test case only represents one of many possible combinations of processor, library, and PMIC, which may implement communication in different ways. If another processor's API were to implement communication with direct peripheral accesses, this method may have been more useful.

b) Dynamic Analysis: If hardware is available to test on, dynamic analysis may be helpful for determining configuration information that exists at runtime. Using the GNU Debugger on an actual processor allows us to track stack changes and other dynamic information that can help isolate the sections of the code that control chip communication or configuration. Of the outlined approaches, a full dynamic analysis is the hardest to conduct due to the necessary hardware dependencies that live debugging requires. In a real-life scenario, an attacker may not have access to hardware debug ports, or have access to hardware at all. Dynamic analysis might be conducted via full system emulation with frameworks like Renode or Unicorn, but such analysis is out of the scope of this paper and left for future research.

C. Binary Rewriting and Payload Construction

During the Reverse Engineering stage the location of any relevant configuration values should have been determined. At this point our goal is now to modify these values. Before the attack can be carried out, we need to determine the binary's format as the final format will determine the best approach to modifying it. A raw binary might call for a simple hex editor like Rizin, an ELF file might be best patched in Ghidra, and an Intel Hex file might be easiest to edit directly in a text editor. In our example, the nRF52 is programmed with Intel Hex files, so the easiest method to produce a payload with that format is described in the following subsection.

1) Rewriting Intel Hex Files: The Intel Hex (IHex) file format stores the binary in ASCII form, where bytes take on their hexadecimal representation. Each line of the file is called a "HEX record" and has 5 fields: the first 2 hex characters encode the number of data bytes in the record, the next 4 characters represent the starting address of the record's data, then another 2 characters that indicate the record type, followed by the data field. The last field is the record's checksum, which is the two's complement of the 2 lowest bytes of the sum of all the preceding hex digit pairs [11]. As indicated by the type field, there are records of different types that carry additional information that the processor might use on startup, like the starting address and segment information. Most reverse engineering tools available are capable of writing Intel Hex files since the format is straightforward, however, there are often small variations in the exact way the binary is written that can result in significant differences when comparing the modified version to the original. Most significantly, since each record has a data field, record lengths can vary, resulting in very different IHex files, even if the binaries they encode are the same. Thus, in order to avoid unnecessary differences, it is more simple to edit the original binary directly using any standard programmer's text editor, since the format encodes the binary as ASCII automatically. To modify an IHex firmware, the attacker needs to change the data at the located addresses, recompute the checksum for each modified record, and overwrite the old one. Directly overwriting values minimizes the number of modified bytes, since record data lengths are preserved. Manually editing IHex files can be tedious, therefore a Python script was written to make such in-place data edits, without modifying record lengths the way most other IHex editing tools do. Additionally, carrying out the edits in this manner makes the modification portion of the attack script-able, such that given the offsets to modify, and the values to write, the script is able to make the edits automatically, and multiple firmware versions can be generated quickly.

D. Payload Deployment

Deploying the payload is the final stage of the attack. At this point, the attacker needs to load the firmware onto the target device, for which very few options are viable. Fortunately (or unfortunately) in the context of the supply chain, it should be relatively easy for an attacker at the final stages of the device's production to have access to these options along with the initial firmware flash, yielding the highest likelihood of success since on-board security measures are not likely to be active yet, and the foundry is likely to have the equipment necessary for loading the binary on hand. Following that, depending on the device's original purpose, other methods may be employed.

1) Initial Flash: In our example, we assume the attacker has access to the necessary tools to carry out the attack as previously mentioned. Hence, we deploy the modified firmware by following the normal procedures for flashing firmware, using the modified Intel Hex file instead.

2) Runtime Updates: Another approach to deliver the payload is taking advantage of firmware updates. Some devices are capable of runtime firmware updates in which new firmware can be loaded over-the-air via Bluetooth or Wifi, or via traditional wired methods. If such a capability is in place, the attacker may be able to exploit these capabilities to deliver the malicious firmware, if the update protocol is insecure, or if the OEM has update authentication credentials. Though we do not implement these in this project, other researchers have explored this area [4], and we believe it would be applicable to the proposed attack as well.

3) Flash Memory Attack: Firmware are stored either in external or the internal flash memory. If an attacker has physical access to the device, the firmware could be extracted from memory, maliciously patched and flashed back. In the case when firmware is stored in external flash memory, the attacker can deploy the payload by modifying the data of the external flash memory if it is not write protected. In the case of internal flash memory, if the device has accessible debug ports like JTAG or SWD, the attacker could exploit that to get access to internal flash memory and deploy the payload.

4) Firmware implementation bug exploitation: In some cases, attackers exploit the existing implementation bugs in the firmware to deploy the payload. These bugs include memory bugs, authentication bypass, vulnerability in the third party library, web/mobile application bugs, input sanitization bugs, and similar other flaws that can be exploited to perform attacks like cross site scripting (XSS), buffer overflow, and remote code execution to deploy the malicious payload.

