POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

In collaboration with University of Calgary

An In-Depth Study of Smart Building Systems: Firmware Analysis and Device Emulation

Supervisors

Prof. Riccardo SISTO

Prof. Lorenzo DE CARLI

Candidate

Gabriele GARDOIS

July 2025

Summary

The Internet of Things (IoT) devices have become key components in modern systems, serving a diverse range of functions, including environmental monitoring, management of critical appliances and automation of processes. This study presents a comprehensive security assessment of IoT devices deployed in Building Automation Systems (BAS), with a focus on firmware analysis and dynamic emulation techniques.

As Smart Buildings evolve and increasingly connect to the internet, security practices often do not keep pace, resulting in the deployment of vulnerable systems with insufficient protection. This research introduces a novel dataset, currently missing in the field, comprising 56 firmware images exclusively associated with BAS-related IoT devices. A tailored methodology was developed to emulate these firmware images, achieving basic emulation for 60% of them. Full network environments were established for 15 images, enabling testing of network services.

Additionally, static analysis revealed an average of 1,570 Common Vulnerabilities and Exposures (CVEs) per firmware image. Further examination of kernel and GCC toolchains versions shed light on vendors' practices in selecting and maintaining core software components.

The goal of this research is to collect firmware, develop an automated analysis methodology, identify software vulnerabilities, investigate firmware components, and conduct case studies to evaluate the emulation process. These findings contribute new empirical data to the domain of Smart Building security, providing a foundation for future research and highlighting the urgent need for improved security standards in the development and deployment of IoT devices.

Ringraziamenti

A mia nonna Margherita

Dedico questo spazio al ringraziamento delle persone che in vari modi hanno contribuito alla creazione di questo elaborato o che mi hanno aiutato lungo il mio percorso universitario e di vita.

Ringrazio i professori Riccardo Sisto e Lorenzo De Carli per avermi permesso di effettuare una bellissima esperienza presso l'università di Calgary e di aver contribuito alla realizzazione di questo lavoro, garantendomi sostegno e guidandomi nella ricerca.

Ringrazio di cuore le due persone che sono il mio punto di riferimento nella vita, mia mamma e mio papà. Mi avete sostenuto lungo questo impegnativo percorso universitario, permettendomi di raggiungere l'ambizioso traguardo della laurea. Tuttavia, vi voglio ringraziare soprattutto perché mi avete cresciuto e aiutato in tutte le fasi della vita a diventare la persona che sono oggi.

A Giulia, la mia sorella preferita, grazie per essermi stata da guida fin dai miei primi passi, quando io spingevo quella coccinella e tu, da vero capitano, ti limitavi a guidarla. Sei fonte d'ispirazione e di tanti miei sorrisi.

Ringrazio tutti i miei amici, parte integrante della mia vita. Vi ringrazio non perché mi abbiate spronato a studiare o consigliato con particolare saggezza, ma perché, togliendo tempo allo studio avete dato vita a quella parte di me che altrimenti non saprei esprimere. Ringrazio Pol, con cui sono cresciuto, per essere sempre stato al mio fianco e aver condiviso tantissime avventure. So di poter sempre contare sul tuo appoggio, come tu sul mio, e ti sono grato per avermi spinto ad andare oltre ai miei limiti. Ringrazio Ricuz per essere il mio consigliere di fiducia e per saper sempre trovare il lato positivo delle cose. Ringrazio Dado, che non si tira mai indietro quando c'è bisogno di aiuto, consigli o di uscire. Ringrazio Albi, perché sa sempre come tirarmi su il morale e rendere ogni esperienza un'avventura. Ringrazio Andre per le lunghe chiacchierate, i consigli e le riflessioni sulla vita. Ringrazio Ferle per avermi insegnato la pazienza; non ottenere mai una tua risposta a messaggi e telefonate mi ha permesso di padroneggiare la frustrazione e affrontare con più serenità esami e avversità.

Ringrazio il mio compagno di avventure canadesi, Francesco, per tutto il suo aiuto e la sua amicizia.

Infine, un ringraziamento speciale va alle mie nonne, per l'amore che mi hanno sempre dimostrato.

Table of Contents

List of Tables

List of Figures

Listings

Acronyms

AI
Artificial Intelligence

BAS
Building Automation System

BMS
Building Management System

CVE
Common Vulnerabilities and Exposures

CWE
Common Weakness Enumeration

DNS
Domain Name System

FTP
File Transfer Protocol

HTTPS
Hypertext Transfer Protocol Secure

HTTP
Hypertext Transfer Protocol

IB
Intelligent Building

IoT

Internet of Things

IP

Internet Protocol

LAN

Local Area Network

MAC

Medium Access Control

MQTT

MQ Telemetry Transport

NAT

Network Address Translation

RAM

Random Access Memory

SB

Smart Building

SoC

System on a Chip

SSH

Secure Shell Protocol

TCP

Transmission Control Protocol

UDP

User Datagram Protocol

VLAN

Virtual Local Area Network

VM

Virtual Machine

WAN

Wide Area Network

Wi-Fi

Wireless Fidelity

WSL

Windows Subsystem for Linux

Chapter 1

Introduction

Smart Buildings (SBs) are rapidly increasing in popularity all over the world, driven by the increasing integration of Internet of Things (IoT) technologies into building operations. By embedding IoT devices into their infrastructure, modern buildings can achieve a high level of automation and responsiveness. These systems are being deployed in a wide variety of contexts: in offices and residential spaces, they manage heating, ventilation, and access control; in airports, they oversee window shutters, smoke detectors, surveillance systems, and fire alarms; in hospitals, they control elevators, lighting, and air conditioning; and in factories and greenhouses, they monitor and adjust parameters such as temperature, humidity, and other environmental variables.

The ability to collect real-time environmental metrics, combined with data analysis, enables significant optimization of resource consumption, reducing energy and water usage while providing user comfort. The seamless integration of smart technologies promotes more adaptive and efficient environments, allowing occupants to flexibly configure their spaces.

At the heart of these buildings lies a complex network of smart devices consisting of sensors, actuators and controllers. This infrastructure is managed by a Building Management System (BMS), which is responsible for orchestrating, monitoring, coordinating, and responding to environmental conditions. These ecosystems are often highly heterogeneous, comprising devices from multiple vendors and incorporating both modern and legacy technologies. In many cases, these systems evolve over time through incremental upgrades, leading to fragmented architectures.

With the advent of internet connectivity for Building Automation Systems (BAS), new capabilities have emerged: remote monitoring, predictive maintenance, and user-driven control. Yet this increased connectivity also opens the door to cyber threats. Many BAS implementations expose critical infrastructure to the internet without adequate security measures, making them vulnerable to remote attacks [1, 2, 3, 4].

In this context, it is essential to conduct a security-focused assessment of commercially available BMS devices. Analysing firmware components in depth can uncover critical vulnerabilities and poor design choices. To facilitate such research, this study presents a novel dataset and a structured methodology to support automated security assessments of smart building firmware. The goal is to contribute to the research in the realm of SB security by offering insightful evaluations, findings, and resources that can aid future investigations.

1.1 Contributions

This research aims to investigate the security landscape of Smart Buildings by reviewing relevant scientific literature and conducting an independent analysis of IoT firmware. To this end, detailed information about BMS was gathered, offering a comprehensive overview of key technologies and innovations in the field. The study places particular emphasis on security issues that hinder the safe development of smart infrastructures, and highlights the widespread vulnerabilities found in IoT devices deployed in such environments.

A central contribution of this work is the creation of a firmware dataset specifically tailored to the domain of BAS resource notably lacking in existing literature. The dataset comprises 56 firmware images from four different vendors, weighting approximately 14 GB. It is intended as a starting point for future research, providing ready-to-use material for analysis and experimentation.

Another core aspect of the research is the development of a methodology for assessing firmware security. This process encompasses multiple phases: acquisition, unpacking, reverse engineering, and ultimately, firmware emulation to enable dynamic analysis. Furthermore, the process was scaled to process the whole dataset.

A security evaluation was performed on the firmware samples collected. By identifying Common Vulnerabilities and Exposures (CVEs), the study outlines the current state of security within the smart building ecosystem. This evaluation also includes an assessment of the overall cybersecurity posture, supported by observations of security trends over time.

Additionally, two case studies were conducted using penetration testing techniques to evaluate the security perimeter of selected firmware. These case studies are presented in a tutorial-like format, providing an in-depth, step-by-step walkthrough of the analysis process and demonstrating how to replicate the methodology.

Finally, the study concludes with a discussion of key challenges and urgent actions required to strengthen security in the BAS domain. It highlights areas for improvement and offers guidance for advancing research in the field of smart building cybersecurity.

1.2 Thesis Outline

This section provides a brief overview of the discussed topics for each chapter of this thesis.

Chapter 2:

Chapter Two offers an introduction to Smart Buildings by examining their key functionalities. It analyses the systems that constitute this infrastructure and discuss how they are structured. The discussion then focuses on the devices employed, the actions they perform, and the communication protocols commonly used in BAS. The background section concludes with an overview of the main security threats that can compromise the reliability and integrity of these systems.

Following this, the thesis presents the related work section, which aims to provide a comprehensive overview of the main challenges in this field through a review of the most relevant academic literature. This analysis covers various aspects of BMS, with particular attention to the efforts required to conduct dynamic firmware testing.

Chapter 3:

Chapter Three focuses on the creation of the dataset. It begins by analysing the current state of vendors operating in the BAS domain, with particular attention to emerging trends in firmware distribution. The chapter then details the process of acquiring firmware from various vendor websites and the methods employed to carry out this task.

The discussion continues with an explanation of the filtering process used to distinguish between encrypted, compressed, and unprocessed firmware samples. This is followed by a description of the organizational structure developed to store the firmware and how it supports subsequent operations. Finally, the chapter presents statistics on the types of devices and communication protocols included in the dataset.

Chapter 4:

Chapter Four outlines the methodology used to process the firmware collected in the dataset. The chapter details the approach, which consists of four main steps: firmware fetching, firmware extraction, firmware analysis, and firmware emulation. Each of these steps is described in terms of the actions performed, the tools utilized, and the challenges encountered during the analysis process.

Finally, to process all the firmware samples in the dataset, this approach is automated using the EMBA tool [5], a framework designed to streamline firmware

analysis and facilitate the creation of emulation environments. The chapter provides an explanation of the tool's functionalities, as well as some of its limitations.

Chapter 5:

Chapter Five presents the results obtained from the conducted research. It is divided into two main sections. The first section provides aggregated results in the form of statistics and analyses of components such as the kernel. A detailed explanation of the collected CVEs and their associated CWEs is also included.

The second section presents two case studies, each corresponding to a specific firmware sample. These analyses explore the status of the emulations generated by processing the firmware, highlighting some troubleshooting interventions required to fully access the system's functionality.

Chapter 6:

Chapter Six further elaborates on the implications of our results. It discusses the limitations of the work, identifying areas for potential improvement. Finally, the chapter offers recommendations for future research in this field, suggesting ways to enhance the depth and scope of the study.

Chapter 7:

The final chapter provides a summary of the work conducted throughout this thesis, with a focus on the contributions made by the study. It summarizes the key findings, highlights the innovations introduced, and presents some noteworthy reflections on the research process.

Chapter 2

Background and Related Work

This chapter provides a brief introduction about smart buildings and a literature review about how these systems work and how to assess their security.

2.1 Background

2.1.1 Smart Buildings

Smart buildings are structures that automate part of their functions through electronic devices, making them cyber-physical systems. This need arises from the necessity of managing complex buildings such as hospitals, airports, residences, and offices, which must efficiently handle resources like ventilation, lighting, and heating. In this context, computing devices such as sensors, actuators, and controllers play a crucial role by collecting data, automating actions, and continuously monitoring the environmental conditions of the building. Among these devices there are heat sensors, photocells, pressure sensors, cameras, smoke detectors, card readers and many others. These devices encompass a wide range of components, such as heat sensors, photocells, pressure sensors, cameras, smoke detectors, card readers, fire alarms, door locks and others.

Within the building, these devices are distributed across a network and they are capable of sensing the environment and responding to the users needs by undertaking actions. The core of this network is the BMS, consisting of both software and hardware components that function as the "brain" of the entire infrastructure. The BMS can be programmed with specific parameters and actions and is then able to enforce these through the network of devices under its control. To facilitate management, this architecture is divided into subsystems, which

include: Heating, Ventilation, and Air Conditioning (HVAC), lighting controls, hydraulic systems, fire safety subsystems, electrical systems, security and access control, and CCTV.

The primary goal of these buildings is to optimize energy consumption, ensure the enforcement of safety measures, improve comfort within the spaces, and provide immediate responses in case of emergencies. Additionally, the connection of these systems to the internet enable remote maintenance, security image monitoring, and diagnostic data access outside the building. This integration allows buildings to be more efficient, environmentally friendly, safe, and effective.

Devices

For an IB to operate effectively, a variety of devices are required. These include sensors, actuators, and controllers, each functioning at distinct levels within the system. The scientific literature typically identifies three layers, each responsible for specific functionalities [3, 6, 7]. These levels are:

- ***Field Level***: This is the lowest level of the system, where sensors and actuators are located. These devices are responsible to collect metrics about the environment, tracking the location of people within the building, and providing responses. In essence, they serve as the "eyes" and "arm" of the system.
- ***Automation Level***: This intermediate layer predominantly consists of controllers. Their primary role is to collect and aggregate data from field-level devices. They are often responsible for managing specific subsystems, executing predefined routines, and monitoring the status of the equipment under their control.
- ***Management Level***: This is the topmost layer, where the entire system can be accessed and managed. It consolidates data from all subsystems, performing analysis and generating reports. This level is also responsible for logging activities, archiving data, and forecasting future actions. Administrators interact with the system at this level to review statistics, modify system parameters, and set rules.

The devices within SBs serve a variety of purposes and can range from dedicated computers designed for specific tasks to very small sensors, often measuring just a few centimeters. These devices are commonly referred to as IoT devices, and they are vital to the operations of a BAS. Although their functions are mission-critical, they are subject to several limitations.

Many of these devices operate in environments where reliable power supply is not always available, which makes them dependent on batteries. This reliance imposes

constraints on their operational lifespan and computational power. Additionally, their compact form factor limits processing capabilities and memory capacity, as they are typically equipped with low-power processors and minimal storage. These technical limitations, when combined, render IoT devices particularly attractive targets for cyber-attacks. Moreover, from a business perspective, security is often perceived as a cost weighed against its potential impact and likelihood of exploitation. As a result, companies frequently make trade off during development, opting to reduce costs and accelerate time-to-market by skipping the implementation of comprehensive security best practices. This approach worsen the inherent constraints of IoT devices, increasing their vulnerability when manufacturers choose not to invest adequate resources in security during the design and development phases.

BAS Protocols and Network Topology

In smart buildings, IoT devices are interconnected through a network, and communication between them is crucial to carry out operations effectively. Within the context of BMS, both open and proprietary protocols are widely used to enable these communications [8, 9, 10, 4]. Below is a list of some of the most commonly used protocols:

- Wired (with potential wireless support): BACnet, KNX, LonWorks, DALI, Mbus, Modbus
- Wireless (which may support wired protocols or manage devices directly): ZigBee, EnOcean, Z-Wave

BACnet, KNX, and LonWorks are the most widely used for managing communications across all levels, from management level to field level operations. The other protocols are typically designed for communication at the field level or automation level and generally lack the structure required to handle a comprehensive Building Automation and Control System.

In most BMS implementations, a single communication protocol is rarely used across the entire infrastructure. This is primarily because, as previously noted, individual protocols are typically designed to serve specific layers, such as management, automation, or field levels, rather than all levels simultaneously. Furthermore, automated systems are often expanded or upgraded incrementally over time, which results in the integration of components that rely on different communication protocols. To illustrate this multi-protocol approach, an example network topology has been developed and is presented in Figure 2.1, based on a review of the scientific literature [4, 11, 7, 3].

The network topology consists of two networks interconnected via an intranetwork, where the central management system resides. This configuration aligns with

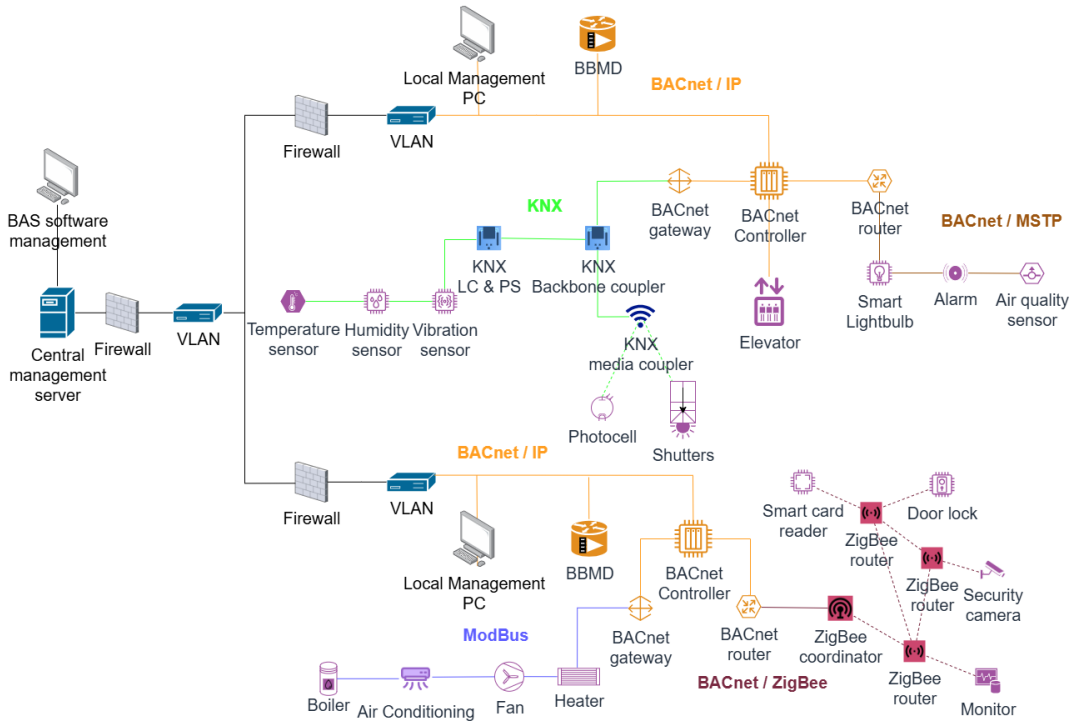


Figure 2.1: Smart building network topology

realistic scenarios, as BAS often need to manage multiple buildings. Additionally, the intranetwork introduces networking challenges that can be leveraged to highlight important aspects, such as the use of firewalls and VLANs, which should be implemented as security layers.

For the management and automation layers, BACnet has been selected as the primary protocol. BACnet facilitates communication between the central server and the main controllers across subnetworks. To manage broadcast traffic between these subnetworks, BACnet Broadcast Management Devices (BBMDs) are deployed.

The field layer in this topology utilizes a variety of protocols to emphasize the heterogeneous nature of a SB. To ensure interoperability between these diverse protocols, the use of gateways and routers is critical. In the context of BAS, gateways are devices that connect networks using different protocols, effectively mapping information between two domains. Routers, on the other hand, enable communication between networks that employ the same high-level protocol but differ in the underlying medium (e.g., BACnet/IP to BACnet/MSTP).

Sensors and actuators are deployed within subsystems to handle specific tasks. These devices communicate with controllers that are compatible with the corresponding protocols in the hierarchy. The controllers are responsible for collecting

data and transmitting it to higher-level controllers through gateways and routers, facilitating data exchange across the network.

2.1.2 Security threats

The ability of SBs to collect information about the building, its environments, and its users, as well as to take actions such as regulating heating, ventilation, and controlling access to areas, is both their greatest strength and their greatest vulnerability. This is because the information they manage is highly sensitive, and the same applies to the actions they can perform. Cyber-physical systems, such as those in SBs, are inherently risky, as demonstrated by the Google Australia Office Attack [1] and Finland Heating System Attack [2] cases. By compromising the IT side of these systems, physical damage can happen.

A list of potential attacks on an IB may involve the following data and actions: tracking users movements and habits, manipulating video streams or alarms, stealing personal and environmental data, locking and unlocking doors, and controlling elevators, ventilation, and heating systems. This makes SBs a perfect target for being tampered, in order to steal their high valuable information or to hijack their mission-critical actions in order to take control over the building, causing significant damages.

Thus, creating secure IT infrastructures to manage operations within BAS should be a primary goal for those developing software and hardware for such systems. However, looking on the internet for connected devices, using search engines like BinaryEdge, Shodan, ZoomEye, and Censys reveals many systems connected without adequate protection.

The root cause of this issue can be attributed to three main factors: buildings using outdated devices that are not easily replaceable, protocols lacking proper security measures, and IoT devices that do not adhere to necessary security standards.

2.2 Related Work

Smart Building Technologies and their Security

Verma et al. offers a comprehensive overview [12] of the state-of-the-art technologies in smart buildings. Smart devices represent a new frontier in environmental monitoring, enabling optimized control of heating, ventilation, and air conditioning (HVAC), artificial lighting, daylighting systems. BMS can optimize energy consumption, support remote monitoring, and assess the structural health of buildings. However, as cyber-physical systems, these infrastructures can potentially cause harm if they fail or are unable to deliver critical services.

Wendzel [13] spotlights risks and vulnerabilities affecting IoT devices in SBs. The communication standards currently employed in SB environments were not originally designed to support exposure to Internet-based communications. Yet, growing automation trends are increasingly driving these systems toward remote connectivity. This issue is exacerbated by the presence of legacy devices in these systems, lacking the computing power necessary to implement modern security measures. Wendzel emphasize the need for collaboration between academia and industry to develop comprehensive solutions for both new and legacy smart buildings.

Similarly, Ciholas et al. [6] underscore that the earliest automated buildings were never intended to be Internet connected. While remote connectivity enables centralized data management, inter-building communication, and remote maintenance introduces significant cyber-attack vectors. Smart buildings, as complex networks of interconnected nodes, must accommodate a wide variety of devices and protocols. This heterogeneity poses substantial security challenges, especially given that the main protocols used in Intelligent Buildings (IB) were not originally designed with security in mind. The authors argue that collaboration between academia and industry, as well as among companies themselves, is critical to audit existing protocols and devices and to develop better standards.

To improve the security level of BAS Younus et al. [14] propose to leverage Software Defined Networking (SDN) technologies. SDN enables centralized network control and offers several advantages, including flexible network configuration, centralized enforcement of security policies, and global visibility across the system. These features can simplify operations and reduce overall system complexity. Despite its numerous advantages, centralized approach could also lead to a single point of failure taking away resilience and fault recoverability to the system.

BAS protocols and security issues

Lohia et al. [8] present an overview of several communication protocols commonly used in IB, including LonWorks, KNX, BACnet, DALI, EnOcean, and Zigbee. Their comparative analysis focuses on key aspects such as network topology, transmission medium, communication modes, and security features. The authors also emphasize the role of gateways, devices that enable integration between different protocol, which can introduce security vulnerabilities into the system if not properly managed.

Ferreira et al. [9] investigate the issue of interoperability among protocols within SBs. Thet split the devices within IB into three hierarchical layers: management level, automation level, and field level. Their analysis focuses on identifying which protocols, particularly BACnet, KNX, and LonWorks, is most suitable for each layer. The authors conclude that interoperability is essential in BAS, as a single protocol is often insufficient to effectively span all layers. A combination of protocols tends to yield better performance and flexibility.

In a complementary study, Domingues et al. [10] provide a systematic review of fundamental BAS characteristics using established standards as reference points. They offer a functional overview of widely used protocols such as BACnet, KNX, LonWorks, Modbus, ZigBee, EnOcean, Insteon, and Z-Wave, outlining their capabilities, use cases, advantages, and limitations. A key insight from their work is that interoperability among these protocols often relies on proprietary, custom-made extensions. This reliance hinders standardization and poses significant challenges in terms of security and long-term maintainability.

Morales-Gonzalez et al. [4] conduct a thorough analysis about BAS protocols and their associated security extensions. Since many of these protocols, such as BACnet, EnOcean, KNX, LonWorks, Modbus, ZigBee, and Z-Wave, were originally designed in the 1990s or earlier, they often lack even the most fundamental security mechanisms. The study highlights a broad range of vulnerabilities affecting these protocols, including brute-force attacks, covert channel attacks, cryptographic attacks, denial-of-service attacks, eavesdropping, false data injection, fuzzing, man-in-the-middle attacks, physical attacks, reconnaissance, replay attacks, spoofing, and side-channel attacks. Furthermore, the authors examine recent security extensions introduced to address these vulnerabilities. They observe that, in recent years, most protocols, except LonWorks, have seen the development of such extensions. However, these enhancements are generally limited to IP-based implementations, leaving non-IP variants exposed to the same threats. Notably, when testing the extensions some security issues persisted, indicating that the current countermeasures remain insufficient.

Reverse engineering IoT firmware

Kaushik et al., in their work [15], highlight significant security vulnerabilities in IoT firmware for smart home devices and propose a structured roadmap for conducting a security analysis. From the initial extraction of firmware via hardware dumping to subsequent analytical stages, the authors delineate the core components of firmware and recommend the most suitable tools for performing comprehensive security assessments.

Similarly, Shwartz et al. [16] investigate the domain of reverse engineering in the context of IoT devices. They place particular emphasis on the methodology for analysing file systems to retrieve critical information, which is subsequently utilized to identify potential security vulnerabilities.

In another contribution, Chen et al. [17] describe an approach for detecting insecure, reused libraries within IoT device firmware. To this end, they develop a web crawler capable of collecting thousands of firmware images from online sources. Their analysis then focuses on techniques for examining the file systems to extract library components, assess their usage, and evaluate their security implications.

Emulation, Hardware Abstraction and Rehosting

Setting up an environment capable of interacting with software, which communicates directly with hardware, at a low level presents a significant challenge. This difficulty arises from the tight integration between firmware and the board on which it is intended to run.

A conventional approach to address this challenge involves hardware emulation, as demonstrated in works [18, 19, 20]. Emulation involves replacing physical hardware with software that can interact transparently with the firmware, imitating the original hardware environment without letting the program to notice it. To achieve this, all relevant hardware components of the target board must be accurately emulated to ensure reliable and consistent interaction during execution.

Clements et al. introduced HALucinator [21], a framework designed to enhance firmware emulation by leveraging Hardware Abstraction Layers (HALs). HALs are software libraries that abstract hardware functionalities, allowing developers to interact with hardware through high-level interfaces. The use of HALs results in firmware that is less tightly coupled to specific hardware platforms, a property that presents promising opportunities for emulation. By identifying the functions provided by these libraries, it becomes possible to construct generic hardware emulation models (i.e., peripheral models) capable of producing realistic responses. HALucinator provides an emulation environment on top of QEMU, providing peripherals interactions.

To address various limitations in firmware emulation, such as difficulties in emulating System-on-Chip (SoC) components, peripheral modelling, and CPU emulation, Fasano et al. proposed an alternative strategy for whole-system dynamic analysis [22]. Their idea consists in modelling just the hardware features necessary to supply firmware's dependencies, giving as a result a reliable representation of firmware running on its original hardware. This happens through rehosting which is based on decoupling the system software stack from its physical hardware. A rehosted environment is tailored to a given firmware image by modelling expected behaviour based on: data collected from physical devices, available documentation, and iterative refinement of responses. Through fidelity analysis, it becomes possible to calibrate this model to accurately reproduce the hardware dependencies of the firmware.

Framework for Emulation and Firmware Analysis

Testing software dynamically presents a significant challenge, as it necessitates the emulation of the underlying structures that support the software. This process can be approached through three distinct levels of analysis: the application level, the process level, and the system level. When the goal is to emulate an entire firmware, rather than just a single application, the system level is the only viable

and effective approach.

The authors Chen et al. developed FIRMADYNE [19], a framework designed to extract firmware, create emulation images, and execute dynamic analysis. The primary objective of this tool is to leverage software-based full system emulation to scale the analysis process, rather than relying on hardware. A key contribution of this framework lies in its method for creating an appropriate emulation environment, which is built upon four fundamental components: NVRAM, the Kernel, system configuration, and QEMU [18].

During an initial emulation learning phase, the NVRAM module is responsible for intercepting calls to non-volatile memory and subsequently providing the appropriate values during emulation. When combined with an orchestrated Linux Kernel capable of hooking system calls, this module ensures a reliable software-based substructure to replace hardware. Furthermore, once the emulated environment is correctly configured, the network interfaces are tested. This phase is critical for learning and inferring the correct configurations needed to enable network communication, thus facilitating subsequent analysis.

On the trail of FIRMADYNE, Kim et al. developed FirmAE [20], a framework designed to address some of the shortcomings of the aforementioned framework. Their research focused on resolving issues encountered during firmware emulation with FIRMADYNE, analysing the causes of these failures, and implementing effective solutions. The technique proposed, called "Arbitrated Emulation" ensures that the high-level behaviour of hardware components is replicated through software, though it sacrifices some fidelity to the physical device's behaviour.

The arbitrations were introduced in four main stages: boot, network, NVRAM, and Kernel. By identifying and addressing critical errors commonly observed during failed emulations, the authors made significant improvements about: the boot sequence, handling of IP aliases, VLAN setup, and support for additional kernel modules and NVRAM configurations. These enhancements resulted in improved performance, particularly in terms of the number of firmware images that could be emulated.

Building upon the work of these two frameworks, the Embedded Analysis Toolkit (EMBA) was developed. This toolkit has become an essential resource, extensively utilized in the analysis conducted throughout the research presented in this thesis.

Fuzzing in emulated environments for IoT and BAS protocols

The ability to emulate device's hardware enables the application of fuzzing techniques to firmware. Fuzzing is a software testing method that involves injecting randomized or unexpected inputs into a system to identify security vulnerabilities by monitoring the system's behaviour and responses.

Zhou et al., in their survey [23], examine several challenges associated with

creating emulation environments suitable for fuzzing firmware on microcontroller units (MCUs), as well as the broader obstacles in applying fuzzing to embedded systems. Among these challenges are the scalability of automated hardware emulation with high fidelity, minimizing false positives and false negatives, and the difficulty of detecting bugs via operating system mechanisms in systems using Real-Time Operating Systems (RTOS). Despite these difficulties, focusing research on this area holds considerable promise. Emulated environments can potentially support fuzzing at speeds unattainable on physical hardware, thereby significantly accelerating vulnerability discovery.

Kim et al. [24] identify several key challenges that hinder accurate fuzzing of IoT devices. These include the difficulty of handling structured inputs, limitations of emulation-based approaches that fail to fully account for all hardware components (e.g., reads and writes to NVRAM that may cause crashes), and memory corruptions classified as bugs when they are emulation errors. To address these issues, the authors developed FIRM-COV, a framework designed to enhance the reliability of fuzzing in such constrained environments. Their method employs a dual-layer emulation strategy that executes firmware in user-mode by default, switching to full-system emulation only when exceptions are raised. This is combined with specialized handling for panic states and hardware-dependent functions, resulting in improved fuzzing performance and reduced false positives.

In the context of Industrial Control Systems (ICS) and BAS, Programmable Logic Controllers (PLCs) are widely used as dedicated computing platforms for executing specific control operations. Recognizing the importance of securing these systems, Tychalas et al. introduced ICSFuzz [25], a fuzzing framework tailored to ICS environments. Their approach includes modelling General-Purpose Input/Output (GPIO) ports to emulate interactions with custom devices that manage sensor input. This enabled reliable system responses during fuzzing analysis, making it possible to assess the security of PLCs more effectively.

Zhang et al. focus their work specifically on BAS with the development of the Building Automation System Evaluator (BASE) [26], a fuzzer designed to assess the security of BAS networks by targeting physical devices. They identify three main challenges in fuzzing BAS environments: the complex structure of BAS messages encapsulated within IP packets, the prevalence of closed-source BAS clients which complicates code coverage analysis, and the limited throughput capacity of BAS devices.

To address these issues, BASE integrates four core components: Protocol Analyzer, Core Fuzzer, Client Inspector, and Server Examiner; which together facilitate effective fuzzing of BAS protocols. By extracting context-sensitive information from network packets, the fuzzer is able to generate meaningful inputs for both clients and servers. Furthermore, through code instrumentation, response analysis, system functions tracking, and the application of AFL-style code coverage metrics,

BASE enhances visibility into the execution paths of proprietary BAS software. To further improve performance, the framework observes normal traffic patterns, records timestamps, and adapts traffic generation according to active session states, thereby optimizing fuzzing efficiency.

Chapter 3

Dataset

In this chapter, the creation of the dataset will be discussed. The discussion begins by detailing the selection process for the firmware and the analysis conducted to identify the various products currently available on the market. Finally, it describes the actual process of assembling the collection. Moreover a description of the dataset is given with aggregated statistics of the collected software.

3.1 Motivation and Overview

The goal of creating a dataset arises from its current absence in scientific literature. As highlighted by Domingues et al. [10], there is a particular scarcity of documentation related to smart buildings. This lack is compounded by the overall underdevelopment of academic research in this domain. This observation was confirmed by Xinwen Fu, a professor at University of Massachusetts Lowell and an active researcher in the field of IoT security and privacy, who has authored studies focusing on BAS, such as [4, 26]

The creation of this firmware collection aims to serve as a foundational resource, enriching and supporting further research efforts. Collecting software associated with automation devices enables and streamlines additional studies by providing:

- A comprehensive survey of companies actively involved in the sector.
- Immediate access to material required for testing, enabling different kind of analyses with various tools and objectives.
- Opportunities for comparative analyses:
  - With similar devices if other datasets are created.
  - With IoT devices across different application domains.

The work by Ciholas et al. [6] supports the points outlined above and offers some critical insights on this topic. Their research team initially collected a total of 1,697 papers, subsequently narrowing these down to 90 through a screening process. This systematic review highlighted how the smart building field is rapidly growing, in parallel with increasing technological complexity. Nonetheless, it also emphasized a notable lack of empirical evaluations. According to the study, nearly 40% of the reviewed papers did not conduct actual tests on devices or protocols, instead focusing on guidelines or general discussions.

3.1.1 BAS Vendors

The field of smart buildings encompasses various product categories, ranging from IoT devices capable of data collection (e.g. sensors) to dedicated computers whose primary function is aggregating data, issuing commands, and managing the traffic of field devices. This extensive diversity of device types has led the industry to develop comprehensive ecosystems within the realm of BAS. To better understand the devices employed within IB, an analysis was conducted to identify active companies in the sector. Starting with references from selected papers, such as [3, 27], an initial list of suppliers was compiled. This list was subsequently expanded through targeted web searches using keywords like BMS, BAS, SB, and IB. The resulting list includes the following vendors:

- ABB
- Automated Logic
- Bosch
- Carrier
- Cisco
- Contermporary Controls
- Computrols
- Delta Controls
- Distech Controls
- Honeywell
- Johnson Controls
- Kentix

- LG
- MIDITEC
- Optergy
- Paragon Controls Inc.
- Priva
- Reliable Controls
- Schneider Electric
- Semtech (formerly Sierra Wireless)
- Siemens
- Trane
- Tridium
- WAGO

Examining the list of vendors, a clear pattern was noticed in the offerings provided by the majority of these companies. Specifically, vendors tend to create comprehensive product packages capable of satisfying most consumer requirements, but the systems remain essentially closed. Among the various ecosystem offerings identified, were found: Automated Logic with WEBCTRL®, Honeywell with Alerton and BMS, Johnson controls with EasyIO and OpenBlue, Kentix with KentixONE, LG with LG MultiSITE™ VM3, Schneider Electric with EcoStruxure™, Siemens with Desigo and Trane with Tracer® SC+.

All these product bundles include more or less the same recurrent kind of devices, software and services. Each company aims to provide an interoperable architecture and platform, which enables the communications between the devices and their management. This interoperability is achieved through a set of open standard protocols (e.g. BACnet, KNX, LonTalk, and Modbus) and with frameworks (e.g. Sedona), but it also often involves proprietary solutions (e.g Niagara framework).

The set of hardware and software components they typically offer includes:

- **Edge devices**: also called field devices, these include sensors and actuators deployed to monitor environmental conditions and execute specific actions.
- **Controllers**: devices designed to manage and aggregate data from edge devices, providing the capability to engineer and control the building's systems.

- **Displays**: portable and not portable devices used to monitor real-time operations and provide building statistics and performance data.
- **Software and Interfaces**: a broad range of software solutions specifically developed to manage controllers and sensors, monitor the building's environment, and function as command centers.

Moreover companies include in their offerings a whole range of services such as consulting, deployment, integration and maintenance. These activities are typically carried out by engineers and technicians employed directly by the company selling the product bundle. This approach underscores their intent to provide complete packages, minimizing the integration or mixing with products from other vendors.

3.1.2 Firmware acquisition

The acquisition of the firmware began with the vendors listed in 3.1.1. By consulting the manufacturers' websites, it is possible to find pages related to the commercial solutions they offer, which include products such as controllers, sensors, and various types of software. However, these descriptions are often very brief, and in only a few cases is it possible to find more detailed information about the products, such as reports and technical specifications. The preferred strategy of nearly all BAS technology suppliers is to connect the buyer with a sales representative, who can be contacted via email and give all kind of information about the products.

Given the initially discouraging landscape, a change in approach was deemed necessary. Therefore, searches were conducted using search bar queries with key phrases such as *<vendor's name> + "firmware" + "device update" + "software update"*. This type of search yielded more promising results, uncovering vendor-related websites, distinct from the main ones. These sites are dedicated to technical specifications related to devices and, in some cases, containing firmware or device updates. However, access to these sites was often restricted unless one was registered. In most cases, registration required the submission of a specific code issued by the manufacturer, likely provided after contacting their sales department and purchasing the product. In a few cases, registration could be completed with just an email address, without additional identifying information from the supplier.

Through our research work, we were able to download software from four vendors: Contemporary Controls, Johnson Controls, Siemens, and WAGO.

The vendors Johnson Controls and Siemens provide two portals with open access to their respective commercial solutions, Desigo [28] and EasyIO [29], while WAGO offers its product software on the portal [30], which requires registration. Finally, Contemporary Controls provides only a limited selection of firmware in the section dedicated to BAS automation on its main website [31], and these often contain only the firmware file system.

The selection of these vendors was driven by the greater ease and accessibility of downloadable materials. Indeed, many others did not provide downloadable material in the same manner, requiring authenticated registrations or not offering their firmware online at all. This highlights the difficulty in obtaining such materials and how challenging it is for the academic world to conduct tests on products originating from the industrial sector.

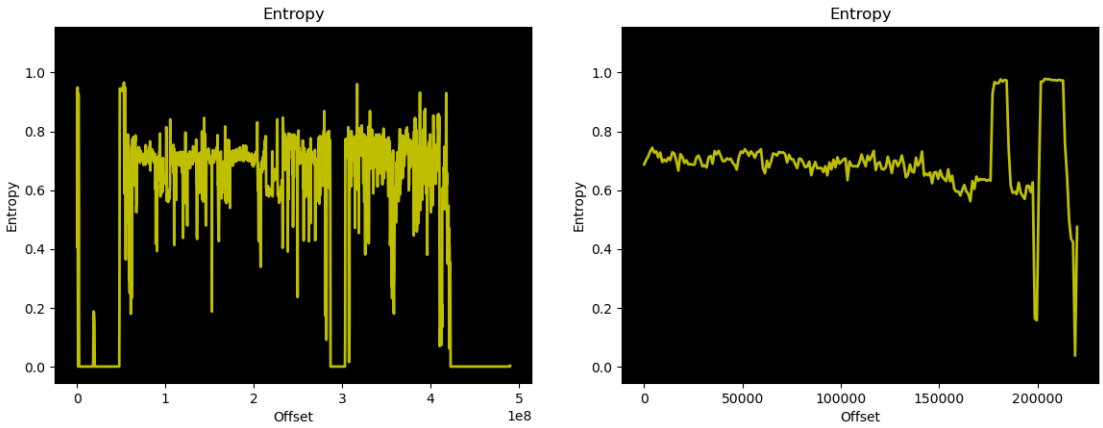
3.1.3 Firmware preliminary analysis

To incorporate firmware into the dataset while adhering to quality constraints, it was necessary to filter the material downloaded from the previously mentioned sources. In most cases, the firmware image was not the only piece of software available for download. Instead, the downloads often consisted of archives containing various materials, such as technical specifications, manuals, other types of software, auxiliary files in proprietary formats and the firmware image. The filtering process was conducted manually; however, in addition to identifying the actual firmware, it was crucial to assess its usability. Specifically, certain formats were deemed unsuitable for our analysis, including the following: Encrypted firmware and Firmware updates.

These two types are deemed unacceptable as they preclude the possibility of conducting a comprehensive analysis of the firmware. Encrypted firmware, in fact, is unusable because its content is obscured, while a firmware update provides only portions of the firmware rather than a complete image. During the manual analysis to identify the nature of the firmware, the decision was made by using Binwalk software [32] [33]. This tool can provide valuable information about the firmware, including entropy and composition, which are crucial for determining whether the firmware should be categorized as one of the two types to be excluded.

Unencrypted and uncompressed firmware

To understand the composition of a firmware image, it is necessary to unpack its contents. However, it is possible to analyse its entropy for an initial, immediate assessment. As introduced by Shannon in [34], in information theory, entropy measures the randomness or unpredictability of a data stream. Thus, entropy values in a file can indicate whether it is encrypted or compressed, or not. In figure 3.1, two plots are shown depicting the entropy of two firmware images, one for the WAGO CC100 device and one for the Contemporary Controls BASRT-B device. The x-axis represents the file offset, that is, the position of bytes from the beginning to the end of the file, while the y-axis indicates the corresponding entropy values. As can be seen, the entropy varies significantly in both cases, with values changing steeply in 3.1a, whereas they are locally more stable in 3.1b.



(a) CC100 V4.6.3 **(b)** BASRT-B V3.0.13

Figure 3.1: Entropy Analysis of Unencrypted and Uncompressed Firmware

Low and highly variable entropy values are indicative of repetitive patterns and ordered structures within the files. Uncompressed files and unencrypted data keep their original structure which is usually made of:

- Repetition: Certain byte values appear more often than others. For example, spaces and common letters dominate plain text.
- Predictable structures: Formats like XML or HTML contain repeated tags and have common headers.
- Patterns: Uncompressed images might have large areas of the same colour and file formats impose structure (e.g., a PDF file starts with %PDF-1.7).

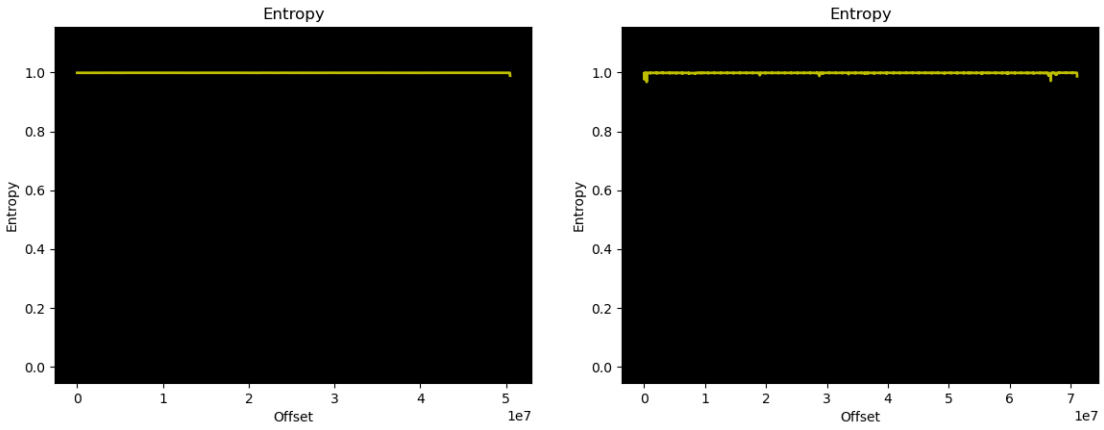
The detected entropy immediately reveals that the analysed files are neither encrypted nor compressed. It can be observed that the firmware shown in 3.1b contains two distinct high-entropy sequences, suggesting that these sections may contain encrypted data or compressed archives.