IV. Results

A. Sensor Firmware Example

In order to verify the effects of the attack, we implement a simple sensor application using the nRF52832 microcontroller, MAX20303 PMIC, a MPX2010DP ratiometric pressure sensor, and LMC6084 operational amplifier. The circuit diagram is

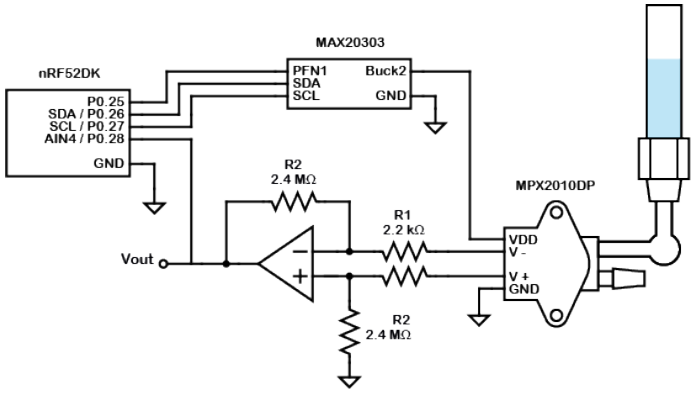


Fig. 5: Basic Hardware Schematic for Example Sensor Application. Only shows used pins on the nRF52 and MAX20303.

```
sensor_2-9V.hex
0000 BBE0: 38 30 30 30 30 43 33 0D  0A 3A 31 30 34 32 44 30  80000C3. .:1042D0
0000 BBF0: 30 30 33 34 38 36 30 30  30 30 34 38 38 36 30 30  00348600 00488600
0000 BC00: 30 30 30 30 32 41 32 36  30 31 35 43 38 36 30 30  00002A26 015C8600
0000 BC10: 30 30 32 33 0D 0A 3A 31  30 34 32 45 30 30 30 32  0023..:1 042E0002
0000 BC20: 38 30 30 30 30 32 30 35  30 38 37 30 30 30 30 41  80000205 0870000A
0000 BC30: 34 38 32 30 30 30 30 38  43 38 32 30 30 30 30 37  48200008 C8200007
0000 BC40: 42 0D 0A 3A 31 30 34 32  46 30 30 30 32 43 38 38  B..:1042 F0002C88
0000 BC50: 30 30 30 30 41 34 30 46  30 30 32 30 30 38 38 38  0000A40F 00200888
0000 BC60: 30 30 30 30 32 34 30 30  30 30 32 30 36 33 0D 0A  00002400 002063..
sensor_3-9V_hack.hex
0000 BBE0: 38 30 30 30 30 43 33 0D  0A 3A 31 30 34 32 44 30  80000C3. .:1042D0
0000 BBF0: 30 30 33 34 38 36 30 30  30 30 34 38 38 36 30 30  00348600 00488600
0000 BC00: 30 30 30 30 33 41 32 36  30 31 35 43 38 36 30 30  00003A26 015C8600
0000 BC10: 30 30 31 33 0D 0A 3A 31  30 34 32 45 30 30 30 32  0013..:1 042E0002
0000 BC20: 38 30 30 30 30 32 30 35  30 38 37 30 30 30 30 41  80000205 0870000A
0000 BC30: 34 38 32 30 30 30 30 38  43 38 32 30 30 30 30 37  48200008 C8200007
0000 BC40: 42 0D 0A 3A 31 30 34 32  46 30 30 30 32 43 38 38  B..:1042 F0002C88
0000 BC50: 30 30 30 30 41 34 30 46  30 30 32 30 30 38 38 38  0000A40F 00200888
0000 BC60: 30 30 30 30 32 34 30 30  30 30 32 30 36 33 0D 0A  00002400 002063..
```

Fig. 6: Byte difference in the Intel Hex files for the original firmware (above) with Buck2 set at 2.9V, and the modified version (below) with Buck2 set at 3.9V. First difference changes the configuration, second difference is the checksum for the record.

shown in Figure 5. The nRF52 connects pins 26 and 27 to the PMIC as the I2C controller, and GPIO pin 25 to PFN1, which needs to be brought low to enable configuration of the PMIC. Pin 28 on the nRF52 is an analog input pin that is used to read the output of the sensor circuit as a digitized voltage. The MPX2010 is a ratiometric, differential pressure sensor with a normal operating voltage of 10V, but can be supplied with less if the static sensitivity is adjusted in pressure calculation to take this into account. Since the raw voltage output of the sensor would be far too small for the nRF52 to reliably detect, the differential gain stage is added after the pressure sensor to amplify the output to a detectable range. The gain is around 1068, chosen based only on the available on-hand components. Though the PMIC would normally use a battery as it's main power source, we instead attach a power supply unit that provides a constant 4V. This is enough to supply the 2nd buck converter on the PMIC (Buck2), which has an output range of 0.8V to 3.95V, and a configuration resolution of 50mV.