For the CC100 and BASRT-B devices, the firmware content is monolithic, embedded within files with the extensions *.img* and *.bin*, respectively. When the CC100's onboard software was analyzed using Binwalk, the output consisted of an extensive list of results—more than 1,000 files—of which only a subset is shown in A.1. These findings should be interpreted with caution, as Binwalk's analysis focuses on detecting file signatures, a process that is susceptible to false positives.

Encrypted and compressed firmware

The plots in figure 3.2 show the entropy analysis conducted on two other firmware images: one for the FS device from Johnson Controls and the other for the

QMX7.E38 device from Siemens.



(a) FS V3.0b62 Encrypted

(b) QMX7.E38 V01.16.53.44 Compressed

Figure 3.2: Entropy Analysis of Firmware: Encrypted vs. Compressed

The Binwalk analysis of the FS firmware is presented in Appendix A.2, showing a very short output. It reports an encrypted file using OpenSSL and the presence of a salt used during the encryption process. The entropy plot is consistent with the presence of the encrypted file, as high entropy throughout the data stream is what one would expect from the output of an encryption algorithm. When data is encrypted using strong algorithms (e.g. AES), the process transforms the original data into a completely new, pseudo-random sequence of bytes [35].

In the plot depicting the entropy of the QMX7.E38 device, there is a strong resemblance to the one previously described. The entropy is almost constant at the maximum value. However, there are slight fluctuations that are absent in the previous plot. If this firmware was also encrypted, there would be issues with the functioning of the encryption algorithm, as any fluctuation would indicate predictability in the data stream, and thus potential vulnerabilities. In fact, this firmware is not encrypted but compressed. The algorithms responsible for data compression work indirectly on entropy. They do not explicitly aim to increase entropy like encryption algorithms do, but since they seek to remove redundancy and represent the data more efficiently, as a side effect, their outcome has a very high entropy. Indeed, these algorithms exploit repeated patterns or frequent occurrences of certain values by replacing them with shorter representations.

Thus, it is possible to immediately distinguish between compressed and encrypted files simply by observing the entropy. Additionally, a more in-depth analysis with Binwalk reveals a series of compressed archives for the QMX7.E38 device firmware A.3.

3.2 Dataset structure

During the firmware collection process, it became immediately evident that a systematic approach was necessary to support the operations of downloading, analysing, and collecting data. To address this requirement, the dataset was structured in an organized manner to facilitate maintenance and streamline related operations.

The structure of the dataset is organized into three main components:

- **Firmware collection**: A collection of files and directories containing the firmware binaries and associated metadata, as downloaded from the official websites of the manufacturers.
- **firmwareDB.csv**: A CSV formatted file that stores metadata and descriptive information related to the downloaded firmware.
- **Scripts**: A set of Python scripts designed to perform various operations (e.g., consistency checks, analysis execution) on the dataset. These scripts ensure consistency between the `firmwareDB.csv` file and the actual firmware content.

3.2.1 Firmware collection

The downloaded firmware is stored in a systematic manner to ensure both consistency and modularity within the dataset. At the root level of the dataset, directories are organized by manufacturer, with each directory grouping all device models associated with that brand. Within each device model directory (a subdirectory of the corresponding manufacturer), folders are further organized by firmware version, as multiple versions may be stored for a given model. The resulting hierarchical structure can be represented as follows: `Manufacturer/DeviceModel/FirmwareVersion`.

Each firmware version directory contains the firmware exactly as it was downloaded from the manufacturer's official website. This may be a single file or a folder comprising multiple files and subdirectories. The firmware itself can take various forms, such as a standalone file (e.g., `.fmws`, `.bin`, `.img`, `.zip`, `.tar.gz`) or a directory containing multiple components. Additionally, any supplementary materials present at the time of download, such as documentation, auxiliary software, or metadata, are preserved within the same directory. These materials are retained as they may provide valuable context for understanding the purpose of the firmware and the functionality of the corresponding device.

3.2.2 firmwareDB.csv

The firmwareDB.csv file is the index of the entire dataset, with each row corresponding to a specific firmware version. Each entry is uniquely identified by three fields:

manufacturer, device model, and firmware version. Additional fields include the firmware release date, device functionality, supported protocols, CPU architecture, encryption status, EMBA emulation status, download URL, technical specifications URL, and the relative path to the firmware within the dataset.

Firmware release dates were retrieved from the manufacturers' websites when available. In cases where the release date was not explicitly stated, it was inferred through an analysis of the firmware files' metadata. Information on device functionality, supported protocols, and CPU architecture was primarily obtained from the manufacturers' websites. Typically, functionality and protocol support are clearly documented, as they are key selling points. In contrast, CPU architecture is rarely disclosed.

Although encrypted firmware was initially intended to be excluded, some encrypted samples were retained to enable further investigation into their structure and usability. To reflect this, the dataset includes an "encryption" column that flags whether each firmware sample is encrypted.

The "firmware relative path" field specifies the location of the firmware file or folder, relative to the corresponding `Firmware Version` directory.

3.2.3 Scripts

The scripts were developed to support and manage operations related to the dataset, with the primary goal of ensuring consistency between the index file and the stored firmware. Implemented in Python, the scripts operate in close interaction with the file system and the operating system. Their integration is essential for maintaining consistency between the firmwareDB.csv index and the actual firmware stored in the dataset. These scripts automate tasks such as consistency checking, analysis execution, monitoring the status of ongoing analyses, cleaning temporary files generated during analysis, and generating statistical reports according to predefined rules.

3.3 The dataset in numbers

This section presents several key figures related to the dataset to provide a clearer understanding of its composition. The aim is to briefly outline the types of firmware included and their respective characteristics, thereby offering a comprehensive overview.

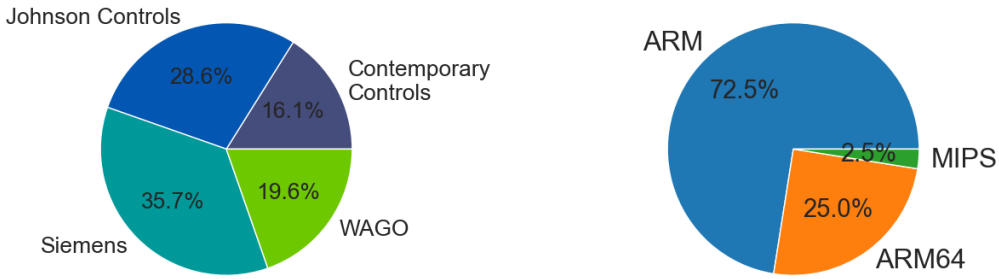
3.3.1 Overview

In the dataset are stored 56 firmware from 4 different manufacturers, the categorization of firmware is showed in Table 3.1.

Manufacturer	Nº of firmware
Contemporary Controls	9
Johnson Controls	16
Siemens	20
WAGO	11

Table 3.1: Number of firmware grouped by manufacturer

The dataset reveals a significant bias toward ARM-based architectures, with 72.5% of devices using ARM (32-bit) and 25% using ARM64 (64-bit). This brings the total ARM representation to 97.5%, highlighting its widespread adoption in embedded devices. In contrast, MIPS accounts for only 2.5%, indicating its rare presence in the dataset. All devices in the dataset use little-endian byte order, which aligns with the prevailing industry standard for most modern architectures such as ARM and x86.



(a) Vendors **(b)** Architectures

Figure 3.3: Vendors and Architectures

During the firmware download process, attention was paid to selecting a variety of software types. However, in order to perform comparative analyses, different firmware versions for the same device were also downloaded. Table 3.2 presents the number of firmware versions for each device, which are in total 35. This allowed us to observe that, in most cases, if one firmware version can be emulated, subsequent versions are likely to be as well. However, in certain instances, partial emulation was successful only for older firmware versions, failing with more recent ones.

Table 3.3 provides a summary of the dataset used in this study. Each row corresponds to a specific firmware and includes several key characteristics. The

Manufacturer	**Device Model**	**Nº of firmware per device**
ContemporaryControls	BASGLX-M1	2
	BASR-8M	1
	BASRT-B	1
	BASRTLX-B	2
	BASRTP-B	1
	BASpi-IO6U4R2A	1
	BASpi-IO6U6R	1
JohnsonControls	30P	1
	FC20	1
	FD-20i	1
	FG	3
	FS	3
	FT	1
	FW	1
	FW-28	2
	FW-8-8V-14	2
	FW-VAV	1
Siemens	DXR2	2
	PXC3	2
	PXC4	2
	PXC5.E003	2
	PXC5.E24	2
	PXC7	2
	PXG3	1
	PXG3.WX00-1	2
	PXG3.WX00-2	2
	PXM	2
	QMX7.E38	1
WAGO	750-891	1
	BC100	2
	CC100	2
	PFC	2
	PFC300	1
	TP600	2
	WP400	1

Table 3.2: Device models per manufacturer and their respective number of firmware

firmware is identified by its manufacturer, device model, and firmware version. Metadata regarding the devices, such as their intended functionality and the communication protocols they support, were extracted from the official websites of the respective vendors.

Among the various functionalities observed, the most common designation is "Controller". Although the term is employed with slight variations across manufacturers, it generally refers to a device designed for multiple purposes. Typically, a controller manages a network of peripheral devices either through physical interfaces or wireless communication (e.g., antennas). It also possesses internal logic for scheduling and executing operations on the devices it controls. Additionally, controllers are often equipped with local storage and computational capabilities that enable them to perform preliminary data analysis on information received from connected sensors.

Other characteristics reported in the table include the firmware's underlying architecture, endianness, and kernel version. However, this technical information is not available for all firmware samples. In some cases, the analysis could not reliably determine these attributes due to insufficient identifiable components. It's worth noting that all the images for which the OS could be detected use a Kernel Linux, outlining its predominance in the realm of building automation.

Finally, the table uses colour-coding to indicate the outcome of the firmware emulation process: green represents firmware where emulation was successful, blue indicates partial emulation, and white denotes failed emulation attempts.

The collected firmware samples exhibit considerable variance in size. Specifically, they range from a minimum of 183.39 KB for the FC20 device to a maximum of 4.17 GB for the BASpi-IO6U6R device. This significant variability, with a median size of 51.98 MB, can be attributed to several factors. Among these, we can observe the presence of compressed firmware, as well as those that have not been optimized for download. Additionally, the intended purpose of the firmware must be considered: some are designed to manage very simple devices (e.g., touch panel) and execute relatively trivial tasks, while others are closer to general-purpose computers (e.g., controllers at the management level). The choice of operating system also plays a role, as it may include basic kernels or, in some cases, additional functionalities that heavily influence the size. Finally, preinstalled software can significantly impact the size, particularly with the inclusion of images and other audiovisual content.

3.3.2 Protocols

During the inclusion of the firmware in the dataset, the supported network communication protocols of each device were recorded. Each device utilizes different protocols and may support various variants of the same protocol (e.g., BACnet/IP, BACnet/MSTP).

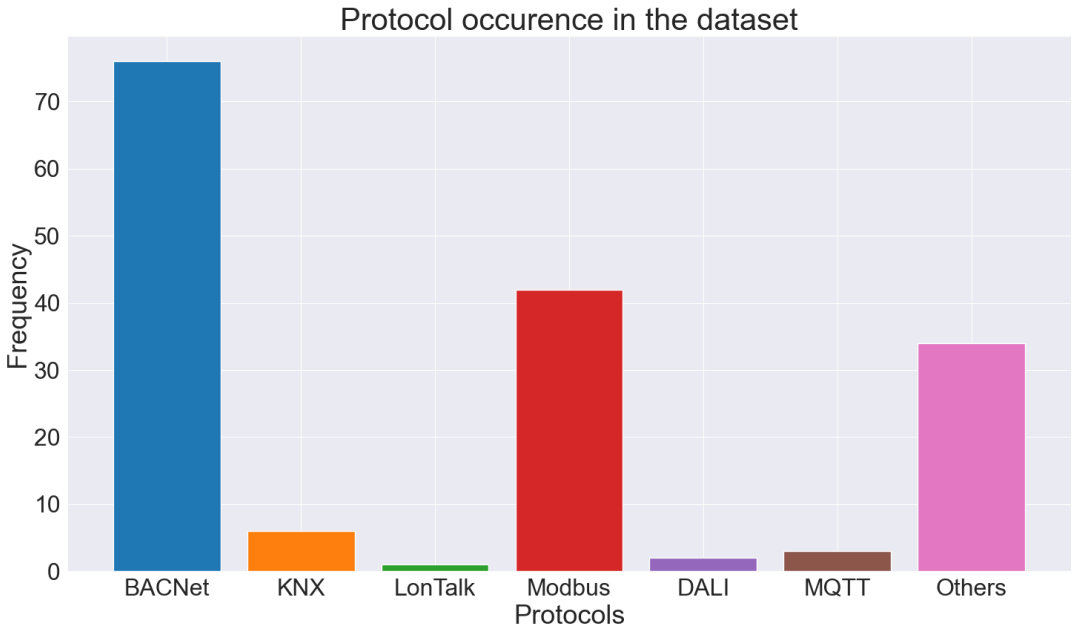


Figure 3.4: Frequency of Protocols in the Dataset

Figure 3.4 illustrates the frequency with which these protocols appear in the dataset. The most widely supported protocol is BACnet, with 76 occurrences. This is particularly noteworthy as it allows for the identification of a trend in the evolution of the most widely used protocols in the context of Building Automation Systems (BAS). In fact, BACnet appears to be gaining dominance, increasingly overtaking other protocols like KNX and LonTalk. However, it should be noted that the collected firmware primarily belongs to controllers, which are typically used for management and automation purposes. In contrast, protocols like KNX and LonTalk seem to be gaining traction in the field level.

Of the four manufacturers listed, three are European and only one is American. This seems to challenge the stereotype of a closed European market, traditionally associated with KNX, and suggests a shift towards broader acceptance of BACnet as an industrial standard.

Equally interesting is the significant presence of the Modbus protocol, with 42 occurrences. Originally designed for communication between industrial electronic devices, it is noteworthy that Modbus has found its place in Building Management Systems (BMS) as well. This trend can be attributed to the fact that industrial automation has been established and mature for a longer period, allowing its protocols to become more structured and adaptable for use in other domains.

Finally, it is evident that the MQTT protocol is still struggling to gain traction

in the BAS field, with its usage in the dataset being quite limited. The protocols listed as "others" encompass a wide range of protocols, often proprietary.

For the vendors Contemporary Controls, Johnson Controls, and Siemens, the most commonly used protocol is BACnet, whereas for WAGO, Modbus is more prevalent. Both Siemens and Johnson Controls tend to utilize less common protocols such as Tcom, Sox, and Island Bus. This is not the case for WAGO and Contemporary Controls, which predominantly rely on BACnet and Modbus. It is also noteworthy that BACnet is supported in many of its variants, ranging from MSTP and IP to its security extension, BACnet/SC.

Manufacturer	Device	Firmware	Functionality	Architecture	Endianess	Kernel
Siemens	PXG3	V01.21.152.8-3236	BACnet router	ARM	EL	Linux v4.4.302
Siemens	PXC3	V01.21.194.18-6633	Automation Station	ARM	EL	Linux v4.4.302
Siemens	PXC3	V01.21.172.22-6095	Automation Station	ARM	EL	Linux v4.4.302
Siemens	PXG3.WX00-1	V02.21.194.25-22980	Web Interface	ARM	EL	-
Siemens	PXG3.WX00-1	V02.20.172.47-21561	Web Interface	ARM	EL	-
Siemens	PXG3.WX00-2	V02.21.194.25-22980	Web Interface	ARM64	EL	Linux v5.15.71
Siemens	PXG3.WX00-2	V02.20.172.47-21561	Web Interface	ARM64	EL	Linux v5.10.35
Siemens	QMX7.E38	V01.16.53.44	Touch Panel	ARM	EL	Linux v3.18.10
Siemens	DXR2	V01.21.194.18-6633	Automation Station	ARM	EL	Linux v4.4.302
Siemens	DXR2	V01.21.172.22-6095	Automation Station	ARM	EL	Linux v4.4.302
Siemens	PXM	V02.21.194.25-22980	Touch Panel	ARM	EL	-
Siemens	PXM	V02.20.172.47-21561	Touch Panel	ARM	EL	-
Siemens	PXC4	V02.21.194.25-22980	Automation Station	ARM	EL	Linux v4.4.302
Siemens	PXC4	V02.20.172.47-21561	Automation Station	ARM	EL	Linux v4.4.302
Siemens	PXC5.E003	V02.21.194.25-22980	Automation Station	ARM64	EL	Linux v5.15.71
Siemens	PXC5.E003	V02.20.172.47-21561	Automation Station	ARM64	EL	Linux v5.10.35
Siemens	PXC5.E24	V02.21.194.25-22980	Automation Station	ARM64	EL	Linux v5.15.71
Siemens	PXC5.E24	V02.20.172.47-21561	Automation Station	ARM64	EL	Linux v5.10.35
Siemens	PXC7	V02.21.194.25-22980	Automation Station	ARM64	EL	Linux v5.15.71
Siemens	PXC7	V02.20.172.47-21561	Automation Station	ARM64	EL	Linux v5.10.35
JohnsonControls	FG	V2.0b52	Plant Controller	ARM	EL	Linux v2.6.39.4
JohnsonControls	FG	V2.0b51	Plant Controller	ARM	EL	Linux v2.6.39.4
JohnsonControls	FG	V1.5b51	Plant Controller	ARM	EL	Linux v2.6.29.2
JohnsonControls	FS	V3.0b62 *	Controller	-	-	-
JohnsonControls	FS	V3.0b51d	Controller	ARM	EL	Linux v3.4.39
JohnsonControls	FS	V3.0b55b *	Controller	-	-	-
JohnsonControls	FW-8-8V-14	V1.0b24 *	Controller	-	-	-
JohnsonControls	FW-8-8V-14	V1.0b16i	Controller	MIPS	EL	Linux v4.14.105
JohnsonControls	FW	V3.0b24 *	Controller	-	-	-
JohnsonControls	FW-28	V2.0b17a *	Controller	-	-	-
JohnsonControls	FW-28	V3.0b22a *	Controller	-	-	-
JohnsonControls	FW-VAV	V3.0b22a *	Controller	-	-	-
JohnsonControls	30P	V2.0.5.24	Controller	-	-	-
JohnsonControls	FC20	V2.2.07	Controller	-	-	-
JohnsonControls	FD-20i	V1.1.00	Controller	-	-	-
JohnsonControls	FT	V2.2b14	Controller	-	-	-
ContemporaryControls	BASRT-B	V3.0.13	BACnet router	-	-	-
ContemporaryControls	BASRTLX-B	V1.3.8	BACnet router	ARM	EL	-
ContemporaryControls	BASRTLX-B	V1.3.0	BACnet router	ARM	EL	-
ContemporaryControls	BASpi-IO6U6R	V1.0.33	Controller	ARM	EL	Linux v5.10.63
ContemporaryControls	BASpi-IO6U4R2A	V1.0.33	Controller	ARM	EL	Linux v5.10.63
ContemporaryControls	BASGLX-M1	V2.0.19	BAS gateway	ARM	EL	-
ContemporaryControls	BASGLX-M1	V2.0.1a	BAS gateway	ARM	EL	-
ContemporaryControls	BASR-8M	V3.7.8	BAS gateway	ARM	EL	-
ContemporaryControls	BASRTP-B	V3.0.13	BACnet router	-	-	-
WAGO	PFC	V4.6.1	Controller	ARM	EL	Linux v5.15.107
WAGO	PFC	V4.1.10	Controller	ARM	EL	Linux v5.15.19
WAGO	750-891	V1.6.4	Controller	-	-	-
WAGO	CC100	V4.6.3	Controller	ARM	EL	Linux v5.15.107
WAGO	CC100	V4.1.10	Controller	ARM	EL	Linux v5.15.19
WAGO	BC100	V1.4.7	Controller	-	-	-
WAGO	BC100	V1.2.0	Controller	-	-	-
WAGO	PFC300	V4.6.1	Controller	ARM64	EL	Linux v6.6.15
WAGO	WP400	V4.6.3	Touch Panel	ARM64	EL	Linux v6.6.3
WAGO	TP600	V4.6.1	Touch Panel	ARM	EL	Linux v5.15.107
WAGO	TP600	V4.2.13	Touch Panel	ARM	EL	Linux v5.15.86

Table 3.3: Summarizing dataset

The green rows report the firmware that were successful emulated
The blue rows report the firmware that were only partially emulated
For the other firmware the emulation failed
* indicates encrypted firmware

Chapter 4

Methodology

This chapter introduces the methodology adopted to conduct the analyses performed on the firmware. The process consists of multiple steps, each necessary to carry out a thorough investigation in accordance with the guidelines established by leading organizations in the field of embedded security. Finally, to scale the analysis and increase its efficiency, we relied on EMBA a tool that enables automation of several key stages in the workflow.

Firmware

Firmware is a low-level software essential to take control over a device's hardware and peripherals. It is typically embedded in the device's non-volatile memory (e.g., ROM, EEPROM, or flash memory) and can operate either as stand-alone software or in conjunction with an operating system. Executing directly on the microcontroller or processor, firmware manages critical functions such as hardware initialization, peripheral communication, memory management, and coordination with other system components.

A firmware is typically responsible for hardware initialization, controlling the boot-up process, and activating hardware components. Its role may include loading an operating system or operating directly on bare metal, managing the device without any additional software support. In simpler systems, firmware often handles all operational tasks, whereas in more complex architectures, it typically functions as an intermediary between the hardware and higher-level software layers.

Given its central role in defining a device's behaviour, firmware must be secure and free from vulnerabilities. Exploitation of firmware flaws can grant attackers low-level access, enabling manipulation not only of the targeted device but also potentially compromising the security of the network. This is a primary reason why proprietary firmware often poses a risk to system security; its closed-source nature and lack of independent review increase susceptibility to a wide range of

attacks.

4.1 Analysis methodology

The idea of developing a methodology emerged from the need to standardize the analysis of firmware. Initially, individual analysis were conducted on firmware samples in order to understand their structure and behaviour. This led to specific observations about each piece of software analysed, along with the necessity of collecting data in a structured way. What began as a rudimentary process gradually evolved into a more structured approach, ultimately requiring the creation of a dedicated pipeline through which all firmware samples could be systematically evaluated.

To learn how to perform firmware analysis and to define the necessary steps, inspiration was drawn from studies such as [15, 17, 36]. Additionally, the OWASP website [37] proved to be a fundamental resource, not only supporting the work overall but also aiding in the breakdown of the process into discrete steps, in accordance with established methodologies for firmware analysis.

The methodology is structured into four phases:

1. Fetching Firmware
2. Firmware Extraction
3. Firmware Analysis
4. Firmware Emulation

These phases therefore cover the entire process, from acquisition, through the analysis of the firmware components, up to its emulation.

4.1.1 Fetching Firmware

The first step in the process involves obtaining the firmware to enable subsequent analysis. This step is described in detail in subsection 3.1.2. Therefore, the following discussion aims to briefly outline the options considered for acquiring software related to BAS.

The first option consists of searching directly on the websites of vendors. Manufacturers sometimes provide the software used in their devices on their official websites, allowing users to reinstall or update the firmware if newer versions have been released. However, in the domain of smart buildings, publicly accessible software is relatively scarce, reflecting a broader tendency to restrict access to this kind of material.

Another possibility is to consult online repositories, whether open access or made available specifically for research purposes. Nevertheless, repositories containing firmware samples directly related to BMS were not found.

An interesting alternative approach involves browsing online forums dedicated to firmware support. In such forums, users occasionally upload firmware files when seeking technical assistance. Similarly, video tutorials on platforms like YouTube may sometimes include links to cloud storage or repositories from which firmware can be downloaded.

Finally, an approach employed in studies such as [4, 16] involved purchasing the physical devices in order to obtain the firmware. The firmware is extracted by performing an hardware dump, leveraging interfaces available on the integrated circuits, such as UART and JTAG ports.

For the dataset created in this thesis, the decision was made to include only firmware obtained from vendors through manual extraction. While this approach could be improved by developing a web crawler to automate the download process, it presents several challenges. Many websites explicitly prohibit the use of web crawlers, and any automatically retrieved firmware would still require manual verification. The option of performing hardware dumps was not considered, as the objective of this work is to analyse a substantial number of firmware samples. IoT devices tend to be quite expensive, with prices ranging from several hundred dollars for the most affordable models to over a thousand dollars for others. Acquiring a significant number of devices for firmware extraction would lead to prohibitively high costs, making large-scale analysis economically unfeasible.

4.1.2 Firmware extraction

The second step involves extracting the firmware in order to analyse its fundamental components. Firmware can be distributed in various formats: as a single file, a compressed archive, an encrypted file, or as a collection of files and directories. As discussed in subsection 3.1.3, it is possible to distinguish between encrypted, compressed, and regular files through entropy analysis.

Once it is confirmed that the firmware is not encrypted, the first task is to identify its internal components. If the firmware is compressed, the appropriate decompression algorithm must be determined, this can often be inferred from the file extension, so that the archive can be properly unpacked.

Tools and procedure

When the firmware is provided as a single file, a binary blob, it becomes necessary to analyse its structure in order to identify its components. To accomplish this task, `Binwalk` [32, 33] is the tool of choice. Binwalk is designed to carry on binary

analysis, especially for firmware images, helping in inspecting the structure of these files and extract hidden components. By identifying byte sequences, file signatures, and structural patterns, Binwalk can effectively parse the binary blob and detect its constituent parts. Moreover, it is capable of automatically decompressing recognized sections and extracting their contents, including recursively unpacking nested structures when present. Additionally, the `firmware-mod-kit`, a utility built on top of Binwalk, can be employed to enhance this process. It offers extended functionality and broader file type recognition, including support for certain proprietary formats that Binwalk alone may not detect.

In some cases, automated extraction may not be feasible using Binwalk or other automated tools such as `Binextractor`. However, it is still possible to identify offsets that indicate where specific components are located within the data blob. These offsets can be discovered using Binwalk itself or through data recovery tools such as `Foremost` e `Scalpel`. The components can then be manually extracted using the command-line utility dd, which allows for precise slicing of files based on specified offsets.

At the end of this procedure, a collection of files separated by functionality is obtained. However, in some cases, the firmware may already be distributed in this way from the outset. When that is the case, it is often possible to infer the function of each component from the filenames. If the filenames are not descriptive, several command-line tools can be employed to perform further investigations.

The `file` command in Linux is employed to determine the type of a file by examining its content rather than relying on its extension. It performs a series of tests to classify the file, such as detecting specific data formats and distinguishing between text and binary files. Additionally, valuable information is often embedded within files in the form of strings. In this context, the `strings` command is particularly useful, as it can extract printable character sequences from data files. These strings can then be examined manually or filtered using tools such as `grep` to uncover relevant information. Another useful utility for data inspection is `Hexdump`, a hex editor that displays the contents of files in hexadecimal and other representations. It can be used to detect file signatures and to localize strings inside a file.

Firmware components

The following section outlines the most commonly encountered components within firmware and their respective roles.

Every device requires a bootloader capable of initializing its software, a process common to both advanced computers and embedded systems. It is not unusual to encounter files named bootloader, SPL (Secondary Program Loader), MLO (Memory Loader), and similar. These software components are specifically tailored

for SoC for which they are designed. They typically contain memory addresses and configuration data necessary to initiate the next stage loader or the operating system itself.

To support the loading of an operating system, the final-stage bootloader often includes an environment configuration file. A widely used bootloader in embedded systems is U-Boot, which commonly utilizes a file named uEnv.txt. Such files define environment variables and parameters that are essential for configuring the kernel prior to its execution.

Another widely used file in embedded systems is the device tree. While advanced hardware platforms often feature buses with built-in discoverability mechanisms (e.g. PCI, USB) many other buses lack this capability. Embedded systems, in particular, make extensive use of non-discoverable buses (e.g. I2C, SPI). In scenarios where on-board hardware and peripherals cannot be automatically detected, a formal hardware description is required. This need is addressed by device tree files, which provide a structured description of the hardware components present on a board. These files also specify the physical memory addresses at which each device is located, enabling the bootloader or operating system to interact with them. With this information, the system can identify the components, their associated buses, communication protocols, and addresses. Device tree files are typically stored in the binary format `.dtb` (Device Tree Blob), but they can be decompiled into the human-readable `.dts` (Device Tree Source) format using the command-line utility `device-tree-compiler`.

One of the most critical components of firmware is the **kernel**, which serves as the core of the operating system. It manages communication between hardware and software and controls essential system resources such as memory and processing power. In IoT devices, it is common to find either an embedded Linux kernel or a Real-Time Operating System (RTOS) kernel. Bare-metal software, where no kernel is present, is significantly less common and was not observed in the devices analysed. Kernel images typically use one of the following file extensions: `.bin`, `.img`, `.zImage`, or `.uImage`. The `.bin` and `.img` formats are generally used for raw or generic binary images. The `.zImage` format represents a compressed Linux kernel image that includes a self-extracting mechanism. In contrast, the `.uImage` format wraps the kernel with a U-Boot header, containing metadata such as the operating system type and loading information.

Finally, the firmware image contains the file system. It constitutes the core of the firmware, as it contains the majority of system information including user accounts, passwords, and all installed programs. File systems can come in a wide variety of formats, such as SquashFS, JFFS2, UBIFS, cramfs, ext, YAFFS2, FAT, exFAT, and ROMFS.

Additionally, other files may be present in some firmware images but not in all;

these can be specific to certain vendors or products. Nevertheless, the aforementioned files represent the most common and essential elements. A representative example of the contents of a firmware image is provided in Appendix A.3.

4.1.3 Firmware analysis

Once the various components have been extracted, the next step is to proceed with the firmware analysis. This analysis aims to understand how the different components interact with each other and to conduct an examination of the file system in order to identify interesting artifacts.

Boot process

First, a clear depiction of how firmware initializes and how the previously described components interact is necessary. Firmware booting typically consists of three main phases: the initial system boot driven by low-level software, the execution of the kernel, and finally the execution of user-level programs.

During the boot stage, usually two or three bootloaders are involved. When the SoC is powered on, the ROM bootloader is executed. Its address is specified in the reset vector, and its primary task is to load the first stage bootloader into the SoC's internal RAM. This step is essential because, at power-up, the CPU cannot access external memory (RAM), so a minimal first-stage bootloader, small enough to fit into the internal RAM, must be loaded to initialize memory control. Once the CPU gains control over the memory, the second stage bootloader takes over.

The second stage bootloader is responsible for reading configuration settings and loading the kernel. It must identify the location of the kernel image and the device tree file, then load the kernel into RAM while passing the necessary arguments to establish the kernel's environment. After completing these steps, control is transferred to the kernel, which then sets up its environment and initializes hardware components using the device tree.

In the final stage, the operating system completes its setup. Using boot arguments, it locates and mounts the root filesystem. Subsequently, the init process is launched to start the user space. Finally, the init process spawns user-space processes according to its configuration file.

Information extraction

Files such as the bootloader and the device tree are often crucial to inspect. They can reveal key information about the processor architecture and more. Specifically, they provide details about the types of installed memory, including RAM, ROM, and flash memory. Additionally, the device tree can offer extensive information about the hardware present on the integrated circuit, potentially allowing identification of

the specific board in use, an information that can be essential during the emulation phase.

However, the most compelling element to analyse is the filesystem. It can be examined manually by navigating through its structure to understand its composition and to identify the files and directories. Through this process, valuable information for dynamic and runtime analysis can be gathered.

By analysing binary files containing executables, valuable information about the system architecture and operating system can be obtained. In UNIX-like systems, directories of particular interest for these types of files include `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin`, which typically contain general-purpose executables related to default system utilities, as well as `/lib`, where dynamically loaded kernel modules reside. Using the command-line utility `Readelf`, it is possible to inspect various components of an executable, such as headers, segments, symbols, and more. When combined with the tool `checksec`, it allows verification of several security features implemented in executables, including stack canaries, RELRO (Relocation Read-Only), NX (No Execute), and PIE (Position Independent Executable). Furthermore, it is often valuable to analyse binaries specifically created for the firmware in question, as proprietary software can frequently harbour security vulnerabilities if not thoroughly audited.

Other important files to review are the various scripts. These can be analysed to understand their functionalities and to determine whether they automatically execute actions at startup, such as configuring system files or launching programs. The `/etc` directory is of particular interest when investigating system settings. It may contain web server configurations, the definition of the init script, certificates, and, more generally, a variety of valuable information for the analysis.

It is vital to check for the presence of network-related daemons and servers. If these are not properly updated or inherently secure, they can serve as easy entry points for attackers. Additionally, it is always advisable to search for hardcoded credentials within configuration files or server binaries. Although this is a highly insecure practice, it is still frequently used or forgotten after debugging, thereby making the entire systems vulnerable.

Many of these search operations can be automated using the tool `Firmwalker`. This tool scans the filesystem for items of interest such as common web servers used on IoT devices, binaries such as ssh, tftp, dropbear, configuration files and passwords.

4.1.4 Firmware emulation

Ultimately, the process aims to emulate the firmware, or specific components of it, in order to perform dynamic evaluations. This step enables a more comprehensive assessment of the software under analysis by observing the runtime behaviour of

certain components.

To perform emulation, an emulator is required; in this study, QEMU (Quick Emulator) was employed for this purpose [38]. QEMU is an open-source emulator capable of emulating a broad range of CPU architectures. It supports various instruction set architectures (ISAs), including x86, x86-64, MIPS, ARM, PowerPC, RISC-V, SPARC, ETRAX CRIS, and MicroBlaze. Its versatility lies not only in its ability to emulate CPU instructions, but also in its capacity to replicate hardware components. QEMU can emulate nearly the entire system-on-chip (SoC), encompassing the processor, memory, and peripheral devices such as network adapters.

To enhance the fidelity of the emulation, QEMU provides architecture-specific virtual boards that replicate the corresponding on-board hardware, offering high reliability in execution. QEMU supports three primary modes of operation:

- **User-mode emulation**: This mode allows for the emulation of individual executables. It is limited to the instructions of a single binary, enabling the execution of programs compiled for different architectures and operating systems on an x86 host system.

- **System emulation**: This mode emulates an entire system, including the CPU, memory, and peripheral devices, thus allowing complete board-level emulation.

- **Hypervisor support**: In this mode, QEMU functions either as a Virtual Machine Manager (VMM) or as a backend for device emulation in virtual machines managed by an external hypervisor.

During the emulation process, both user and system modes were employed. The user mode enabled attempts to emulate individual files; however, the focus was primarily on system emulation. This approach is more comprehensive, as it allows the full firmware to be emulated, from the boot sequence to its operational state. Such emulation enabled runtime analysis of the firmware, allowing for testing of programs within the file system as well as available network services. A more detailed discussion of this process, including two practical examples, is provided in Section 5.2.

One of the main challenges associated with system emulation is debugging. Emulating an entire system is inherently complex, as it requires the accurate replication of critical components such as certain kernel modules and dynamic libraries. These dependencies often cause the emulation to fail, necessitating a time consuming effort to identify and replace the problematic components. Furthermore, emulating the network stack adds an additional layer of complexity and is not always feasible, which can further limit the scope of dynamic testing.

It is also important to highlight that devices within BAS frequently rely on communication protocols that utilize specialized hardware interfaces. For example, BACnet MS/TP and Modbus RTU require physical ports and cabling that comply to their respective standards, and do not operate over standard Ethernet. QEMU, by contrast, supports emulation of the ISO/OSI network stack only for protocols that operate over Ethernet and IP-based technologies. This results in inherent limitations when attempting to fully emulate devices that depend on such hardware-specific communication standards.

4.2 Scaling the analysis

One of the objective of this thesis is to apply the previously described methodology to the entire dataset. This goal prompted a consideration of how to scale the process from a manual, sequential analysis of individual firmware images to a broader, parallelized evaluation. This transition highlighted the necessity of automation as a means to improve efficiency and scalability. Consequently, efforts were directed toward identifying tools capable of automating portions, or ideally the entirety, of the analysis workflow. This search led to the adoption of EMBA [5], a tool well-suited for comprehensive firmware analysis.

4.2.1 EMBA

EMBA is a tool designed for penetration testing focused on executing firmware analysis. It is able to perform various actions such as firmware extraction, static analysis and dynamic analysis via emulation. EMBA aims at automatically discover possible weak spots and vulnerabilities in firmware, by investigating insecure binaries, old and outdated software components, potentially vulnerable scripts, or detecting hard-coded passwords.

EMBA is written almost totally in bash scripting language and has a long list of dependencies helping it to achieve its goal. It relies on Binwalk, Ghidra, Radare2, QEMU, NMAP and a long list of other tools and command line utilities. It supports multiple operational modes, which can be selected via configurable options, enabling either a comprehensive or a targeted, partial analysis of the firmware.

EMBA's operations can be assimilated to the methodology depicted previously. Through the command-line interface, it is possible to specify a folder or file containing the firmware, and, by using the appropriate options, both static analysis and emulation can be executed. As first step EMBA analyses the composition of the firmware by identifying the main parts of the firmware image (e.g. zImage, file system, bootloader), along with the foremost information for creating an emulation environment (e.g. architecture, OS).

Once the relevant components are identified, EMBA proceeds by analysing them separately, with a primary focus on the kernel and the file system. Three dedicated modules are responsible for detecting kernel related vulnerabilities. These modules determine the kernel version and other characteristics to compile a list of known vulnerabilities associated with that configuration. Additionally, three other modules are tasked with analysing the file system, focusing on the detection of potentially vulnerable artifacts. This includes identifying insecure packages and performing static analysis of binaries and scripts to uncover potential vulnerabilities.

After extracting useful information from the previous steps, EMBA attempts to launch the emulation environment. This phase is non-trivial, as it must overcome numerous obstacles, including missing files and misconfigurations. EMBA addresses these issues by automatically detecting and correcting errors, repeatedly attempting to achieve a successful emulation.

This stage involves several key challenges. First, a suitable kernel image must be selected to handle hardware-specific firmware calls effectively. Subsequently, the original file system is modified to include a dedicated directory containing various diagnostic tools and an enhanced version of BusyBox, which provides a more comprehensive set of command-line utilities. This directory also includes a Bash script responsible for configuring the network interface.

A significant contribution of EMBA to the emulation process is its capability to identify and test a range of potential initialization (init) files. EMBA iteratively attempts to boot the system using these files, switching between them until it finds one that successfully initiates the system and activates the network interface. If both the system boot and network configuration succeed, the emulation is considered successful and a QEMU image is generated. Otherwise, the result may be a partial emulation, where the system boots but the network remains inactive, or a complete failure to emulate.

At the conclusion of all analysis phases, EMBA generates comprehensive reports summarizing the results obtained. These reports include a wide range of files detailing identified vulnerabilities, insecure services, and, when possible, an assessment of the network services exposed by the firmware.

4.2.2 Supporting scripts

To support EMBA's operation, a set of Python scripts was developed to assist both during the initialization phase and after the final results are generated.

EMBA analyses can be highly time-consuming, particularly during emulation attempts, which may take between 7 to 10 hours per firmware image. The scripts enable remote execution of the analysis on a server via SSH, providing functionality to monitor the progress of the analysis while it is running.

A crucial aspect of this workflow is the consolidation of the information produced

once the analysis is complete. Although EMBA generates a comprehensive web-based report, the relevant data is scattered across numerous files in various formats. Through a manual review process, the most informative output files were identified. By employing regular expressions (regex), it was possible to extract and organize key data into structured tables that summarize the most relevant analysis results. Finally, the scripts also facilitated the generation of aggregated statistics and summary tables, which are instrumental in assessing the status and characteristics of the analysed firmware samples.

Chapter 5

Results

5.1 Statistics

In the following sections, data will be presented concerning: the composition of the dataset, the vulnerabilities identified within the firmware, and statistics derived from the collected information. The primary objective is to analyse the data obtained through the information extraction process applied to the dataset, with the aim of providing a comprehensive overview of the current state of firmware development for devices employed in BMS.

5.1.1 High level data analysis

The analysis conducted over the firmware images had as objective to detect key components within the firmware, including: architecture, kernel and various pieces of software used. Out of 56 firmware samples, the architecture of 40 was successfully identified, as shown in Table 5.1 (the endianess of all of them is Little Endian). Of the remaining images, six were encrypted, and ten could not be unambiguously identified by EMBA.

Architecture	**Nº of firmware**
ARM	29
ARM64	10
MIPS	1

Table 5.1: Number of firmware grouped by architecture

The data collected from the dataset clearly indicates that ARM is the predominant architecture among the devices analysed. This observation is not aligned with trends in the IoT ecosystem. Previous studies, such as those by Kim et al. [20] and

Chen et al. [19], report that IoT firmware across various domains predominantly targets the MIPS architecture, with ARM accounting for only 10% to 25% of observed cases.

Table B.1 in the appendix provides an overview of the analysed firmware images. The table presents aggregated statistics obtained from the EMBA automated analysis tool and includes data for 31 firmware samples. For the remaining 25 images, it was not possible to identify the kernel version. Kernel information was successfully identified for all the 31 firmware, however, the GCC version used was detected in only 29 cases. The two undetected instances are both from the vendor Contemporary Controls. Across all analysed firmware images, a total of 48,692 Common Vulnerabilities and Exposures (CVEs) were identified, the vast majority of which pertain to the kernel. This results in an average of approximately 1,570 CVEs per firmware image. When considering only those vulnerabilities assessed using the CVSSv3 scoring system, the distribution is as follows:

- Number of Critical CVEs [Score: 9.0-10.0]: 237
- Number of High CVEs [Score: 7.0-8.9]: 14,270
- Number of Medium CVEs [Score: 4.0-6.9]: 31,220
- Number of Low CVEs [Score: 0.1-3.9]: 831

The overall average severity score across all CVSSv3 rated vulnerabilities is approximately 6.1.

Table 3.3 gives an overview of the results of firmware emulation. Among the 56 firmware 30 were emulated, 15 partially and 15 completely. The EMBA tool achieves emulation through a trial and error process that can lead to different outcomes. This process must ensure not only the correct initialization and booting of the system but also the proper configuration of the network interface. The emulation process can be split in 4 phases:

- Booting
- IP detection
- ICMP packets exchange
- Nmap scan

The first step involves detecting whether emulation through the QEMU emulator has actually started, which is verified by the booting process. Once the firmware successfully completes the startup phase, without encountering technical issues related to hardware emulation, the next step is to determine the IP address. If

the IP interface is set, the functionality of the network layer protocol is tested by sending and receiving ICMP packets. Finally, if this stage is also successfully completed, a scan is performed using the Nmap tool, which identifies open services on the firmware, thereby providing an overview of the available services. This list, however, may not be exhaustive, as not all services listen on standard ports, and firewalls may filter traffic, preventing some services from being detected. In the analysis conducted across 30 firmware the following stages were reached:

- Number of firmware with detected IP addresses: 1
- Number of firmware capable of exchanging ICMP packets: 14
- Number of firmware successfully scanned with Nmap: 15

A comprehensive overview of the software running within the analysed firmware is presented in Table B.2 in the appendix. The table reports for each firmware the services detected in two distinct analysis phases. The first phase involves EMBA probing the system during startup to detect services launched for debugging purposes. These services can later be verified during emulation by using the *ps* command. In the second phase, once the firmware has successfully completed its boot process, network services are scanned using Nmap.

EMBA leverages Nmap to perform this task using an approach that consists of two scans. Initially, a general scan is executed on well-known ports. Following this, a tailored scan is carried out. To guide this second phase, the *netstat* command is used to identify services actively listening on network interfaces. Based on this information, a refined scan is launched, focusing on the specific ports detected. The results of both scans are made available into separate reports. It is important to note that this automated analysis may fail to detect some services even if they are running. As such, while the results offer valuable insights into exposed services, they may not be exhaustive. The final summarized findings are reported in the table.