The original firmware configures the Buck2 supply pin of the PMIC to output 2.9V, then begins collecting data from the ADC pin. The collected voltage data is converted internally to pressure using the following equation:

$$P(V_{out}) = \frac{V_{out}}{GK_{2.9V}}$$

where G is the gain and $K_{2.9}$ is the static sensitivity of the pressure sensor at 2.9V, calculated as:

$$K_{2.9V} = \left(2.5\frac{mV}{kPa}\right)\left(\frac{2.9V}{10V}\right) = 0.725\frac{mV}{kPa}$$

A button interrupt can pause and resume data collection. Our goal in implementing this attack is to reverse engineer the resulting binary, modify it without using the original C source code, then show that the modified binary can be flashed to hardware and affect the function of the application. The steps for the attack have been described in previous Sections III-B, III-C, and III-D. Following the processes outlined, we reverse engineered the binary to locate the Buck2 configuration information, which was hard-coded as an array. We then modify the Buck2 configuration bytes and the checksum for the corresponding record, which can be seen in Figure 6. This process is repeated several times with the supply set at multiple different voltages, producing new firmware files with our script and re-flashing the firmware each time to examine the resulting changes in supply voltage and sensor output. This data is summarized in Table I and plotted in Figure 7. The pressure being measured is the barometric water pressure at the bottom of a small manometer. A diagram can be seen in Fig. 7.

Config	V_{Supply}	$D[V_{Out}]$	V_{Out}	Pressure (Pa)	% Error
0x10	1.60	545	0.48	618.44	44.04
0x15	1.85	635	0.56	720.99	34.77
0x1C	2.20	710	0.62	805.54	27.12
0x22	2.50	810	0.71	919.40	16.81
0x26	2.70	941	0.83	1068.09	3.36
0x2A	2.90	974	0.86	1105.23	0.00
0x2C	3.00	1043	0.92	1184.12	7.14
0x32	3.30	1161	1.02	1317.43	19.20
0x37	3.55	1211	1.06	1374.38	24.35
0x3D	3.85	1371	1.20	1556.19	40.80

TABLE I: Table of sensing stage outputs averaged over 100 samples with a sampling period of 100ms. The original setting and reading is highlighted in yellow.

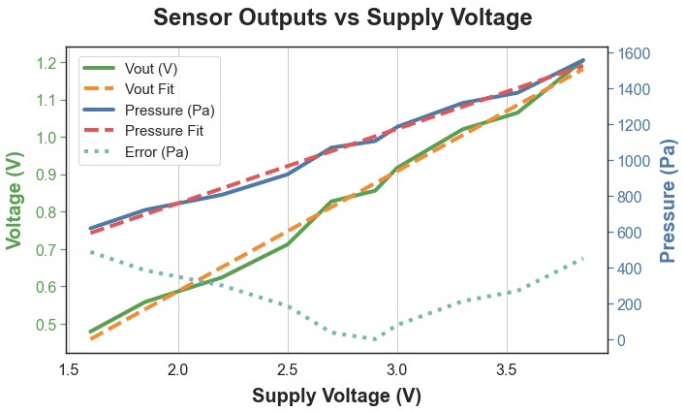


Fig. 7: Sensor output voltage, pressure reading, and pressure error relative to the original voltage of 2.9V (arbitrary), all as a function of the supply voltage. The slope of the dashed red fit regression line is 414.73 Pa/V and the slope of the dashed orange regression line is 0.321 V/V.

V. Discussion

The usage of a PMIC chip to manage power in various range of devices could be a potential attack vector. It could be prone to both hardware and firmware based attacks. The proposed FANDEMIC attack clearly highlights the affects of a successful PMIC attack on device operation and behavior. The bad power attack by researchers from Tencent Security Xuanwu Lab [16] as mentioned in the article [30] shows the threatening and major impact on the fast charging feature by maliciously patching the PMIC firmware. Apart from physical damage, the malicious firmware might even adversely change system functioning, insert malware for remote access, and many other possibilities to indirectly affect the major parts of the system. There are many factors that can lead to a successful attack on a PMIC including the system vulnerability to flash firmware without verification, improper data filtering, insecure firmware updates, lack of verification in supply chain attacks, hardware attacks, trojan insertion and various other factors. The proposed attack can have significant social and economical impacts as well as national security threats. As an example we mentioned the SolarWinds attack, which had massive implications for the US economy and national security. Notably, we consider that it is possible to send messages to the PMIC via either a side-channel or sensor readings after a device is deployed depending on the target and whether the attacker can influence the target post-deployment. However, it may be more convenient for the attacker to attack at a supply chain stage in which the device is currently being handled by a third party. Depending on the stage of attack, whether it is within the supply chain or after deployment on field, the effort required to mitigate such attacks would vary.