The first eight firmware images (from PXG3 to PXM) correspond to devices released by Siemens. A notable observation is the widespread use of web servers such as *nginx* and *lighttpd*. These servers are embedded within the firmware to provide services either to other devices or to a network administrator, who can access device functionality via a graphical web interface. Given their purpose, these services are typically discoverable by other devices within the same network. Nmap was able to detect the presence of web servers in all cases except one. Another frequently observed service is labelled as *ba-device*. Although it is not a well known program, online research [39] suggests that it is likely a daemon responsible for handling BACnet communications. This hypothesis is plausible, given that, as discussed in the dataset chapter, many devices in the BAS domain use the BACnet protocol. However, this service was not detected by Nmap during the automated

scans, nor was it identified in subsequent, more targeted scans specifically designed to detect BACnet traffic. It was only observed during the boot phase, when EMBA detects services launched at system startup.

The firmware images ranging from FG to FW-8-8V-14 were released by Johnson Controls. These firmware versions expose a wide variety of services. As with the previously discussed firmware, web servers are also present in this group. Additionally, the service *tls_tunnel* is deployed to provide security at the transport layer. Among the more commonly observed services are daemons for SSH and FTP, which allow secure remote access to the device and support file transfers, respectively. Other notable services include *nmbd* and *smbd*, components of the NetBIOS suite responsible for session layer communication and file/printer sharing capabilities. Furthermore, some firmware samples utilize Redis and SQL databases for storing information collected from other devices. The presence of *mosquitto*, a lightweight message broker that implements the MQTT protocol, also reflects a trend toward publish-subscribe messaging in these systems. This tendency is confirmed by *redis-server* being not only able to host an in-memory database but also acting as a message broker supporting publish-subscribe messaging framework to send messages between services in real-time.

The remaining firmware images (from PFC to TP600) were released by WAGO. The services in these images cover an heterogenous range of purposes. Similar to previously analysed firmware, they include web servers, along with FTP and SSH daemons. Additionally, the *inted* service is present, which is used to intercept network requests and spawn the appropriate process in response to client demands.

It is worth noting that firmware from Contemporary Controls is not included in this section. Although two samples from this vendor were successfully analysed, neither of them could be emulated, preventing further inspection of the services they might expose.

5.1.2 Statistical Analysis

In this section, the data obtained from the analysis of various firmware samples are examined to generate aggregate statistics and temporal trends aimed at providing insights into the security of their development processes. The information covers several aspects of the firmware, including kernel versions, GCC versions, onboard software versions, and the related vulnerabilities identified.

To analyse vulnerabilities that are present in firmware, it was necessary to develop an infrastructure that could detect such vulnerabilities and collect related data. The tool EMBA was employed for extracting vulnerabilities. It is able to perform a comprehensive analysis of the firmware image, initially checking whether the kernel is present and subsequently analysing other common pieces of software. Upon completion of this procedure, EMBA creates a set of heterogeneous reports

summarizing the vulnerabilities identified for every component being researched.

To retrieve specific information on identified vulnerabilities, the NVD database maintained by NIST has been utilized. NVD is the U.S. government repository of standards based vulnerability. It includes databases of security checklist references, security-related software flaws, product names, and impact metrics. The entire dataset was downloaded in JSON and imported into a MongoDB database for analysis. This was due to the fact that the format of vulnerability entries is very heterogeneous, and a non-relational database was thus the best choice for flexible querying and data handling.

Finally, a Python script was used for extracting data, querying the NVD database, and inserting the relevant information into a relational PostgreSQL database. The script extracts CVEs identifiers from EMBA output files and, and using the info in MongoDB can populate any gaps in missing data in PostgreSQL databases. This approach enables continuous updates of the analyses, even when new firmware images are introduced.

CVSS analysis

The analysis of 31 firmware samples revealed a total of 48,692 CVEs. The identified security vulnerabilities span a broad time range, from 1998 up to the date of the firmware analysis. As a result, the dataset includes vulnerabilities assessed using various versions of the base score metric, which is employed to determine their severity. [40] The Common Vulnerability Scoring System (CVSS) provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes. The evaluation systems present in the dataset include CVSSv2 and CVSSv3. CVSSv2 was introduced in 2007, while CVSSv3 was designed to correct shortcomings in v2 and was introduced in 2015. The identified vulnerabilities fall into three categories: those containing only a CVSSv2 base score, those containing only a CVSSv3 base score, and those for which both assessments are available. Specifically, the database includes 3,594 unique CVEs: 611 are assessed solely with CVSSv2, 1,792 solely with CVSSv3, and 1,178 include both scoring versions. The remaining 13 CVEs are marked as 'rejected', indicating that they were initially added to the NVD database but were later deemed invalid.

Given the differences between the two scoring systems, an analysis was conducted to assess the consistency between them. To this end, only vulnerabilities with both CVSSv2 and CVSSv3 scores were selected. The absolute difference between the two scores was calculated for each vulnerability, resulting in the distribution shown in Figure 5.1. The mean absolute difference was found to be 1.445. This

is consistent with the difference reported by Santos [41] which states that "the average base score increased from 6.5 (CVSSv2) to 7.4 (CVSSv3)" with a mean difference increase of 0.9. (To be noted that the mean difference increase of 0.9 is computed without the absolute value).

Based on:

- An average absolute difference of 1.445 between CVSSv2 and CVSSv3 base score
- The number of vulnerabilities containing only CVSSv3 being considerably larger than the ones containing only CVSSv2
- The CVSSv3 is being released more recently to overcome the shortcoming in CVSSv2

the subsequent analyses consider only vulnerabilities with a CVSSv3 base score to ensure greater consistency in severity assessment.

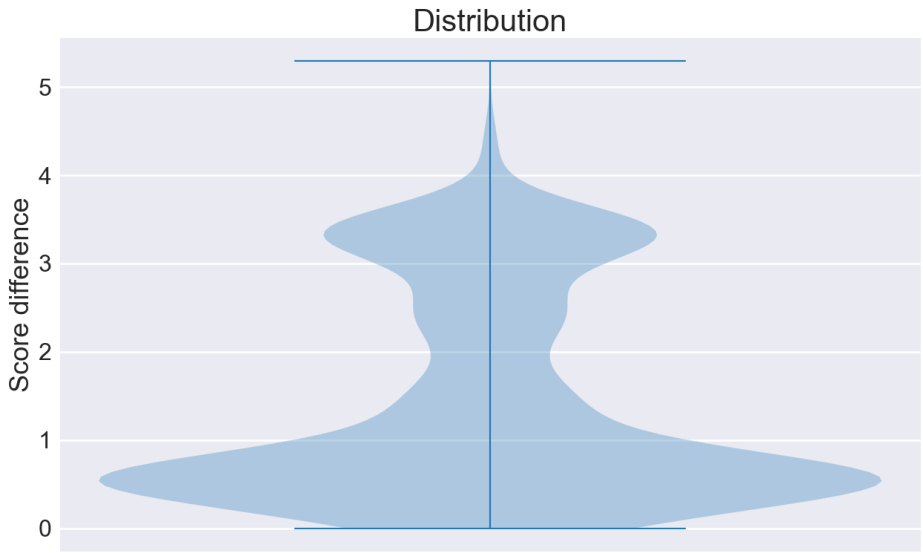


Figure 5.1: Comparison between CVSSv2 and CVSSv3 score

Among the various aspects considered in the analysis, one of the most significant is the presence of vulnerabilities within different firmware images. Indeed, in order to assess the current security posture of firmware, identifying the vulnerabilities it contains is essential to understanding and reporting its level of security.

In the constructed database, the vast majority of identified vulnerabilities pertain to the firmware's kernel. Only 26 vulnerabilities, related to two different versions of Dropbear SSH found in four distinct firmware instances, do not involve the kernel.

For the purposes of analysis, it was observed that the 31 firmware samples include only 14 distinct kernel versions. Consequently, in evaluating the vulnerabilities, a decision was made to focus on unique vulnerabilities rather than counting repeated occurrences across multiple firmware versions. This approach reduced the total number of vulnerabilities from 48,692 to 3,594. Thereby offering a clearer picture of the distinct issues present across the dataset

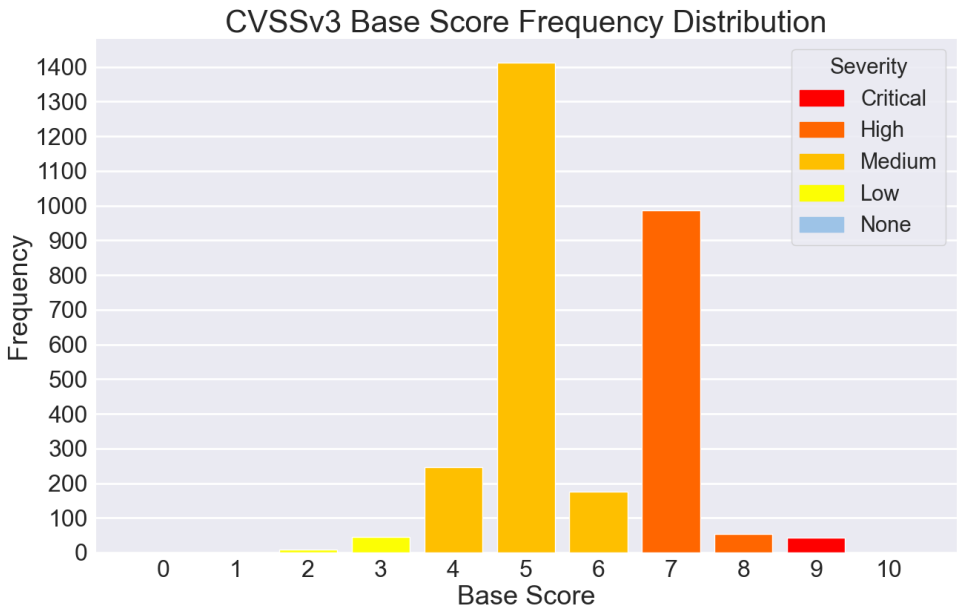


Figure 5.2: CVSSv3 CVEs distribution

Figure 5.2 presents the distribution of vulnerability severity base scores, categorized according to the NVD CVSS ratings [42]. The scores, which range from 0 to 10 with a precision of one decimal place, are binned in the figure into integer values only (e.g., scores ranging from 5.0 to 5.9 are labelled as 5). The graph clearly illustrates that the majority of vulnerabilities are concentrated between scores of 5 and 7, with an average score of 6.25. This distribution indicates that vulnerabilities predominantly fall within the medium to high severity categories, with relatively few categorized as low or critical. These findings suggest that most firmware vulnerabilities are sufficiently severe to be exploitable, though not necessarily critical. However, it is important to note that the vulnerabilities are primarily associated with the kernel component of the firmware. A deeper analysis

of other firmware components, particularly those tailored to specific vendors, would be necessary to gain a comprehensive understanding of their security posture.

CWE ID	Frequency	Name	Description
CWE-476	557	NULL Pointer Dereference	A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.
CWE-416	482	Use After Free	Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.
CWE-362	196	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.
CWE-401	164	Missing Release of Memory after Effective Lifetime	The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.
CWE-200	162	Exposure of Sensitive Information to an Unauthorized Actor	The product exposes sensitive information to an actor that is not explicitly authorized to have access to that information.
CWE-787	152	Out-of-bounds Write	The software writes data past the end, or before the beginning, of the intended buffer.
CWE-119	149	Improper Restriction of Operations within the Bounds of a Memory Buffer	The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.
CWE-125	134	Out-of-bounds Read	The software reads data past the end, or before the beginning, of the intended buffer.
CWE-20	116	Improper Input Validation	The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.
CWE-667	98	Improper Locking	The software does not properly acquire or release a lock on a resource, leading to unexpected resource state changes and behaviors.

Table 5.2: Top 10 most frequent CWE identifiers in the data set

To classify vulnerabilities not only by severity but also by type, the CWE system [43] provides a comprehensive taxonomy. Maintained by The MITRE Corporation, CWEs are often assigned to CVE upon approval. This association enables a more precise characterization of the nature and impact of each vulnerability, facilitating a deeper understanding of which aspects of a system are affected.

Table 5.2 summarizes the top 10 most frequent CWEs identified during the analysis of firmware CVEs, in conformity with NVD directives [44]. The table omits 372 CWEs labelled as NVD-CWE-noinfo and 113 labelled as NVD-CWE-Other. NVD-CWE-noinfo are CWEs where there is insufficient information about the issue to classify it, while NVD-CWE-Other is due to the fact that NVD is only using a subset of CWE for mapping instead of the entire CWE, and the weakness type is not covered by that subset. The majority of CWEs are related to memory safety

issues, as shown by the following vulnerabilities: NULL pointer dereference (CWE-476), use-after-free (CWE-416), and out-of-bounds access (CWE-787, CWE-125). These are indicative of vulnerabilities primarily affecting kernel-space components of the firmware, which lack robust memory protection mechanisms. Additionally, the relatively high incidence of race conditions (CWE-362) and improper locking (CWE-667) highlights the risks associated with concurrent execution, a common feature in low-level firmware kernels. These results are aligned with the predominant presence of vulnerabilities tied to the Linux kernel.

Only a small subset of vulnerabilities were associated with Dropbear, making not possible to analyse deeper the vulnerabilities tied to the userland and specifically related to network activities. Investigate these aspects is crucial to get a better understanding of the security posture of BAS devices. Overall, the table provides valuable insights into the primary issues affecting devices within the BMS ecosystem, particularly concerning low-level components.

5.1.3 Kernel analysis

This subsection presents a qualitative analysis of the extracted kernels, aiming to understand their distribution across different vendors, the timelines for their updates, and the policies governing their selection.

The dataset consists of 56 firmware images, but kernel-related information could be retrieved for only 31 of them. Figure 5.3 illustrates the software images collected in the archive, organized by year of release. A clear skew toward the year 2024 is observed, most likely due to vendors routinely updating their software products, search engines prioritizing more recently updated pages, and the frequent unavailability of historical versions, as many vendors do not maintain publicly accessible archives of past releases.

To assess the maintenance status of the firmware included in the dataset, the analysis focused on the versions of the kernels used. Extracting and examining kernel versions provides insights into the reliability of the firmware and the security standards adopted by manufacturers in developing these products. It is reasonable to assume that the use of more recent kernels reflects more deliberate and informed choices regarding various components within the software.

Figure 5.4 presents a bar chart showing the Linux kernel versions used in firmware releases, grouped by release year. For clarity, kernel versions are represented by their major and minor numbers, omitting the full version strings. The dataset includes Linux kernel versions ranging from 2.6 to 6.6. Version 2.6.29 was released in 2009, while version 6.6 came out in 2023 [45]. This range highlights the wide variability in software freshness across IoT devices, with differences of up to five major kernel versions. Notably, none of the firmware images released in the past three years have used a kernel older than version 4.4, that was released in 2016.

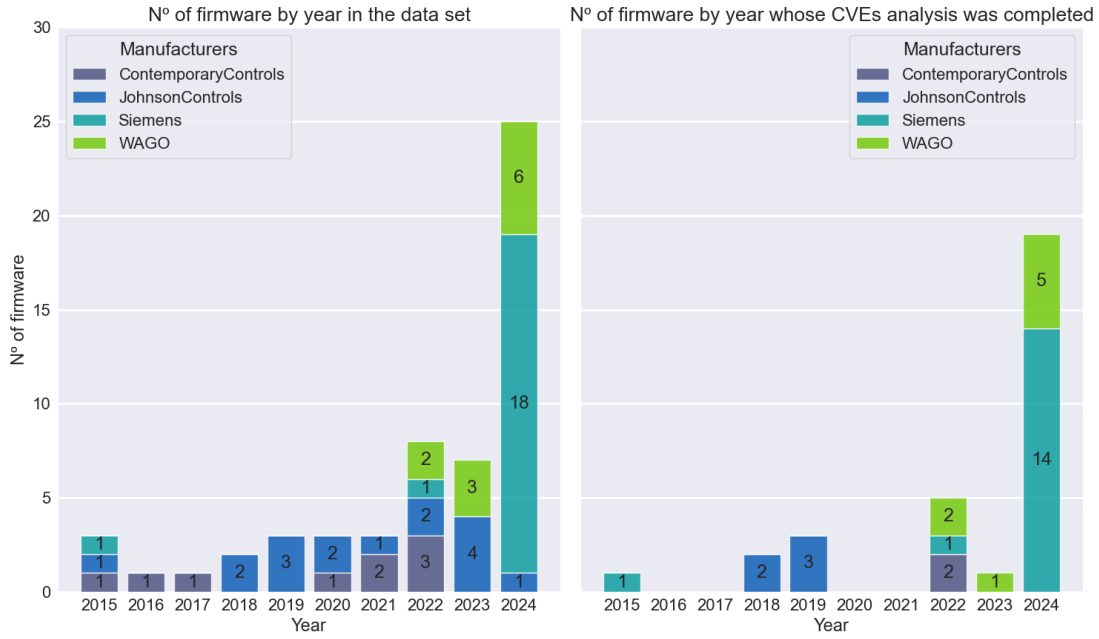


Figure 5.3: Firmware by year in the data set

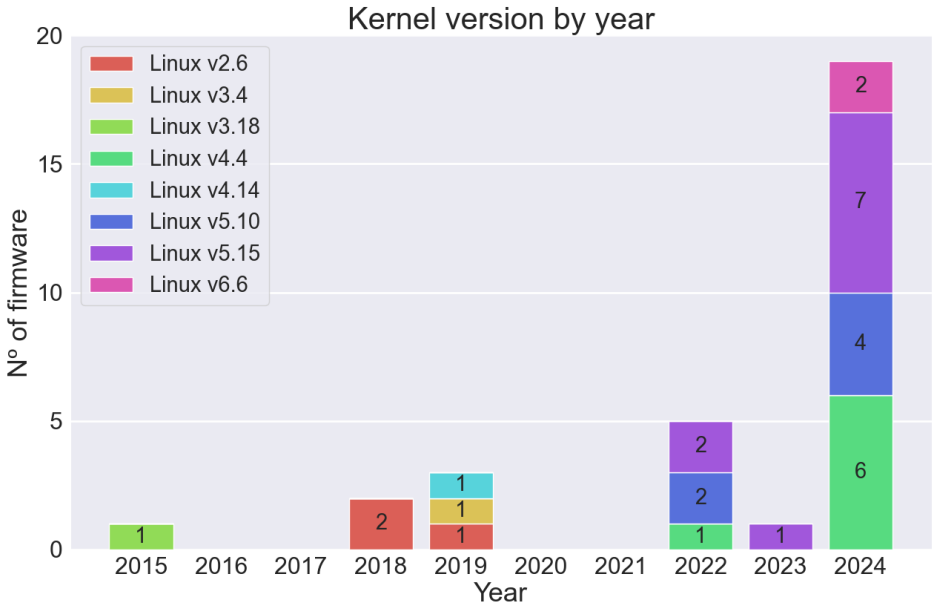


Figure 5.4: Firmware split by kernel version and year

To better understand trends in the delay of kernel integration within firmware, Figure 5.5 illustrates the median delay per year in the integration of kernels into firmware images, relative to their release date. The median was chosen instead of the mean due to the limited amount of data available (making it difficult to conduct robust statistical analyses). Additionally, the median when just limited data are available is less sensitive to outliers. It is worth noting that the mean values do not differ significantly from the median values in this case. The graph indicates that using old kernel releases in not a practice that is increasing over time, which is encouraging regarding the general health status of these devices. In 2024, the year with the most available data, the median delay is 4 years. This is significantly better compared to the 8 year delay observed in 2018. Moreover, aside from the six firmware instances using kernel version 4.4, all other firmware utilize versions 5.10 or newer, released after 2020. The firmware exhibiting the greatest delay between kernel release and implementation into the images belong to the years 2018 and 2019. However, since these years include only five firmware samples, drawing meaningful conclusions from such limited data remains challenging.

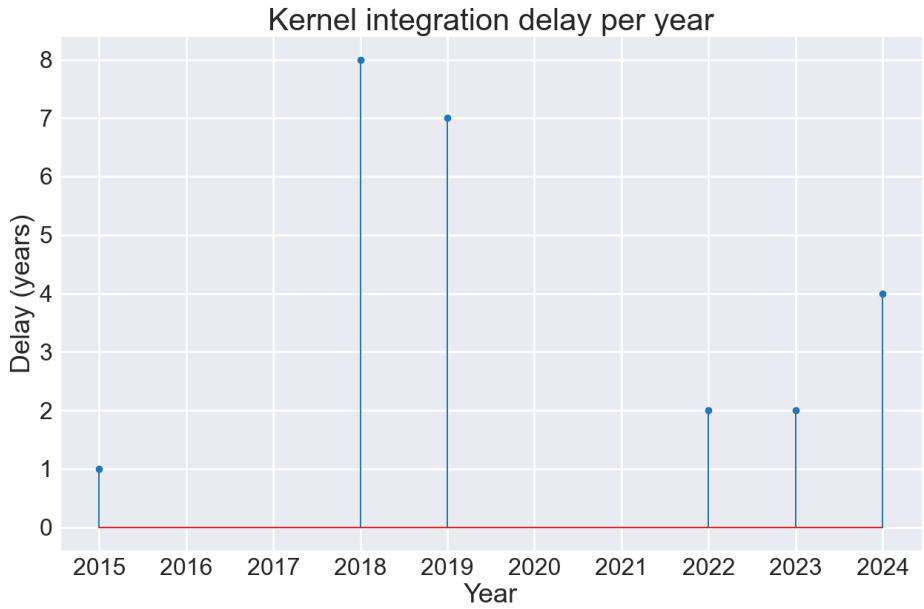


Figure 5.5: Median kernel integration delay

To gain a deeper understanding of the security status of commercially available firmware, it is important to consider not only the delay in kernel release and its integration inside the firmware but also its maintenance over time. Each Linux kernel version has not only a release date but also a defined support duration,

known as its End of Life (EOL). Once a kernel reaches its EOL, it no longer receives updates about: security patches, bug fixes, stability or performance improvements. Therefore, choosing the appropriate kernel type is crucial to ensure adequate ongoing maintenance. Linux kernel can be categorized into several types for maintenance, including mainline kernels, stable kernels, and long-term support (LTS) kernels. Mainline kernels introduce new features, stable kernels receive regular bug fixes, and LTS kernels are maintained for extended periods to ensure ongoing support and security updates.

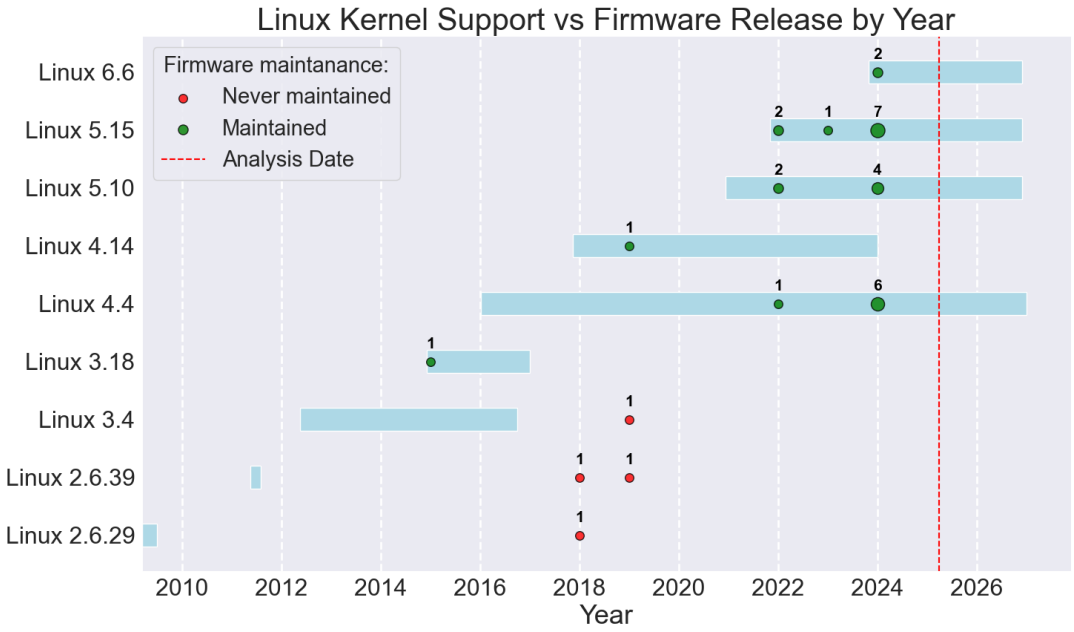


Figure 5.6: Kernel support compared to firmware release.
In this plot is represented in each row a Linux kernel span life, from its release to its end of life The scattered points represent the number of firmware released by year The analysis date line represents when the analysis has been conducted

Figure 5.6 illustrates the various firmware versions, organized by release year (x-axis) and kernel version used (y-axis). Additionally, the figure depicts the lifecycle duration of each kernel version, thereby enabling an evaluation of the appropriateness of kernel selection in each firmware.

The firmware versions using Linux 2.6 kernels released in 2018 and 2019 immediately stand out. Both kernel versions had notably short maintenance periods and were initially released approximately 7 to 9 years prior. Similarly, the firmware employing the Linux 3.4 kernel, despite benefiting from a slightly longer maintenance duration, was already obsolete at the time of the firmware's release. It is

crucial to highlight that selecting a kernel version no longer receiving maintenance poses significant security risks, as the lack of ongoing security updates makes these systems vulnerable to well-known and easily reproducible attacks.

Fortunately, it can also be observed that all other firmware versions were released using kernel versions actively maintained at the time of release. Notably 25 of these employ kernels that will remain supported until early 2027. The remaining two firmware versions, using Linux kernels 3.18 and 4.14, deserve separate consideration. The firmware released in December 2014 by Siemens for the QMX7.E38 device employs Linux kernel version 3.18. This choice is not optimal, as kernel 3.18 was officially maintained only until January 2017, resulting in a support lifespan of just two years. An LTS kernel would have been preferable. Moreover, subsequent firmware updates for this device are not available in the dataset, suggesting the firmware may have never been updated. Conversely, the firmware using Linux kernel version 4.14, released in 2019 by Johnson Controls for FW-8-8V-14 device, appears to represent an appropriate kernel choice due to its extensive support period and LTS. Additionally, the dataset includes a subsequent firmware version released in 2023. Unfortunately, determining whether this newer firmware utilizes a more updated kernel is impossible, as the analysis could not unpack it due to encryption.

It is important to notice that the choice of the Linux kernel in IoT firmware development is strongly influenced by hardware constraints and driver availability. Most IoT boards are tied to a specific BSP (Board Support Package) provided by the SoC vendor, which includes a particular kernel version and custom drivers. These drivers are often tightly coupled with the kernel, making upgrades difficult without extensive modifications. As a result, developers often rely on older LTS kernels that balance stability and long-term security updates. Additionally, compatibility with build systems like Yocto or Buildroot further narrows the kernel options. These constraints often result in devices shipping with outdated kernels, increasing their exposure to known vulnerabilities if not properly maintained.

In summary, the firmware included in the dataset exhibits a generally strong security posture regarding kernel updates. Out of 31 firmware images analysed, 25 employ kernel versions that are still actively maintained, 2 use kernel versions that, although no longer supported, were maintained at the time of firmware release, and only 4 firmware versions were initially released with outdated kernels. Additionally, the kernel integration delay is generally short, with only a few firmware images raising notable concern.

5.1.4 GCC analysis

Assuming firmware update status age from kernel version could be overly simplistic. Hence, another common item in these firmware images, the GNU Compiler

Collection (GCC), was examined. GCC is a key element within the toolchain used to compile kernels and executables contained in firmware, when combined with kernel version data, analysing GCC versions gives a clearer indication of firmware image components.

Figure 5.7 presents the distribution of firmware by year of release and GCC version used. With respect to the previous kernel analysis here two firmware images have been left out, due to the fact that was not possible to determine the GCC version employed. GCC version is identified by looking into strings in compiled objects, based on which the most probable GCC version is guessed. The firmware images that are missing are from vendor Contemporary Controls and were published in 2022. The GCC versions seen in the dataset vary between version 3.4.1, published in 2004, and version 11.4.0, published in 2023 [46]. Obviously, using a version dating back to 2004 is well out of date and has potential security implications.

The use of outdated compiler toolchains, particularly older versions of GCC, presents a significant security concern in firmware development. More modern GCC versions include essential security patches and enhancements for stack canaries, position independent executables (PIE), and control-flow integrity (CFI), which mitigate common exploitation techniques. Older compilers not only lack these defences but may also contain bugs that can lead to faulty or insecure binary generation. In constrained environments, such as the ones for embedded systems, where older toolchains are often retained for compatibility, this can result in firmware that is inherently more vulnerable to memory corruption and code reuse attacks.

It is possible to carry out a comparative analysis of the integrating delays between kernels and GCCs. Figure 5.8 illustrates the delay in GCC integration using median values (an explanation for this choice was provided in subsection 5.1.3). The graph somewhat reflects the pattern seen earlier in Figure 5.5. Especially, a considerably integration delay can be noted for firmware released in 2018 and 2019. As aforementioned, these are critical years and all firmware of these years belongs to Johnson Controls. Conversely, an observable improvemente is noted in the most recent three years with a declining trend. The newer firmware versions belong to the manufacturers WAGO and Siemens.

This analysis emphasize the consistency between the findings derived from kernel and GCC version analysis. Significant issues were identified with the components used in firmware released during 2018 and 2019. That said, they do not indicate a broader or general trend. On the other hand, in recent years, time range when a larger number of samples are available, use of new components appears to have been the norm.

Overall, while security within the realm of BMS still requires improvement, it does not appear to be in a critical state. As demonstrated by the statistical

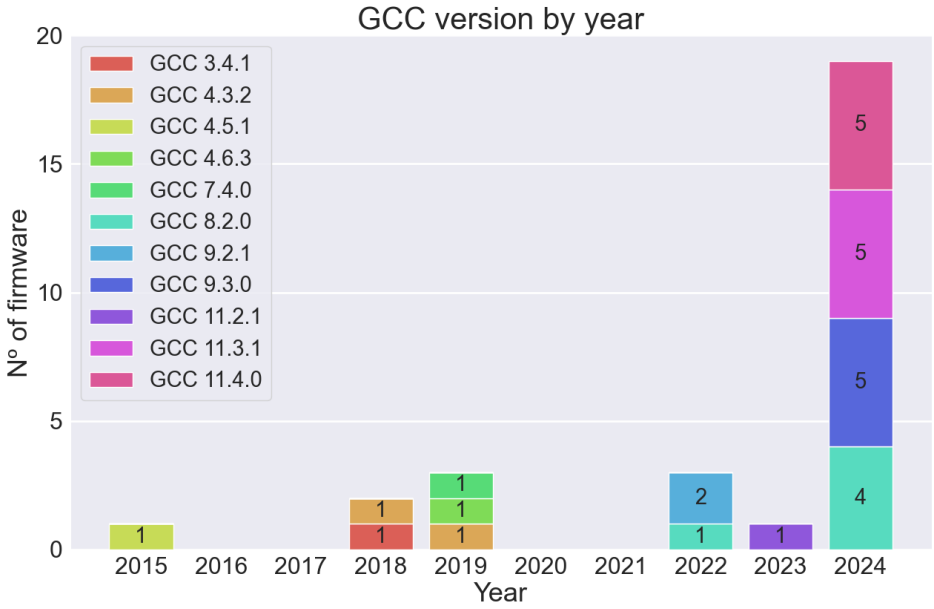


Figure 5.7: Firmware split by GCC version and year

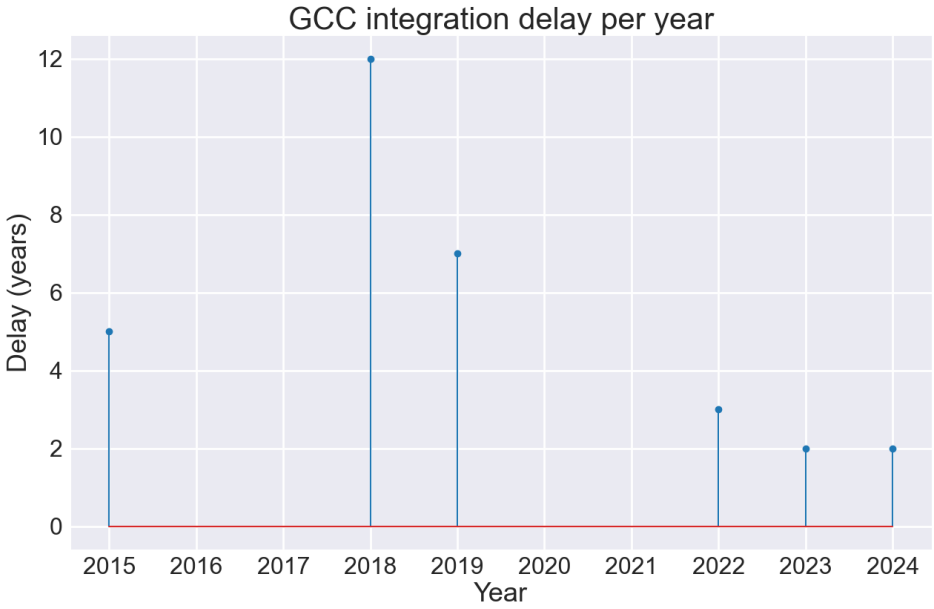


Figure 5.8: Median GCC integration delay

analysis of the CVEs, the majority fall within the medium to high severity range. This indicates that these systems require frequent updates to keep pace with evolving threats that could compromise their security posture. However, the analysis of kernel and GCC component versions suggests that vendors have recently shown greater awareness of security concerns. In particular, there is evidence that more modern versions of kernels and toolchains have been adopted in recent years. Additionally, the preference for LTS kernel versions over those with shorter maintenance windows highlights an increased sensitivity to security best practices. In conclusion, although there remain areas where significant improvements are needed, the findings suggest that meaningful steps have been taken in recent years to enhance the security of the analysed firmware.

5.2 Case Studies

This section presents two in-depth case studies conducted using EMBA to analyse firmwares from the dataset. The goal is to get a better insight in the analysis procedure and identify gaps between the current state and a scalable, in-depth analysis framework. Two fundamental steps to reach this objective are:

- Establishing a working network connection for the device, where its services can be reached
- Gaining terminal access to control the device

Network access is crucial to assess all services exposed by the device on its network interface. A connection enables monitoring of device traffic and could allow fuzzing analysis. While terminal access is essential to carry out internal firmware inspection, including process analysis, service discovery, and log/metrics retrieval.

The testing environment comprised the following components:

- Host device: Windows 11 machine
- Virtual Machine: Ubuntu 22.04 running on either VirutalBox or Windows Subsytem for Linux
- Emulated device: QEMU emulator running on Ubuntu, emulating Johnson Controls firmwmare FW-8-8V-14 V1.0b16i or FG V2.0b52

The tested firmware require the interaction with a proprietary Windows software (provided as a .exe executable) to evaluate specific network services. However, since the QEMU emulation is reachable only via the Ubuntu VM, the Windows host machine and the VM were configured to bridge the communication between the executable and the emulated environment. The following steps were implemented to establish this configuration:

1. Creating a LAN between the Windows host and the Ubuntu VM
   - VirtualBox: network between VM and host is configurable via the Network Manager
   - WSL: by default has a network between the host machine and the virtual environment
2. On the Ubuntu VM
   - Enabling IP forwarding: `sudo sysctl -w net.ipv4.ip_forward=1`
   - Setting NAT postrouting rule: `sudo iptables -t nat -A POSTROUTING -o <TAP_NAME> -j SNAT --to-source <IP_UBUNTU>` to make the Windows traffic appear as sent by the Ubuntu VM. <TAP_NAME>: name of the TAP device created by the `run.sh` script. <IP_UBUNTU>: IP address of the Ubuntu machine used by the network interface enabled to communicate with the emulated firmware.
3. On the Windows host creating a route rule to address the traffic sent to the emulated device towards the Ubuntu VM that acts as a gateway: `route add <destination_network> MASK <subnet_mask> <gateway_ip>`. <destination_network>: IP address of the network shared by the Ubuntu VM and the emulated device <subnet_mask>: subnet mask of the destination network <gateway_ip>: IP address of the Ubuntu VM in the network shared with the Windows host

5.2.1 Report on Johnson Controls FW-8-8V-14 V1.0b16i

Introduction

The device FW-8-8V-14, running firmware version V1.0b16i, made by Johnson Control is characterized by a MIPS architecture and little endian mode. It is the only MIPS based architecture in the data set. The original Linux version used in the firmware is v4.14.105, but for creating an emulation compatible image Linux v4.1.52 is used. The device belongs to the EasyIO FW series, it is a controller that can be used as part of an existing network or as an isolated BMS Wi-Fi network. The main function of this controller is to handle Variable Air Volume applications and sensors. It is designed to manage field devices through its four universal inputs ports and integrated antenna, it is also equipped with a differential pressure transducer. Additionally, the controller is capable of initiating routines, storing data in its internal database and performing analysis. It supports multiple communication protocols, including BACnet IP client/server, BACnet MSTP client/server, Modbus TCP/RTU M/S as well as Sox, TCOM, P2P and MQTT.

These are used to provide services and also to coordinate with other nodes in the network when available. It exposes a web server and a Sox server (Sox is a Sedona framework protocol) through which it can be programmed.

The FW-8-8V-14 V1.0b16i firmware was selected because it is the only device that supports a working telnet connection. However, the TCP services booted by the emulation are not discoverable by Nmap, which means an investigation should take place. Using this firmware as the starting point was considered the optimal approach for investigating emulation, as it provided the necessary connectivity to access the system during runtime.

System overview

To fully understand the emulation process and how the device works, it is crucial to first give an overview of the steps taken to achieve the emulation. EMBA, throughout its analysis, identifies the main parts of the firmware image (e.g. zImage, file system, bootloader), along with the foremost information for creating an emulation environment (e.g. architecture, OS). Once EMBA obtains this information, it attempts to run the emulation environment with different configurations until the firmware boots successfully with a working TCP connection. At the end of this process a folder with all the components required to rerun the emulation with QEMU is created. Appendix C provides the command to run the emulation on QEMU.

The serial connection that can be established via the `-serial telnet:localhost:4321,server,nowait` option using the command `telnet localhost 4321` enables a telnet connection from the VM to the device. Although this behaviour should be available for all the devices starting successfully the emulation in practice it is not. Just this device enables a working telnet connection, all the others open the connection but then the terminal becomes unresponsive. Asking to Michael Messner, a creator and maintainer of EMBA, about this issue he replied that is not the first time someone reported this behaviour as can be seen by two github issues: Issue 459 and Issue 382. Maintainers say that this happens primarily on ARM architectures, so most likely there are issues with the ARM compiled kernel, console or code.

Furthermore, since a network configuration has been set up, the device is also accessible over the network. EMBA configures a TAP interface, which serves as the gateway to reach the emulated device from the VM.

Manual analysis

This phase focuses on understanding which services are running within the system. This involves analysing their behaviour both from a network perspective and an internal perspective, with the goal of identifying any discrepancies between the

two. Following this assessment, troubleshooting can be performed to address any issues that may hinder the proper deployment of the services. Finally, an analysis is conducted to evaluate the state of the services and so the ability to perform operations on them.

The initial test conducted on the device involved scanning its ports with Nmap to assess the response from the services. The scan aimed to identify which ports were active during system boot. The services booted were tracked by EMBA during its analysis along with the ports and protocols they use. It targeted 9 TCP ports and 3 UDP ports. The outcome indicated that the UDP ports were open, while the TCP ports were filtered. The results were not aligned with other scans executed on products of the same vendor, specifically on Johnson Controls FG and FS series, where all the services booted resulted open with no distinction between TCP and UDP.

The manual analysis continued by looking into the network configurations and processes running on the system to gain a deeper understanding of the system's behaviour during runtime. Executing the command `ip addr` revealed a total of 11 interfaces. Among these were taken into consideration: lo (loopback interface), eth0, eth0.1@eth0, eth0.2@eth0 and br-lan. The main interface was eth0 which had two subinterfaces eth0.1 and eth0.2, likely used for handling a VLAN. The bridge interface, br-lan, was configured by EMBA and was assigned an IP address reachable via the TAP interface on the VM. It shared the same MAC address as eth0.1.

By using commands such as `ifconfig` or inspecting the contents of `/proc/net/dev` an overview of the traffic transmitted through the interfaces was obtained. It was noticed that the traffic was mainly transmitted by the two subinterfaces, eth0.1 and eth0.2. The subinterface eth0.1 transmitted and received traffic, likely originating from the network linked to the VM, while the subinterface eth0.2 just transmitted traffic but did not receive any.

Identifying listening services was possible using `netstat -tulp` and `netstat -tuln`. When combined, these commands provide information about which services are listening on specific ports and their respective origins. The output indicated that all services previously detected as filtered were, in fact, actively listening.

The analysis proceeded by verifying whether firewall rules were in place. To accomplish this task the command `iptables -L` was used. The output, which was extensive and complex, was analysed using ChatGPT AI tool. Querying the AI agent about whether the configured rules could have potentially blocked the traffic, it answered negatively. To validate the AI's assessment, the firewall rules were saved using the command `iptables-save > iprules` and then flushed with `iptables -F`. With no rules in place, an additional Nmap scan was performed. The results of this scan, as shown in 5.1, revealed no TCP filtered ports, only open and closed ones. This outcome contradicted the AI's earlier analysis, which stated

that the firewall rules were not blocking the traffic.

Listing 5.1: Nmap result FW-8-8V-14 V1.0b16i

```
Nmap scan report for 192.168.10.30
Host is up (0.0059s latency).

PORT      STATE          SERVICE      VERSION
22/tcp    open           ssh          Dropbear sshd (protocol 2.0)
53/tcp    open           domain       Cloudflare public DNS
80/tcp    open           http         nginx 1.17.5
139/tcp   open           netbios-ssn  Samba smbd 3.X - 4.X (
workgroup: WORKGROUP)
443/tcp   open           ssl/http     nginx 1.17.5
445/tcp   open           netbios-ssn  Samba smbd 3.X - 4.X (
workgroup: WORKGROUP)
2812/tcp  closed         atmtcp
3333/tcp  closed         dec-notes
53/udp    open           domain       Cloudflare public DNS
137/udp   open           netbios-ns   Microsoft Windows netbios-ns (
workgroup: WORKGROUP)
138/udp   open|filtered  netbios-dgm
MAC Address: 38:D1:35:02:00:00 (EasyIO Sdn. Bhd.)
```

With the services reachable was possible to perform further analysis useful for assessing their state. For each device sold from Johnson Controls there are multiple guides, aimed at helping the final user getting a better understanding of their products. Particularly useful are the FAQ guides developed for all the EasyIO series. In these documents it is possible to find interesting insights ranging from the network configurations and login credentials to protocol limitations. Using the credential extracted from the FAQ document it was possible to test the SSH service. It was responsive and allowed the login with the command `ssh -oHostKeyAlgorithms=+ssh-rsa sdcard@192.168.10.30` and using the password *123456*. Other credentials were also tried but they did not provide access much less root privileges.

The Web services were checked as well. From the Firefox web browser the connection to the URL `http://192.168.10.30` was redirected the to `http://192.168.10.30/sdcard/cpt/app/signin.php`. The page displayed correctly a login form asking for credential to access the CPT Tools via web. The attempt to login into the services with the credential listed in the FAQ was unsuccessful, it also failed the attempt of trying credentials of different EasyIO series. Furthermore it was possible testing, setting up a proper network configuration, the Sedona sox protocol. CPT Tools is an open source software programming tool that provides third party configuration and management tools for products that run in a Sedona environment, such as the FG series, the FC series, the FW series and the 30P. Testing it required establishing a working network communication between the

host machine and the emulated device as explained at the beginning of the section.

With this configuration deployed the communication between the Windows host and the emulated device was working correctly as confirmed by the web services reachability. However the connection by mean of CPT Tools was unsuccessful, most likely because of the incorrect credentials listed in the FAQ document.

System insights

Further analysis revealed that the system operates on top of OpenWrt, a Linux operating system targeting embedded devices. Given this information, a better understanding of the various configurations of the system was achieved. OpenWrt configurations can be found at `/etc/config`. The network configurations show the creation of 3 interfaces. The loopback interface, the "lan" interface and the "wan" interface. The "lan" interface is associated with the ifname eth0.1 while the wan interface is associated with the ifname eth0.2. These settings explain why traffic was tracked on the eth0.1 subinterface, probably booted services were trying to communicate with other devices over the LAN area. While the outbound traffic recorded on subinterface eth0.2 was likely generated by the DNS services. A more accurate analysis of the traffic transmitted by eth0.2 could not be conducted because the interface was not connected to a network and the commands available inside the firmware were not suitable.