A. Attack Objectives and Consequences

Apart from a firmware based attack as proposed in FANDEMIC, attacks on a PMIC could also be hardware based with different objectives like manipulation of sensor signal, performing a denial of service (DoS) attack, system failure, physical damage of the complete system associated with PMIC, inserting malware to infect the connected parts, and other different objectives. The below subsections discuss about some of them.

1) Sensor Data Corruption: Many sensing systems are often dependent on a stable, predictable supply voltage for correct operation. Signals generated by passive sensing components like thermocouples, resistance temperature detectors (RTDs), and

strain gauges are often either far too small to be useful without a gain stage, or dependent on a powered circuit, both of whose outputs are heavily influenced by supply voltage. Ratiometric sensors like the MPX2010 pressure sensor used in our example are especially susceptible to supply voltage, as their outputs are directly proportional to supply voltage. As a result, a PMIC attack could potentially cause data corruption by altering sensor power supplies, which many sensing systems expect to remain fixed. From our example attack in Figure 7, we can calculate an approximation for the error rate with respect to the original reading as follows:

$$\begin{aligned} \%\text{ Error Rate} &= \Big(\frac{\Delta\text{Pressure Reading}}{\Delta\text{Supply Voltage}}\Big)\Big(\frac{1}{\text{Expected Pressure}}\Big) \\ &= \frac{414.73\ {}^{Pa}/_{V}}{1105.23\ Pa} = 37.52\ {}^{\%}/_{V} \end{aligned}$$

This gives an error rate of about 3.75 % relative to the pressure reading at 2.9 V for every change in 0.1 V of the supply voltage. This is a huge amount for such a small change in supply voltage. If such a system were used in a safety critical system, this attack could have potentially devastating consequences due to the apparent corruption of data.

While this issue might be corrected if the controller also takes the supply voltage into account when calculating the final measurement, this would require a physical connection from the supply to an analog-to-digital converter (ADC), to the controller. Since a MCU's on-board ADC is unlikely to tolerate higher supply voltages, an external one would need to be introduced, occupying valuable floor space on the device PCB.

2) Logic Circuit Wear-Out: Logic circuit wear-out usually happens around a well defined stage of life of circuits, in which their failure rate increases due to the aging of components, which causes critical sections of the device to be worn-out. This wear-out is caused by a function of the stress the components are subject to, and the time they are exposed to it. Fig. 8 illustrates how a normal life cycle of a circuit would be, along with how FANDEMIC would affect it. The solid blue line describes the random failures that happen across the life cycle of the component, which is constant across time. The red solid line shows the normal behavior of the expected life of an IC, in which the failure rate is higher at the early life due to early failures (green solid line). As the IC enters the useful stage it remains constant due to the early failures dying off at the early stage. Towards the end-of-life stage, the failure rate of the IC starts increasing again due to wear-out failures (brown solid line). On the other hand, the red dashed line shows the observed failures due to FANDEMIC, in which the stress causes wear-out failures (gold dashed line) in the IC as early as the end of the early life stage.

3) Induced Battery Failure: Most mobile IoT devices will require a battery to provide power to the system when other sources are unavailable. Because of their high energy density, performance, and lack of memory effect, lithium-ion batteries are one of the most popular options for electronic devices [44]. However, they are particularly sensitive to overcharge conditions, in which the battery is fully charged, but continues to receive current past that point. Under such conditions, the materials in the battery begin to deteriorate, causing performance and lifetime degradation. In extreme cases, material degradation can result in thermal runaway, in which the mechanisms for self heating in the battery outpace thermal dissipation and ultimately result in explosion [44]. Because of this, IoT system engineers need to pay careful attention to the battery charging circuits in their devices. Often, PMICs like the MAX20303 that we used incorporate charging mechanisms internally, with different charging configurations available. Thus, a PMIC attack can potentially cause major damage over time if the battery charging mechanisms are targeted, at minimum degrading lifetime and performance, and at worst inducing thermal runaway if overcharge conditions are created.

The threat of exploding batteries cannot be understated. Exploding batteries can cause thermal and chemical burns and represent a major concern for public safety [33]. Moreover, faulty devices can cost companies billions of dollars in lost capital, as was the case for Samsung in 2017 with the recall of the Galaxy Note 7 [24]. As such, steps ought to be taken to provide assurances against improper charging.

B. Attack Transferability

The attack in this work is intentionally presented in a general way so that it can be applied to other vulnerable bare-metal devices. Studies on firmware attacks have demonstrated that entire families of devices across different instruction set architectures can be affected by the same modification attacks as they often contain the same fundamental flaws [4], [28]. As such, one can conceivably construct a PMIC attack against other MCUs and PMICs if the other assumptions are satisfied, provided that the attacker has sufficient knowledge of the target system. In particular, since this attack ultimately boils down to making a malicious I2C transmission, any MCU capable of sending such a transmission can be targeted, though each binary needs to be tailored to that MCU's particular architecture. In fact, an attack might even install an entirely separate controller on the I2C bus as a hardware trojan that can carry out this attack without going through the main processor. Additionally, though we have already stated that the attack in this paper is not meant for OS-based systems, future work can be done on extending it to apply to such systems, particularly by targeting the hardware drivers that are usually present for the OS to interface with hardware. This could even apply to Real-Time Operating Systems, whose kernels are often written for bare metal. Porting this attack to other platforms as specified in this section is considered for future work.

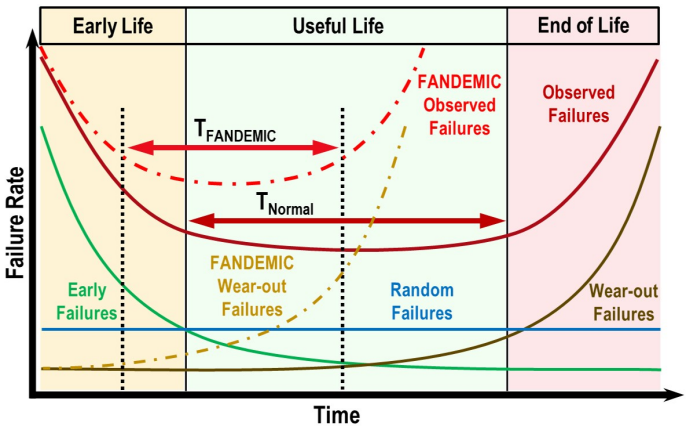


Fig. 8: Expected behavior of the life cycle of logic circuits in presence of FANDEMIC vs. Normal conditions. The red solid line shows the expected life cycle of IC components, and the red dashed line depicts the estimated behavior caused due to FANDEMIC ($T_{FANCEMIC} < T_{Normal}$). The brown solid line shows the normal wear-out failures, and the gold dashed line shows the wear-out failures caused by FANDEMIC. Additionally, the solid blue line and solid green lines represent the random failures and early failures, respectively.

C. Mitigation

Mitigation for attacks within the supply chain gets challenging. The verification stage should not only comprise of functional/logical testing, but also comprise of security aspects. Also, security standard guidelines and practices should be followed. For example, organizations should follow risk mitigation guidelines as mentioned in detail in in the document, "Defending Against Software Supply Chain Attacks" by NIST [26]. Cyber resilience techniques within the supply chain have been covered by Mitre in technical report, "Supply Chain Attacks and Resiliency Mitigations" [38].

A few of the mitigation techniques mentioned below are directed towards various forms of firmware attacks. While providing the mitigations, we generalized such that similar attacks could be tackled within other devices as well.

1) Test Points: Loopholes in hardware design can lead to significant system attacks. To make the firmware patching attack successful, attackers need a way to send back the patched firmware or malicious data to the device that could be made possible by various methods. One of them is the loophole in the hardware PCB design. The critical test points like debug ports present on the PCB pave the way for the attacker to get unauthorized access to the processor internal memory if proper debug port disabling and authorization checks are not done properly.

Apart from debug port access, the access to test points on the PCB can also lead to signal tampering and corruption. Access to hardware peripheral test points can lead to signal sniffing (data leakage if critical data is being transmitted without encryption), and signal tampering via creating hardware glitches on the respective test points that could change system behavior. Hence, to mitigate these attack possibilities it is recommended not to expose the critical test points on the PCB. Even the package of the processor used and the PCB schematic design should be complex enough to prevent PCB reversing based attacks. This mitigation approach might not be effective if the attack is targeted within the supply chain as attacks within the supply chain could insert vulnerability in the PCB design phase itself. To detect such vulnerabilities, there needs to be targeted validation and verification after the design, such that manipulation in the design should not be left unnoticed.

2) Binary Auditing: Device firmware, the main business logic of a whole system, passes through various phases from the design stage to the deployment stage. There is possibility of malicious payload insertion or patching at these stages within the supply chain that can go unnoticed if robust auditing is not done before actual deployment on the field. Even for devices that have already been deployed on the field that were not audited prior to deployment, binary auditing assists in identifying potential threats and vulnerabilities, including hard coded sensitive credentials/data, command execution, remote code execution (RCE) bugs, vulnerable applications, and other implementation based loopholes in the firmware.

Static analysis, dynamic analysis, or the combination of both are used during binary auditing depending on the firmware type, architecture, and resource availability. There are various tools useful for static analysis: Ghidra [23], Radare [31], and IDA [10] help to disassemble, decompile, and perform firmware reversing. Tools like Binwalk [2], Firmwalker [9], and Firmware Mod Kit [8] help to extract OS based binaries, traverse through the internal files, hunt for potential vulnerabilities, and statically analyze the file system. Additionally, dynamic auditing is performed on the binary to identify possible remote code execution vulnerabilities, and evaluate how the system is affected after exploitation. Tools used for dynamic auditing include GDB, Qiling, Unicorn, and a few other tools come as a savior for dynamic auditing when the device hardware is not accessible. Similarly, tools like AFL, Radamsa, boofuzz and other similar tools are used for performing fuzzing based auditing.