OpenWrt's firewall management is mainly configured through the file `/etc/config/firewall`. Reviewing this file it was observed that no rules were set for the LAN zone, allowing both outgoing and ingoing communications. Instead, roughly all services were blocked for the WAN zone. Given this, the necessity of flushing the iptables rules to establish a connection was unclear, as no rules were defined for the LAN zone. A plausible explanation is that some rules might be set elsewhere, outside of this configuration file.

5.2.2 Report on Johnson Controls FG V2.0b52

Introduction

The device FG, running firmware version V2.0b52, made by Johnson Control is characterized by an ARM architecture and operates in little endian mode. The original Linux version used by the firmware is v2.6.39.4a, but during the emulation Linux v4.1.52 is used. The device is part of the FG+ series of EasyIO, it is a plant controller, meter and data aggregator. It supports multiple, concurrent protocols including BACnet, Modbus, TCOM, and web services. It is equipped with 16 input ports able to communicate with sensors of different types, from which can collect data inside is internal SQL database. It also has hardware to perform actions such as starting routines and aggregate data.

Manual analysis

The device FG, which is based on ARM architecture, was inaccessible via telnet communication. Specifically, upon establishing a connection, the terminal became unresponsive. This issue hindered any further analysis, since there was not a way to access the running firmware.

To establish network-based connections and initialise some custom services EMBA infers few essential scripts (`preInit.sh`, `network.sh`, and `run_service.sh`) in the firmware's filesystem. It is possible to trigger the initialization of Netcat and Telnet daemons by passing the parameter **EMBA_NC=true** during the kernel boot as shown in C.2.

Testing this mechanism by explicitly setting **EMBA_NC=true** in the kernel parameters failed. The expected daemons listening on ports 9876 and 9877 at the IP address 192.168.10.11 were unreachable.

Further validation on FW-8-8V-14 showed no active daemons via `ps` and `netstat -tulp`, though manual daemon initiation succeeded. Attempts to capture script output by redirecting stdout and stderr resulted in unpredictable system behavior, and live-system edits of `run_service.sh` caused file corruption, prompting abandonment of this debugging method.

Although no ways to get access to the system were found it was still possible to assess the network services. A Nmap scan identified four services: Pure-FTPd on port 21 listening for FTP connections over TCP, Dropbear on port 22 listening for SSH connections over TCP, and Embedthis webapp on ports 80 and 443 listening for HTTP and HTTPS connections. In contrast to the FW-8-8V-14 firmware, this firmware seemed to have no firewall rules set in place. To better understand the network configuration within the device, an examination was conducted regarding the settings of the interfaces on the VM. This analysis revealed a difference compared to the previously examined device. Specifically, instead of creating a TAP interface followed by the creation of a subinterface, only a single TAP interface was instantiated. This suggested a potential difference in network configuration between the devices, where the absence of subinterfaces might implicate the absence of VLANs.

The device's manual was consulted revealing a list of services, along with their respective credentials. Among these were found credentials for Sedona service, file transfer via FTP, CPT graphics deployment via web browser. Nmap was able to detect the services showed in 5.2.

Listing 5.2: Nmap result FG V2.0b52

```
Nmap scan report for 192.168.10.11
Host is up (0.0017s latency).

PORT      STATE          SERVICE      VERSION
21/tcp    open           ftp          Pure-FTPd
22/tcp    open           ssh          Dropbear sshd 2015.67 (
protocol 2.0)
80/tcp    open           http
443/tcp   open           ssl/https?
3333/tcp  closed         dec-notes
1001/udp  open|filtered  unknown
1876/udp  open           ewcappsrv?
5021/udp  open|filtered  zenginkyo-2
```

The Web services were accessible through a web browser. The page `http://192.168.10.11/fg_utility_app/app/signin.php` displayed a login form for accessing the CPT tools. This time, the credentials listed in the manual were successful, allowing to sign in. Afterwards the CPT tools were accessed via the sox protocols. The application, running on the host Windows machine, displayed a working interface. Through the app, it was possible to check the firmware status and view the available services. Additionally, the available Sedona kits could be reviewed, with the option to install new ones.

The next step involved testing the SSH connection. Although this service was not listed in the device's manual, the Nmap scan highlighted its presence. Upon establishing the connection, the SSH daemon prompted for a password. Surprisingly, two pairs of credentials, intended for different services, were accepted during login: using the commands `ssh sdcard@192.168.10.11` or `ssh webuser@192.168.10.11` and entering the password *123456*. This allowed access to the system with the two accounts, sdcard and webuser.

Eventually, by exploiting the SSH service it was possible to access the live system but not as root user. Examination of the file system confirmed that the network setup differed from that of FW-8-8V-14. Moreover, the FG firmware was found to be not based on OpenWrt.

In appendix C is possible to see the procedure followed to recover a password and take control of the system as admin.

In this case study presented, initially, it was only possible to access the exposed network services without direct terminal access. Through a deeper investigation of EMBA, device documentation, and the device itself, it became possible to achieve terminal access. Unlike the previously analysed device, this time control was obtained via network using the SSH protocol.

5.2.3 Case studies summary

The case studies presented in this section are useful for understanding the current state of EMBA's emulation capabilities. As observed, this valuable tool is capable of performing an in-depth analysis of a firmware image, from which it can extract a wealth of highly useful data. Furthermore, it can consolidate this information into various reports and leverage it to attempt the reconstruction of an emulated environment.

In many instances, as illustrated in Section 5.1, EMBA is not able to automatically carry out the emulation process. Nevertheless, as demonstrated in the two preceding examples, there are cases in which successful automatic emulation is indeed achieved. This represents a significant advancement in terms of automation and the potential for large-scale firmware analysis.

Unfortunately, even when emulation is successful, the firmware is often not immediately ready for direct use. As previously outlined, two criteria have been defined to determine whether a firmware is suitable for deeper analysis: the firmware must be capable of network communication and service exposure, and it must allow terminal access to the running system.

The case studies also highlight the manual efforts required to prepare firmware for analysis, enabling the setup of an environment suitable for conducting more advanced investigations.

The outcome of this experience serves as a representative example of the current scalability level of this technique.

Chapter 6

Discussion

This chapter aims at gathering the results and considerations made in Chapter 5 to grasp the implications of them and outline the state of health in the realm of BAS IoT devices. Moreover it will highlight the spots where this method shows bottlenecks and further work should be done to achieve better performance and a deeper analysis.

6.1 Implications

The implications of this work combine some of the results presented in Chapter 5 with a critical discussion aimed at contextualizing them. This serves to investigate the current state of security of IoT devices within the domain of SB.

Vulnerabilities and Attacks

The statistical analysis conducted on the identified vulnerabilities aims to highlight one of the most critical aspects of firmware: their security status. In the research that was carried out, vulnerabilities were found to be almost exclusively related to the kernel. This provides insight into the low-level types of weaknesses that affect these devices, as well as the potential severity of their impact on the overall system. However, this focus excludes an important portion of the software stack, particularly the network services exposed by the firmware.

The presence of a high number of vulnerabilities classified as medium and high severity raises significant concerns about the potential attacks that these firmware images may be subjected to. This observation should be considered alongside another noteworthy metric presented in the appendix, in Table B.1. The final column of the table lists the number of publicly available exploits associated with each firmware. These exploits are ready-to-use modules designed for Metasploit, which is commonly used to automate attacks. As a result, not only are these

vulnerabilities present, but practical attack methods are also readily available; even to individuals with limited technical expertise. On average, there are 47 exploit modules per firmware.

Software components and Updates

The analysis conducted on the kernels and compilers versions within firmware aims to highlight the level of awareness vendors apply when selecting system components. The results are not discouraging; they indicate a trend in recent years toward releasing firmware that incorporates up-to-date software. The same cannot be said, however, for certain firmware versions released prior to 2020, which were found to use kernels and compilers already outdated at the time of release.

Choosing secure components for devices that are expected to operate over long periods of time is a critical consideration. Once smart buildings are constructed, the devices within them are typically expected to remain in use for as long as possible. IoT devices, once purchased, may have a lifespan of several decades and firmware running on them are generally released after being tested with that specific hardware. The introduction of new low-level software can destabilize the system, therefore selecting a kernel version with long-term support ensures that the device can receive updates without jeopardizing system stability.

Closely tied to this issue is the problem of firmware updates. Every day, new vulnerabilities are discovered worldwide, and, depending on responsiveness, patches are released by vendors or maintainers. However, the release of a patch alone is insufficient. The responsibility for applying these updates lies with those managing the infrastructure. As shown in studies such as [3], search engines like Shodan and Censys reveal a vast number of internet-connected devices that remain vulnerable to well-known attacks.

This raises important questions about how firmware updates should be managed. An internet-connected smart building would enable automatic software updates for its devices, which could significantly reduce the burden on system administrators while improving security posture through timely patching. However, such connectivity also exposes the building to potential remote attacks. An alternative approach would be to avoid connecting smart devices to the internet altogether, thereby reducing the attack surface for remote exploits that could compromise critical building functions. Nonetheless, this strategy does not inherently guarantee security unless accompanied by regular installation of security updates. Moreover, this choice would then require on-site technical intervention and potentially delayed deployment.

6.2 Limitations of our work

Data collection and Research

The creation of the dataset is undoubtedly a valuable contribution to research in the field of IoT devices used in smart buildings. However, the current sample of 50 firmware images is insufficient for conducting broader and more robust statistical analyses. A larger dataset would be necessary to obtain a comprehensive overview of the state of security in this domain.

Unfortunately, acquiring additional firmware is not merely a matter of investing more research hours. As highlighted in Chapter 3, resources in this area are scarce, and the research is quite limited. This challenge is further hindered by the growing trend of encrypting firmware, as seen in recent releases from Johnson Controls [29].

Therefore, both the availability of analysable material and the overall state of research in this area are quite discouraging. As emphasized in previous works [13, 6], collaboration among all stakeholders involved in this field is essential to raise security standards.

Methodology scalability

The methodology presented in Chapter 4 aims to standardize the analyses performed on firmware, enabling systematic processing and subsequent testing. Achieving automation of these steps represents a significant milestone, as it allows for the analysis of a large number of firmware images and the extraction of meaningful insights without the need for manual inspection of each one. This objective is partially realized through the use of the EMBA tool.

However, while EMBA has proven to be useful and essential to the research conducted, it cannot be considered a complete solution. Written entirely in Bash, it suffers from several limitations related to customization, readability, debugging, and computational efficiency. Although Bash is a powerful tool for automating tasks involving Linux utilities and file system interactions, it is best suited for short scripts rather than large-scale applications involving thousands of lines of code. Its syntax is prone to errors and inconsistencies, particularly in the handling of conditional statements and other control structures, making it a language that is difficult to maintain. These shortcomings also affect debugging and customization processes when adjustments are required.

Additionally, it is important to note that EMBA relies on two frameworks, FIRMADYNE and FirmwAE, which suffer from lack of maintenance. This trend may eventually affect EMBA itself in the coming years.

Kernel study and vulnerabilities

Chapter 5 presents the results of the conducted research, which include a statistical analysis of the vulnerabilities primarily found in the kernel of the investigated devices. While this analysis provides valuable insights, it focuses exclusively on the kernel, thus be limiting when aiming for a more comprehensive assessment.

The kernel is undoubtedly a core component in the execution of firmware, and analysing it allows for the identification of some of the most critical vulnerabilities. However, extending the analysis to include a deeper inspection of the applications installed within the firmware would broaden the scope of the investigation. This expanded focus could help uncover vulnerabilities present in these additional software modules, thus offering a more complete understanding of the overall security posture.

Emulation shortcomings

One of the objectives of this thesis is to achieve emulation of the collected firmware images. Among the 50 unencrypted firmware samples in the dataset, full emulation was successfully achieved for 15 of them, with partial emulation reached for another 15. This results in a total emulation rate of 60%. While this is certainly an encouraging outcome, further work is needed to improve this metric.

A key limitation of this approach lies in the method of emulation itself. Although the substantial efforts behind FIRMADYNE, FirmAE, and EMBA greatly support automation in the emulation process, there are still several areas that require improvements. In particular, the unpacking phase and component recognition need to be revisited and refined in order to increase the proportion of firmware images that can be successfully emulated. Furthermore, considerable work remains in configuring the emulation environment, specifically in the booting process and in supporting a broader range of hardware configurations.

It is also essential to address the issues encountered once an emulation is deemed complete. As highlighted in the case studies presented in Section 5.2, achieving full emulation does not necessarily enable all types of analysis. In some cases, even gaining terminal access to the emulated device can be non-trivial.

Additionally, a structural challenge associated with devices used in BAS is their reliance on communication protocols that do not operate over Ethernet or IP. This poses a significant obstacle, as emulating non-standard networking interfaces would require extensive hardware modelling efforts. Nevertheless, it is encouraging to observe a growing trend toward the adoption of IP-oriented protocols, suggesting that this model is likely to dominate in the future of IB.

6.3 Future Work

Fuzzing

In this work, a fuzzer was not deployed to assess the various firmware images. However, fuzzing represents a crucial step in identifying vulnerabilities that may be hidden within the service-providing daemons. As demonstrated in previous studies [24, 25, 26], fuzzing can uncover a wide range of potential security issues. Nevertheless, setting up a comprehensive fuzzing environment is a non-trivial task. As shown in the aforementioned works, effective fuzzing requires more than simply establishing communication with the service daemon; it also depends on selecting appropriate input seeds and implementing mechanisms to monitor code coverage.

Although the current emulated systems achieve a reasonable level of stability, they sometimes could generate unexpected errors. The lack of accessibility of some systems complicates the instrumentation of services, making it difficult to assess how much of the code has been covered during testing. Despite these challenges, fuzzing remains a highly promising technique that should be further explored.

One potential approach is to perform code coverage analysis even when full system access is not available, evaluating test results to infer coverage indirectly. Moreover, fuzzing could also contribute to improve the throughput efficiency. As noted in [23, 26], physical devices such as PLCs typically scan for inputs at fixed intervals. This behaviour imposes strict timing constraints that can slow down fuzzing, requiring synchronization to not overflow of requests the device. Emulation, on the other hand, may help overcome this limitation by enabling higher input rates, thus facilitating faster and more effective fuzz testing.

Emulation

An open research topic in this domain is firmware emulation itself. Significant work remains to be done in supporting a wider range of hardware devices. As discussed in Section 2, emerging approaches seek to emulate or rehost firmware by leveraging high-level abstractions of hardware interfaces. These methods may offer improved portability and scalability over traditional low-level emulation techniques.

Nevertheless, building upon existing frameworks and platforms could also benefit the ongoing research. A promising direction would involve rethinking tools such as EMBA at a structural level. This could begin with the design and development of a new tool, implemented in a more suitable programming language (e.g., Rust) to improve maintainability and performance. While EMBA represents a valuable tool, there is considerable potential to improve its architecture and expand its capabilities. A new implementation could still rely on proven tools like Binwalk, Radare2, and QEMU for core analysis tasks, while re-implementing and integrating selected EMBA components in a more modern and efficient manner.

In addition, enhancements could be made directly to EMBA itself. These may include extending support to additional processor architectures and improving support for those already implemented. As observed in the case studies, for instance, ARM-based firmware exhibited issues with Telnet terminal. Furthermore, this research could benefit of improvements to EMBA's network environment, augmenting the number of complete emulations.

Verification and Exploitability of Identified Vulnerabilities

An important extension of the work carried out could involve verifying the identified vulnerabilities. In this study, vulnerabilities were discovered using the EMBA tool and subsequently aggregated into the results discussed. However, no further verification was performed to assess whether these vulnerabilities are exploitable, and if so, how an attacker might leverage them. This type of verification could be guided by existing exploits identified during the analysis phase. These are readily available exploit modules, such as those provided through the Metasploit framework, as shown in Table B.1. Starting from such existing attack modules, one could evaluate whether the firmware is vulnerable or whether the exploitation chain is complex and not easily actionable. Moreover, a deeper investigation into the vulnerabilities for which no exploits currently exist would allow for a more comprehensive assessment of the weaknesses present across the various firmware samples.

Chapter 7

Conclusions

This thesis presented a comprehensive investigation into the security posture of IoT devices used in SBs systems. The study addressed a significant gap in the current literature by constructing a dataset, from scratch, containing 56 firmware images collected from four different manufacturers. These firmware samples form a basis for subsequent analysis.

From the outset, the research aimed to explore the vulnerabilities inherent in smart building infrastructures, emphasizing how increased connectivity and the reliance on legacy devices can introduce significant security risks. The thesis demonstrated the feasibility of analysing embedded firmware at scale, using a carefully crafted methodology that included firmware acquisition, extraction, static and dynamic analysis, and full-system emulation. By developing a pipeline for firmware processing and leveraging tools such as EMBA, the analysis was efficiently extended across the entire dataset.

A key outcome of this study was the successful emulation of 30 firmware samples, with 15 achieving partial emulation and the remaining 15 reaching full emulation. This enabled active probing of exposed services and further inspection of system behaviours. Static analysis revealed a concerning average of 1,570 CVEs per firmware image, indicating widespread vulnerabilities within the ecosystem. Additional insights were gained by analysing kernel and GCC toolchain versions. Although some firmware incorporated kernels released as early as 2009 and compilers dating back to 2004, highlighting significant delays in the adoption of security updates, the majority of firmware released between 2022 and 2024 relied on substantially more recent components.

The research also included two detailed case studies, which served to validate the methodology and investigate in-depth two devices. These studies aimed at evaluate the emulation outcomes by diagnosing issues and highlighting both misconfigurations and security oversights. It was observed that the emulation process may require manual intervention, such as modifying firmware rules to enable network

services and recovering passwords to gain access to the devices.

Overall, the study contributes valuable resources and empirical data to the field of smart building security. It emphasizes the urgent need for more rigorous practices in firmware development, timely patching, and transparent documentation from vendors. Furthermore, the thesis underscores the value of emulation-based testing as a practical approach to exposing latent vulnerabilities in embedded systems. Future research can build on these results by expanding the dataset, refining emulation fidelity, and applying dynamic testing techniques such as fuzzing to further uncover vulnerabilities in BAS environments.

In conclusion, the findings of this thesis not only shed light on the current security state of smart building devices but also offer a foundation for the advancement of secure testing in IoT field. As Smart Buildings continue to proliferate in critical infrastructure, such research is essential to ensuring the safety, reliability, and resilience of the environments they control.

Appendix A

Dataset Binwalk commands

Listings A.1, A.2, and A.3 display the results of Binwalk analysis performed on different firmware images. The analysis was conducted using the command `binwalk <firmware_name>`, which identifies known file signatures within the binary data blob. In Listing A.1, only a subset of the output is presented, as the whole result was too long.

Listing A.1: Binwalk analysis of CC100 v4.6.3

```
1  DECIMAL       HEXADECIMAL     DESCRIPTION
2
3  65536         0x10000         Flattened device tree, size: 19491
   bytes, version: 17
4  ...
5  445228        0x6CB2C         LZO compressed data
6  ...
7  1895326       0x1CEB9E        Executable script, shebang: "/bin/
   sh"
8  ...
9  141523175     0x86F78E7       mcrypt 2.2 encrypted data,
   algorithm: blowfish-448, mode: CBC, keymode: 8bit
10 143435144     0x88CA588       SHA256 hash constants, little
   endian
11 143901696     0x893C400       ELF, 32-bit LSB executable, ARM,
   version 1 (SYSV)
12 ...
13 150141788     0x8F2FB5C       Zip archive data, encrypted
   compressed size: 123469, uncompressed size: 94953810, name:
14 152481792     0x916B000       Linux EXT filesystem, blocks count:
    115200, image size: 117964800, rev 1.0, ext4 filesystem data,
   UUID=5de99579-fe67-4129-a007-f2358df28df2
15 ...
16 270924799     0x1025FBFF      eCos RTOS string reference: "ecos"
```

```
271343107     0x102C5E03     mcrypt 2.2 encrypted data, algorithm: 3DES, mode: ECB, keymode: 4bit
...
274864453     0x10621945     Certificate in DER format (x509 v3), header length: 4, sequence length: 1288
274988659     0x1063FE73     OpenSSH RSA1 private key, version "—END OPENSSH PRIVATE KEY———"
...
275007876     0x10644984     Unix path: /usr/sbin/ssh-askpass
275008423     0x10644BA7     PARity archive data - file number 19534
```

Listing A.2: Binwalk analysis of FS V3.0b62

```
DECIMAL       HEXADECIMAL     DESCRIPTION

0             0x0             OpenSSL encryption, salted, salt: 0xBCE06841CB086723
42825612      0x28D778C       VMware4 disk image
```

Listing A.3: Binwalk analysis of QMX7.E38 V01.16.53.44

```
DECIMAL       HEXADECIMAL     DESCRIPTION

0             0x0             Zip archive data, at least v2.0 to extract, compressed size: 1708, uncompressed size: 9243, name: fmwz.xsd
1774          0x6EE           Zip archive data, at least v2.0 to extract, compressed size: 655, uncompressed size: 2381, name: init.xml
2495          0x9BF           Zip archive data, at least v2.0 to extract, compressed size: 33889, uncompressed size: 60836, name: MLO
36445         0x8E5D          Zip archive data, at least v2.0 to extract, compressed size: 394772, uncompressed size: 2158592, name: qmx7.e38-release image-01.16.53.44-rut.configuration.ubifs
431333        0x694E5         Zip archive data, at least v2.0 to extract, compressed size: 66325692, uncompressed size: 83804160, name: qmx7.e38-release-image-01.16.53.44-rut.ubifs
66757127      0x3FAA207       Zip archive data, at least v2.0 to extract, compressed size: 727004, uncompressed size: 3578350, name: Readme3rdPartySoftware.txt
67484215      0x405BA37       Zip archive data, at least v2.0 to extract, compressed size: 1430, uncompressed size: 8192, name: u-boot-rut.env
67485717      0x405C015       Zip archive data, at least v2.0 to extract, compressed size: 281684, uncompressed size: 585212, name: u-boot-rut.img
```

```
11 67767473      0x40A0CB1      Zip archive data, at least v2.0 to
   extract, compressed size: 3294317, uncompressed size: 3328116,
   name: uImage-rut.bin
12 71062681      0x43C5499      End of Zip archive, footer length:
   22
```

Appendix B

Data summary

Table B.1 presents a summary of the firmware components alongside the results of the vulnerability analysis. The column titled "Number of Exploits" refers to ready-to-use exploits available through Metasploit, a widely used framework designed to automate the execution of known attacks.