Unfortunately, despite the availability of different tools for binary auditing, there are still challenges that have a wide research scope. These include automation in bare metal, RTOS based firmwares, the accuracy of findings, malware detection, hardware dependency during emulation, fuzzing and many other feature inclusion to automate the whole auditing process.

3) Secure firmware updates: One of the main reasons behind the success of maliciously patched firmware is the insecure implementation of firmware update/flashing mechanisms into the system. In the IoT ecosystem, for the devices deployed on the field, there are various ways to update the firmware such as via USB, ebug interface (e.g., JTAG), over the air (OTA) or CAN bus. These updates mechanisms have different levels of security implementations. Devices could be easily patched with the malicious firmware if no security checks are implemented during firmware upload. In some devices only integrity checks are implemented, for instance using hashing or checksum, but these are also prone to attack depending on the kind of implementation. The implementation of secure boot via digital signature validation provides both authentication and integrity validation. The implementation of cryptographic operations should be compliant with the required security strength, secure key transfer and management. If the implementation weakens, it provides the opportunity for attackers to bypass the security feature and successfully flash the malicious content. Even with secure boot there have been hardware attacks and exploitation of implementation related vulnerabilities. Embedded systems being resource constrained faces trade offs between security and resource involvement. For example, in order to store the cryptographic keys, certificates, and sensitive information, hardware security modules are considered to be relatively secure but due to the involved cost, they are not incorporated in low cost IoT devices. Similarly, there are other trade offs with respect to computational and memory requirements.

The above mitigation via secure firmware update mechanisms to protect from the upload of malicious firmware might not be effective if the attack is within the supply chain itself. There are many possibilities of firmware being patched or hardware trojan insertion at the early stage in the supply chain that might go unnoticed during validation.

VI. Conclusion

In this manuscript, we expose and expand upon a novel class of firmware vulnerability targeting bare-metal IoT devices' power management IC that can be exploited for a supply chain attack. In particular, we propose a firmware attack construction and deployment on power management IC called FANDEMIC, and discuss the potential consequences of such an attack, like accelerated aging, battery failure, and sensor data corruption, demonstrating the latter in a hardware prototype. Additionally, we proposed a few different approaches to reverse engineer a system's binary for attack construction, implementing disassembly and decompilation in conjunction with static analysis, as well identification of library and API functions using simple function matching techniques based on cross-correlation. We further demonstrate it is possible to cause an incorrect reading in a pressure sensor of up to 3.75% by changing the voltage by only 0.1V, proving that this attack is feasible, and the voltage does not need to drastically change to have significant effects on the target. Lastly, we discuss several mitigation techniques against this kind of attack. Future work includes further research in the additional attack approaches presented, analyzing which other components could potentially be targeted by this firmware attack, and developing robust mitigation techniques able to secure bare-metal IoT devices.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) through Computing Research Association for CIFellows Project 2030859.

References

[1] D. Berry, D. R. Towill, and N. Wadsley, "Supply chain management in the electronics products industry," *International Journal of Physical Distribution & Logistics Management*, vol. 24, no. 10, pp. 20–32, Jan 1994. [Online]. Available: https://doi.org/10.1108/09600039410074773

[2] Binwalk, "Tool for searching a given binary image for embedded files and executable code," accessed July 16, 2021. [Online]. Available: https://tools.kali.org/forensics/binwalk

[3] S. Borg, "Securing the supply chain for electronic equipment," submitted to the NSC by The Internet Security Alliance. as part of the 60-day review of Cyberspace Policy, 2009. [Online]. Available: https://obamawhitehouse.archives.gov/files/documents/cyber/ISA%20-%20Securing%20the%20Supply%20Chain%20for%20Electronic%20Equipment.pdf

[4] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *2013 NDSS Symposium*, 02 2013.

[5] Department of Homeland Security, "Study on mobile device security - april 2017." accessed July 21, 2021. [Online]. Available: https://www.dhs.gov/sites/default/files/publications/DHS%20Study%20on%20Mobile%20Device%20Security%20-%20April%202017-FINAL.pdf

[6] C. Eagle and K. Nance, *The Ghidra Book*. No Starch Press, 2020.

[7] FireEye, "Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor," *Threat Research*, 2020. [Online]. Available: https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html

[8] Firmwalker, "Easy deconstruction and reconstruction of firmware images for various embedded devices." accessed July 16, 2021. [Online]. Available: https://github.com/rampageX/firmware-mod-kit/wiki

[9] ——, "A simple bash script for searching the extracted or mounted firmware file system," accessed July 16, 2021. [Online]. Available: https://github.com/craigz28/firmwalker

[10] IDA, "A powerful disassembler and a versatile debugger," accessed July 16, 2021. [Online]. Available: https://hex-rays.com/ida-pro/

[11] A. Inc, "General:intel hex file format," accessed July 20, 2021. [Online]. Available: https://developer.arm.com/documentation/ka003292/latest

[12] I. Jibilian and K. Canales, "The us is readying sanctions against russia over the solarwinds cyber attack. here's a simple explanation of how the massive hack happened and why it's such a big deal," *Business Insider*, 2021. [Online]. Available: https://www.businessinsider.com/solarwinds-hack-explained-government-agencies-cyber-security-2020-12

[13] JSOF, "Ripple 20: 19 zero-day vulnerabilities amplified by the supply chain," *Disclosures*, 2020. [Online]. Available: https://www.jsof-tech.com/disclosures/ripple20/

[14] C. Karamitas and A. Kehagias, "Function matching between binary executables: efficient algorithms and features," *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 4, pp. 307–323, Dec 2019. [Online]. Available: https://doi.org/10.1007/s11416-019-00339-6

[15] J. Keane and C. H. Kim, "An odomoeter for cpus," *IEEE Spectrum*, vol. 48, no. 5, pp. 28–33, 2011.

[16] T. S. X. Lab, "Safety tips for "badpower" risks in some fast charging products," *Disclosures*, 2015. [Online]. Available: https://patents.justia.com/patent/9721093

[17] Y. R. Lee, B. Kang, and E. G. Im, "Function matching-based binary-level software similarity calculation," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, ser. RACS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 322–327. [Online]. Available: https://doi.org/10.1145/2513228.2513300

[18] E. Lopatto, "Tim cook's trick for making iphones is now at risk from the pandemic: The perils of just-in-time manufacturing," *The Verge*, 2020. [Online]. Available: https://www.theverge.com/2020/3/13/21177024/apple-just-in-time-manufacturing-china-coronavirus-supply-chain

[19] A. Ltd., *DUI0553B Cortex-M4 Devices Generic User Guide*, ARM Ltd., 2011-8-3.

[20] F. E. McFadden and R. D. Arnold, "Supply chain risk mitigation for it electronics," in *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, 2010, pp. 49–55.

[21] M. McGuire, U. Ogras, and S. Ozev, "Pcb hardware trojans: Attack modes and detection strategies," in *2019 IEEE 37th VLSI Test Symposium (VTS)*, 2019, pp. 1–6.

[22] D. Mehta, H. Lu, O. P. Paradis, M. A. M. S., M. T. Rahman, Y. Iskander, P. Chawla, D. L. Woodard, M. Tehranipoor, and N. Asadizanjani, "The big hack explained: Detection and prevention of pcb supply chain implants," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 4, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3401980

[23] National Security Agency, "Ghidra," 2019, accessed June 23, 2021. [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra/

[24] B. News, "Samsung confims battery fauls as cause of note 7 fires," January 2017, accessed July 16, 2021. [Online]. Available: https://www.bbc.com/news/business-38714461

[25] NIST, "National vulnerability database," accessed July 16, 2021. [Online]. Available: https://nvd.nist.gov/

[26] ——, "Defending against software supply chain attacks," April 2021. [Online]. Available: https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf

[27] J. Obermaier, M. Schink, and K. Moczek, "One exploit to rule them all? on the security of drop-in replacement and counterfeit microcontrollers," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/obermaier

[28] J. Obermaier and S. Tatschner, "Shedding too much light on a microcontroller's firmware protection," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[29] T. Omitola and G. Wills, "Towards mapping the security challenges of the internet of things (iot) supply chain," *Procedia Computer Science*, vol. 126, pp. 441–450, 2018, knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918312547

[30] V. Prabhu, "Bad power attack: Hackers can modify the power management ic (pmic) firmware to make your smartphone explode remotely," *Disclosures*, 2020. [Online]. Available: https://androidrookies.com/fast-charging-vulnerability-could-be-used-by-hackers-to-explode-smartphones-remotely/

[31] Radare, "A free/libre toolchain for easing several low level tasks," accessed July 16, 2021. [Online]. Available: https://rada.re/n/radare2.html

[32] N. Semiconductor, *nRF52832 Product Specification*, Nordic Semiconductor, 2017.

[33] A. Semuels, "When your amazon purchase explodes," April 2019, accessed July 16, 2021. [Online]. Available: https://www.theatlantic.com/technology/archive/2019/04/lithium-ion-batteries-amazon-are-exploding/587005/

[34] S. Skorobogatov, *Physical Attacks and Tamper Resistance*. New York, NY: Springer New York, 2012, pp. 143–173. [Online]. Available: https://doi.org/10.1007/978-1-4419-8080-9_7

[35] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. USA: California Technical Publishing, 1997.