Device	Firmware	Kernel	GCC	N° CVE	N° CVE high	N° CVE medium	N° CVE low	N° exploits
PXG3	V01.21.152.8-3236	Linux v4.4.302	GCC 8.2.0	1677	437	1128	112	60
PXC3	V01.21.194.18-6633	Linux v4.4.302	GCC 8.2.0	1677	437	1128	112	60
PXC3	V01.21.172.22-6095	Linux v4.4.302	GCC 8.2.0	1677	437	1128	112	60
PXG3.WX00-2	V02.21.194.25-22980	Linux v5.15.71	GCC 11.4.0	1325	448	863	14	23
PXG3.WX00-2	V02.20.172.47-21561	Linux v5.10.35	GCC 9.3.0	1571	546	1000	25	38
QMX7.E38	V01.16.53.44	Linux v3.18.10	GCC 4.5.1	1952	615	1208	129	88
DXR2	V01.21.194.18-6633	Linux v4.4.302	GCC 8.2.0	1677	437	1128	112	60
DXR2	V01.21.172.22-6095	Linux v4.4.302	GCC 8.2.0	1677	437	1128	112	60
PXC4	V02.21.194.25-22980	Linux v4.4.302	GCC 11.4.0	1677	437	1128	112	60
PXC4	V02.20.172.47-21561	Linux v4.4.302	GCC 9.3.0	1677	437	1128	112	60
PXC5.E003	V02.21.194.25-22980	Linux v5.15.71	GCC 11.4.0	1325	448	863	14	23
PXC5.E003	V02.20.172.47-21561	Linux v5.10.35	GCC 9.3.0	1571	546	1000	25	38
PXC5.E24	V02.21.194.25-22980	Linux v5.15.71	GCC 11.4.0	1325	448	863	14	23
PXC5.E24	V02.20.172.47-21561	Linux v5.10.35	GCC 9.3.0	1571	546	1000	25	38
PXC7	V02.21.194.25-22980	Linux v5.15.71	GCC 11.4.0	1325	448	863	14	23
PXC7	V02.20.172.47-21561	Linux v5.10.35	GCC 9.3.0	1571	546	1000	25	38
FG	V2.0b52	Linux v2.6.39.4	GCC 4.3.2	2134	540	1395	199	105
FG	V2.0b51	Linux v2.6.39.4	GCC 4.3.2	2134	540	1395	199	105
FG	V1.5b51	Linux v2.6.29.2	GCC 3.4.1	2348	600	1487	261	143
FS	V3.0b51d	Linux v3.4.39	GCC 4.6.3	2118	572	1355	191	103
FW-8-8V-14	V1.0b16i	Linux v4.14.105	GCC 7.4.0	1702	548	1090	64	46
BASpi-IO6U6R	V1.0.33	Linux v5.10.63	-	1535	527	985	23	35
BASpi-IO6U4R2A	V1.0.33	Linux v5.10.63	-	1535	527	985	23	35
PFC	V4.6.1	Linux v5.15.107	GCC 11.3.1	1185	372	799	14	18
PFC	V4.1.10	Linux v5.15.19	GCC 9.2.1	1501	540	944	17	30
CC100	V4.6.3	Linux v5.15.107	GCC 11.3.1	1185	372	799	14	18
CC100	V4.1.10	Linux v5.15.19	GCC 9.2.1	1502	540	945	17	31
PFC300	V4.6.1	Linux v6.6.15	GCC 11.3.1	1036	291	736	9	7
WP400	V4.6.3	Linux v6.6.3	GCC 11.3.1	1085	323	753	9	7
TP600	V4.6.1	Linux v5.15.107	GCC 11.3.1	1185	372	799	14	18
TP600	V4.2.13	Linux v5.15.86	GCC 11.2.1	1232	400	818	14	23

Table B.1: Vulnerabilities summary in the analysed firmware

Table B.2 presents the services detected for the firmware images for which emulation was successfully completed. The column titled "Startup services" lists the services identified by EMBA during the firmware startup process. It is worth noting that EMBA detects only a subset of the services during startup, primarily for debugging purposes; therefore, this list may be incomplete. The "Nmap services"

column displays the services detected by Nmap after the firmware was deployed and brought online by EMBA. This list may also be incomplete, particularly if certain services were configured to run on non-standard ports that EMBA was unable to trace.

Device	Firmware	Startup services	Nmap services
PXG3	V01.21.152.8-3236	nginx	nginx
PXC3	V01.21.194.18-6633	ba-device nginx	nginx
PXC3	V01.21.172.22-6095	ba-device nginx	nginx
PXG3.WX00-1	V02.20.172.47-21561	ba-device nginx	-
QMX7.E38	V01.16.53.44	dhcpcd lighttpd	lighttpd
DXR2	V01.21.194.18-6633	ba-device nginx	nginx
DXR2	V01.21.172.22-6095	ba-device nginx	nginx
PXM	V02.20.172.47-21561	watchdogd	-
FG	V2.0b52	inetd svm.exe tls_tunnel pure-ftpd	Dropbear sshd 2015.67 (protocol 2.0) Pure-FTPd
FG	V2.0b51	inetd svm.exe tls_tunnel pure-ftpd	Dropbear sshd 2015.67 (protocol 2.0) Pure-FTPd
FG	V1.5b51	inetd svm.exe pure-ftpd	
FS	V3.0b51d	redis-server nginx nmbd tls_tunnel TComSQL pure-ftpd smbd monit mosquitto	nginx 1.10.1 Pure-FTPd
FW-8-8V-14	V1.0b16i	nginx tls_tunnel dropbear smbd dnsmasq monit nmbd	Cloudflare public DNS Microsoft Windows netbios-ns
PFC	V4.6.1	lighttpd	-
PFC	V4.1.10	lighttpd	-
CC100	V4.6.3	lighttpd syslog-ng dropbear inetd CMHooksTask	Dropbear sshd 2022.83 (protocol 2.0) lighttpd
CC100	V4.1.10	lighttpd syslog-ng dropbear inetd CMHooksTask	Dropbear sshd 2022.82 (protocol 2.0) lighttpd
TP600	V4.6.1	lighttpd syslog-ng dropbear inetd CMHooksTask	Dropbear sshd 2022.83 (protocol 2.0) lighttpd
TP600	V4.2.13	lighttpd syslog-ng dropbear inetd CMHooksTask	Dropbear sshd 2022.82 (protocol 2.0) lighttpd

Table B.2: Services detected in the emulated firmware, during the startup phase and by Nmap in a network scan

Appendix C

EMBA commands

The following Bash command C.1 is taken from the script *run.sh* located inside the emulation folder, and shows the QEMU command used to start the emulation. An explanation of the options used is provided in the following list:

- `-m`: sets guest startup RAM size to megs megabytes, in this case 2 GB
- `-M`: selects the emulated machine by its name
- `-drive`: defines a new drive
  - `-if`: defines on which type on interface the drive is connected
  - `-format`: specifies which disk format will be used rather than detecting the format
  - `-file`: defines which disk image to use with this drive
- `-append`: pass the string as a command line to the kernel
- `-nographic`: disables graphical output so that QEMU is a simple command line application
- `-device` and `-netdev`: these commands are used together for creating a network device and configuring the relative interface
- `-serial`: redirects the virtual serial port to a host character device
- `-monitor`: redirects the monitor to a host device

Listing C.1: QEMU command

```
qemu-system-mipsel -m 2048 \
                   -M malta \
                   -kernel ./vmlinux.mipsel.4 \
                   -drive if=ide,format=raw,file=./1569030-14254692.squashfs_v4_le_extract_mipsel-8623 \
                   -append "root=/dev/sda1 console=ttyS0 nandsim.parts=64,64,64,64,64,64,64,64,64,64 rdinit=/firmadyne/preInit.sh rw debug ignore_loglevel print-fatal-signals=1 EMBA_NET=true EMBA_NVRAM=true EMBA_KERNEL=true EMBA_ETC=true user_debug=0 firmadyne.syscall=1" \
                   -nographic \
                   -device e1000,netdev=net0 \
                   -netdev tap,id=net0,ifname=tap290_0,script=no \
                   -device e1000,netdev=net1 \
                   -netdev socket,id=net1,listen=:2001 \
                   -device e1000,netdev=net2 \
                   -netdev socket,id=net2,listen=:2002 \
                   -device e1000,netdev=net3 \
                   -netdev socket,id=net3,listen=:2003  \
                   -serial file:./qemu.serial.log \
                   -serial telnet:localhost:4321,server,nowait \
                   -serial unix:/tmp/qemu.1569030-14254692.squashfs_v4_le_extract_mipsel-8623.S1,server,nowait \
                   -monitor unix:/tmp/qemu.1569030-14254692.squashfs_v4_le_extract_mipsel-8623,server,nowait
```

Listing C.2: Kernel Command-Line Parameters

```
-append "root=/dev/vda1 console=ttyS0 nandsim.parts=64,64,64,64,64,64,64,64,64,64 rdinit=/firmadyne/preInit.sh rw debug ignore_loglevel print-fatal-signals=1 EMBA_NET=true EMBA_NVRAM=true EMBA_KERNEL=true EMBA_ETC=true user_debug=0 firmadyne.syscall=1"
```

Password recovery

Further exploration revealed an unusual format in the `/etc/passwd` file. Typically, Linux systems authenticate users by referencing `/etc/passwd` and `/etc/shadow`. The `/etc/passwd` file lists user accounts with fields: Username, Password (usually '*' or 'x' if stored in `/etc/shadow`), UID, GID, Home directory, and Login shell. However, in the FG firmware, the `/etc/shadow` file was missing, and the password field in `/etc/passwd` contained hashed passwords, as illustrated in C.3.

Since the hash appeared relatively short and insecure the three hashed password correlated to the root, sdcard and webuser accounts were processed with John

the Ripper. The sdcard and webuser passwords were cracked within seconds, revealing the already known password *123456*. Cracking the root password required approximately seven days, but ultimately, the password *F6beasT* was found. This enabled root access to the system via the SSH service using the root credentials.

Listing C.3: Passwd file

```
1  root:qGKU1saPeFDV2:0:0:root:/:/bin/sh
2  ftp::14:50:FTP User:/var/ftp:
3  bin:*:1:1:bin:/bin:
4  daemon:*:2:2:daemon:/sbin:
5  nobody:*:99:99:Nobody:/:
6  sdcard:sbjHkuuFxmSvo:1001:1001:Linux User,,,:/sdcard:/bin/sh
7  webuser:HQolRVjV67MiQ:1002:1002:Linux User,,,:/mnt/appweb/web:/
   bin/sh
```

Bibliography

[1] Infosecurity Magazine. *Google Australia Office Attack*. URL: https://www.infosecurity-magazine.com/news/researchers-hack-googles-australian-office/ (cit. on pp. 1, 9).

[2] International Business Times. *Finland Heating System Attack*. URL: https://www.ibtimes.co.uk/hackers-leave-finnish-residents-cold-after-ddos-attack-knocks-out-heating-systems-1590639 (cit. on pp. 1, 9).

[3] Sipke Mellema Gjoko Krstic. *I Own Your Building (Management System)*. White Paper. Applied Risk BV, 2019. URL: https://www.slideshare.net/slideshow/i-own-your-building-management-system/192682282 (visited on 05/27/2025) (cit. on pp. 1, 6, 7, 17, 67).

[4] Christopher Morales-Gonzalez, Matthew Harper, Michael Cash, Lan Luo, Zhen Ling, Qun Z. Sun, and Xinwen Fu. «On building automation system security». In: *High-Confidence Computing* 4.3 (2024), p. 100236. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2024.100236. URL: https://www.sciencedirect.com/science/article/pii/S2667295224000394 (cit. on pp. 1, 7, 11, 16, 33).

[5] The EMBA Team. *EMBA*. 2024. URL: https://github.com/e-m-b-a/emba (cit. on pp. 3, 39).

[6] Pierre Ciholas, Aidan Lennie, Parvin Sadigova, and Jose Such. «The Security of Smart Buildings: a Systematic Literature Review». In: (Jan. 2019). DOI: 10.48550/arXiv.1901.05837 (cit. on pp. 6, 10, 17, 68).

[7] Andrzej Ożadowicz. «Generic IoT for Smart Buildings and Field-Level Automation—Challenges, Threats, Approaches, and Solutions». In: *Computers* 13.2 (2024). ISSN: 2073-431X. DOI: 10.3390/computers13020045. URL: https://www.mdpi.com/2073-431X/13/2/45 (cit. on pp. 6, 7).

[8] Karan Lohia, Yash Jain, Chintan Patel, and Nishant Doshi. «Open Communication Protocols for Building Automation Systems». In: *Procedia Computer Science* 160 (2019). The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2019) / The 9th International

Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops, pp. 723–727. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.11.020. URL: https://www.sciencedirect.com/science/article/pii/S187705091931720X (cit. on pp. 7, 10).

[9] Fábio Ferreira, A. Osório, João Calado, and C Pedro. «Building automation interoperability–A review». In: (2010). URL: https://api.semanticscholar.org/CorpusID:26730900 (cit. on pp. 7, 10).

[10] Pedro Domingues, Paulo Carreira, Renato Vieira, and Wolfgang Kastner. «Building automation systems: Concepts and technology review». In: *Computer Standards & Interfaces* 45 (2016), pp. 1–12. ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2015.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0920548915001361 (cit. on pp. 7, 11, 16).

[11] James Sinopoli. *Smart Building Systems for Architects, Owners and Builders*. Elsevier Inc., 2010. ISBN: 978-1-85617-653-8. DOI: https://doi.org/10.1016/C2009-0-20023-7 (cit. on p. 7).

[12] Anurag Verma, Surya Prakash, Vishal Srivastava, Anuj Kumar, and Subhas Chandra Mukhopadhyay. «Sensing, Controlling, and IoT Infrastructure in Smart Building: A Review». In: *IEEE Sensors Journal* 19.20 (2019), pp. 9036–9046. DOI: 10.1109/JSEN.2019.2922409 (cit. on p. 9).

[13] Steffen Wendzel. «How to increase the security of smart buildings?» In: *Commun. ACM* 59.5 (Apr. 2016), pp. 47–49. ISSN: 0001-0782. DOI: 10.1145/2828636. URL: https://doi.org/10.1145/2828636 (cit. on pp. 10, 68).

[14] Muhammad Usman Younus, Saif ul Islam, Ihsan Ali, Suleman Khan, and Muhammad Khurram Khan. «A survey on software defined networking enabled smart buildings: Architecture, challenges and use cases». In: *Journal of Network and Computer Applications* 137 (2019), pp. 62–77. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2019.04.002. URL: https://www.sciencedirect.com/science/article/pii/S1084804519301146 (cit. on p. 10).

[15] Keshav Kaushik, Akashdeep Bhardwaj, and Susheela Dahiya. «Framework to analyze and exploit the smart home IoT firmware». In: *Measurement: Sensors* 37 (2025), p. 101406. ISSN: 2665-9174. DOI: https://doi.org/10.1016/j.measen.2024.101406. URL: https://www.sciencedirect.com/science/article/pii/S2665917424003829 (cit. on pp. 11, 32).

[16] Omer Shwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. «Reverse Engineering IoT Devices: Effective Techniques and Methods». In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4965–4976. DOI: 10.1109/JIOT.2018.2875240 (cit. on pp. 11, 33).

[17] YongLe Chen, Feng Ma, Ying Zhang, YongZhong He, Haining Wang, and Qiang Li. *AutoFirm: Automatically Identifying Reused Libraries inside IoT Firmware at Large-Scale*. 2024. arXiv: 2406.12947 [cs.CR]. URL: https://arxiv.org/abs/2406.12947 (cit. on pp. 11, 32).

[18] Fabrice Bellard. «QEMU, a fast and portable dynamic translator». In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41 (cit. on pp. 12, 13).

[19] Daming Chen, Manuel Egele, Maverick Woo, and David Brumley. «Towards Automated Dynamic Analysis for Linux-based Embedded Firmware». In: *Proceedings of the 2016 NDSS Symposium*. Jan. 2016. DOI: 10.14722/ndss.2016.23415. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf (cit. on pp. 12, 13, 43).

[20] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. «FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis». In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. Austin, USA: Association for Computing Machinery, 2020, pp. 733–745. ISBN: 9781450388580. DOI: 10.1145/3427228.3427294. URL: https://doi.org/10.1145/3427228.3427294 (cit. on pp. 12, 13, 42).

[21] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. «HALucinator: firmware re-hosting through abstraction layer emulation». In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5 (cit. on p. 12).

[22] Andrew Fasano et al. «SoK: Enabling Security Analyses of Embedded Systems via Rehosting». In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 687–701. ISBN: 9781450382878. DOI: 10.1145/3433210.3453093. URL: https://doi.org/10.1145/3433210.3453093 (cit. on p. 12).

[23] Wei Zhou, Shandian Shen, and Peng Liu. «IoT Firmware Emulation and Its Security Application in Fuzzing: A Critical Revisit». In: *Future Internet* 17.1 (2025). ISSN: 1999-5903. DOI: 10.3390/fi17010019. URL: https://www.mdpi.com/1999-5903/17/1/19 (cit. on pp. 13, 70).

[24] Juhwan Kim, Jihyeon Yu, Hyunwook Kim, Fayozbek Rustamov, and Joobeom Yun. «FIRM-COV: High-Coverage Greybox Fuzzing for IoT Firmware via Optimized Process Emulation». In: *IEEE Access* 9 (2021), pp. 101627–101642. DOI: 10.1109/ACCESS.2021.3097807 (cit. on pp. 14, 70).

[25] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. «ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications». In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2847–2862. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/tychalas (cit. on pp. 14, 70).

[26] Yue Zhang, Zhen Ling, Michael Cash, Qiguang Zhang, Christopher Morales-Gonzalez, Qun Zhou Sun, and Xinwen Fu. «Collapse Like A House of Cards: Hacking Building Automation System Through Fuzzing». In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 1761–1775. ISBN: 9798400706363. DOI: 10.1145/3658644.3690216. URL: https://doi.org/10.1145/3658644.3690216 (cit. on pp. 14, 16, 70).

[27] Marco Caselli, Emmanuele Zambon, Johanna Amann, Robin Sommer, and Frank Kargl. «Specification mining for intrusion detection in networked control systems». In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 791–806. ISBN: 9781931971324 (cit. on p. 17).

[28] Siemens. *Desigo ABT Firmware library*. URL: https://support.industry.siemens.com/cs/document/109780759/desigo-abt-firmware-library (cit. on p. 19).

[29] Johnson Controls. *EasyIO Firmware Library*. URL: https://www.solutionnavigator.com/s/contentnavigator?language=en_US&id=aAO4w000000GrAM&app=Resources%2FControls%2FEasyIO&country=US&region=NAM (cit. on pp. 19, 68).

[30] WAGO. *WAGO Firmware Library*. URL: https://downloadcenter.wago.com/wago/software (cit. on p. 19).

[31] Contemporary Controls. *Contemporary Controls Firmware Library*. URL: https://ccontrol.ccontrols.com/basautomation/ (cit. on p. 19).

[32] The Binwalk Team. *Binwalk: A Firmware Analysis Tool for reverse engineering and extracting firmware images*. Version v2.3.2. 2024. URL: https://github.com/ReFirmLabs/binwalk (cit. on pp. 20, 33).

[33] The Binwalk Team. *Binwalk: A Firmware Analysis Tool for reverse engineering and extracting firmware images. Rust implementation*. Version 3.1.1. 2024. URL: https://github.com/ReFirmLabs/binwalk (cit. on pp. 20, 33).

[34] C. E. Shannon. «A mathematical theory of communication». In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x (cit. on p. 20).

[35] Apostol Vassilev and Timothy A. Hall. «The Importance of Entropy to Information Security». In: *Computer* 47.2 (Feb. 2014), pp. 78–81. ISSN: 0018-9162. DOI: 10.1109/MC.2014.47. URL: https://doi.org/10.1109/MC.2014.47 (cit. on p. 22).

[36] Elliott Wen, Jiaxing Shen, and Burkhard Wuensche. *Keep Me Updated: An Empirical Study of Proprietary Vendor Blobs in Android Firmware*. 2024. arXiv: 2410.11075 [cs.SE]. URL: https://arxiv.org/abs/2410.11075 (cit. on p. 32).

[37] OWASP. *Firmware Security Testing Methodology*. URL: https://scriptingxss.gitbook.io/firmware-security-testing-methodology (cit. on p. 32).

[38] The QEMU Team. *QEMU: Quick Emulator*. Version v6.2.0. 2024. URL: https://www.qemu.org/ (cit. on p. 38).

[39] Siemens. *Climatix BACnet Library*. URL: https://mybuilding.siemens.com/d015628993239/help/engineeringhelp/en-us/index.html#14833332107 (cit. on p. 44).

[40] Forum of Incident Response and Security Teams. *CVSS*. URL: https://www.first.org/cvss/ (cit. on p. 46).

[41] Omar Santos. *The Evolution of Scoring Security Vulnerabilities: The Sequel*. URL: https://blogs.cisco.com/security/cvssv3-study (cit. on p. 47).

[42] National Vulnerability Database. *Vulnerability Metrics*. URL: https://nvd.nist.gov/vuln-metrics/cvss (cit. on p. 48).

[43] The MITRE Corporation. *Common Weakness Enumeration*. URL: https://cwe.mitre.org/index.html (cit. on p. 49).

[44] National Vulnerability Database. *NVD CWE Slice*. URL: https://nvd.nist.gov/vuln/categories (cit. on p. 49).

[45] Wikipedia. *Linux kernel version history*. URL: https://en.wikipedia.org/wiki/Linux_kernel_version_history (cit. on p. 50).

[46] GNU Operating System. *GCC Releases*. URL: `https://gcc.gnu.org/releases.html` (cit. on p. 55).