[36] SVD-Loader Contributors, "Svd-loader for ghidra: Simplifying bare-metal arm reverse engineering," accessed July 16, 2021. [Online]. Available: https://leveldown.de/blog/svd-loader/

[37] L. Tung, "Microsoft: Firmware attacks are on the rise and you aren't worrying about them enough," *ZDNet*, 2021. [Online]. Available: https://www.zdnet.com/article/microsoft-firmware-attacks-are-on-the-rise-and-you-arent-worrying-about-them-enough/

[38] G. J. S. William J. Heinbockel, Ellen R. Laderman, "Supply chain attacks and resiliency mitigations," Oct 2017. [Online]. Available: https://www.mitre.org/sites/default/files/publications/pr-18-0854-supply-chain-cyber-resiliency-mitigations.pdf

[39] K. Yang, D. Forte, and M. M. Tehranipoor, "Protecting endpoint devices in iot supply chain," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 351–356.

[40] K. Yang, D. Blaauw, and D. Sylvester, "Hardware designs for security in ultra-low-power iot systems: An overview and survey," *IEEE Micro*, vol. 37, no. 6, pp. 72–89, 2017.

[41] K. Yang, D. Forte, and M. Tehranipoor, "Resc: An rfid-enabled solution for defending iot supply chain," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 3, Feb. 2018. [Online]. Available: https://doi.org/10.1145/3174850

[42] K. Yang, D. Forte, and M. M. Tehranipoor, "Cdta: A comprehensive solution for counterfeit detection, traceability, and authentication in the iot supply chain," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 3, 4 2017. [Online]. Available: https://doi.org/10.1145/3005346

[43] S. Yang, A. Alaql, T. Hoque, and S. Bhunia, "Runtime integrity verification in cyber-physical systems using side-channel fingerprint," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, 2019, pp. 1–6.

[44] M. Yoshio, R. J. Brodd, and A. Kozawa, *Lithium-ion batteries*. Springer, 2009, vol. 1.

[45] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "Codecmr: Cross-modal retrieval for function-level binary source code matching," in *NeurIPS*, 2020.

[46] D. Zhang, Y. Han, and Q. Ren, "A novel authorization methodology to prevent counterfeit pcb/equipment through supply chain," in *2019 IEEE 4th International Conference on Integrated Circuits and Microsystems (ICICM)*, 2019, pp. 128–132.

[47] W. Zhou, Y. Jia, A. Peng, Y. Zhang, and P. Liu, "The effect of iot new features on security and privacy: New threats, existing solutions, and challenges yet to be solved," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1606–1616, 2019.

[48] H. Zhu, X. Guo, Y. Jin, and X. Zhang, "Pcbench: Benchmarking of board-level hardware attacks and trojans," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 396–401. [Online]. Available: https://doi.org/10.1145/3394885.3431596

APPENDIX

A. Full Attack Tree

The full PMIC Attack Tree is presented below. It outlines the minimum necessary assumptions for realizing this attack, and then describes it with a tree-like structure in which the root node represents the attack goal, and subtrees represent subgoals necessary to accomplish the higher ones. The | is used to denote an "or" branch, in which any of the branches that begin with | can be used to accomplish its parent goal. The & is used to denote an "and" branch, in which all branches that begin with & must have their goals completed in order to accomplish the parent goal. Additional assumptions are included as necessary. Note that some branches may be present without further elaboration. These branches may represent alternative methods to accomplish the parent goal that are not explored in this paper, or are leaf goals which are trivial or explained in the paper.

```
Assumptions:
  - PMIC is present on the target device
  - PMIC is external to device processor
  - PMIC is digitally configurable
  - device controls PMIC via chip-to-chip communication protocol
  - PMIC datasheet and reference manual available
  - processor datasheet and reference manual available

Goal: Alter behavior of external PMIC
  | - Alter configuration of PMIC via firmware modification
    & - Analyze firmware for PMIC configuration information
      | - Reverse engineer original firmware source code
        & - Gain access to original source code
        & - Locate/determine PMIC configuration section of code

      | - Reverse engineer original firmware binary
        & - Gain access to original firmware
        & - Disassemble firmware binary
        & - Locate PMIC configuration section of assembly
          & - manual analysis (necessary to some extent in most cases)
          + - address cross referencing
            Assumptions:
              - access to static analysis tool capable of generating memory cross
                 references (like Ghidra)
              - peripherals are memory-mapped with known addresses

            | - locate instructions that make reference to peripheral addresses

          + - function matching
            Assumptions:
              - firmware relies on manufacturer-provided libraries
              - libraries are open source
              - libraries can be publicly compiled
              - attacker can compile libraries on any publicly available OS
              - attacker can compile libraries with any possible optimization

            | - function matching via binary cross correlation
            | - function matching via instruction cross correlation
            | - other function matching techniques

          + - symbolic execution
            Assumptions:
              - symbolic execution can be performed on target architecture
              - firmware is sufficiently small to limit state space explosion
              - peripherals (like comm. protocols) are memory-mapped
              - sufficient time and processing power for execution
```

```
          & - explore execution for accesses to comm. peripherals
          & - backtrace execution for configuration section

  & - Construct modified firmware
  & - Load modified firmware to device

| - Alter configuration of PMIC by hijacking communication via hardware trojan
| - Alter configuration of PMIC by external processor via probing
```