POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Firmware Development and Certification for IoT Devices

Supervisors

Prof. Luca ARDITO

Dr. Michele VALSESIA

Candidate

Giuseppe Marco BIANCO

April 2024

Abstract

The Internet of Things (IoT) presents itself as an innovative technology that facilitates the interconnection of devices and physical systems via the Internet, allowing the exchange of data and the execution of automated actions. This thesis aims to examine the large number of challenges a developer faces when developing firmware for IoT devices, as well as the essential criteria for their certification.

The introduction outlines the fundamental significance of IoT in the contemporary technological landscape, highlighting its crucial role in establishing a digital ecosystem in which everyday objects acquire intelligence and the ability to communicate fluidly. We explore the evolution of IoT, from its embryonic phase to its practical applications in various fields, with particular attention to its impact on smart home environments.

From a developer's perspective, developing firmware for IoT devices presents unique challenges:

- Ensuring code robustness,
- Managing its complexity,
- Resolving security issues.

Therefore, ci-generate is introduced as a tool aimed at streamlining the process of establishing Continuous Integration (CI) pipelines that perform:

- Code Formatting
- Code Linting
- Static Code Analysis
- Code Coverage Analysis
- Dependency Management
- Unsafe Code Checks

In addition, to ensure greater security of the binary running in a smart home, the firmware developed for IoT devices must undergo a certification process. Which requires a thorough analysis of the firmware binaries, starting from the identification of basic information, such as found APIs and architectural details, up to the detailed examination of specific characteristics, such as the generation of a syscall flow for each API. To this end, we have introduced the **manifest-producer**, a tool designed to extract the essential information needed for the certification process, culminating in the production of a detailed JSON manifest that provides a representation of the characteristics identified during the analysis.

The implementation of the ci-generate tool has reduced the effort to create a Continuous Integration pipeline for the manifest-producer tool. Additionally, a thorough analysis was conducted on the performance of the latter, including an analysis on real projects. This assessment focused primarily on factors such as execution speed and memory consumption, in order to ensure the efficiency and effectiveness of the tool in practical contexts. It is worth noting that both tools have been written in Rust, a language known for its performance and reliability.

Acknowledgements

"Desidero esprimere la mia profonda gratitudine alla mia famiglia, una presenza fondamentale nel mio percorso accademico. Vi ringrazio sinceramente per la vostra comprensione profonda e discreta, per aver rispettato i miei spazi senza aggiungere ulteriori pressioni al percorso della mia formazione. Il vostro sostegno è stato un faro costante, illuminando le mie scelte e le mie sfide. La vostra presenza e saggezza hanno arricchito straordinariamente questo viaggio, conferendogli un valore unico e prezioso. Con immensa gratitudine, dedico questo traguardo a voi."

Giuseppe

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the context of firmware development for IoT devices, developers often face complex challenges related to code robustness and system security. These challenges may include managing code complexity, handling changes and versions, as well as implementing effective testing procedures to ensure firmware stability and security. To address these challenges, we introduce ci-generate: a tool designed to simplify Continuous Integration in software development workflows. Ci-generate facilitates code optimization and rapid error detection, thus helping to improve the overall quality of the firmware.
Furthermore, the important role of firmware certification in ensuring safety and reliability standards is highlighted. In this regard, the manifest-producer has been developed, a tool designed to extract and encapsulate the fundamental characteristics of the firmware, to simplify the certification process. In this context, we delve into the role of binaries in the ELF format. These files represent an essential element in the development and analysis process of firmware for IoT devices, as they contain the executable instructions and data necessary to perform a consistent and comprehensive certification.

The thesis is structured in the following chapters:

- **Introduction to the Internet of Things:** In this chapter, we explore the current landscape of the Internet of Things, with a particular focus on the evolution of technology within the smart home environment, as well as key concepts related to the development and security of firmware for IoT devices.
- **A tool for Continuous Integration:** In this chapter, we explore the paradigm of Continuous Integration as a fundamental approach to secure and efficient software development. Specifically, we delve into the capabilities of the ci-generate tool. Through the process of refactoring its code and adding a

test suite, we examine in detail how ci-generate can contribute to improving the quality and reliability of the developed software.

- **A dummy firmware for IoT:** In this chapter, we introduce a dummy firmware implementation, designed to simulate the behaviours of IoT devices and address challenges such as:
  - Significance of developing firmware in multiple programming languages.
  - Address challenges in firmware development and testing.

  The chapter also describes the design and structure of the dummy firmware, including its architecture, compilation and deployment processes for each firmware variant. The main variants are implemented in C, C++ and Rust, with each of them subject to both static and dynamic compilation. Additionally, variants with and without debug symbols are generated. Finally, a specific variant in C is designed to introduce APIs that do not conform to expected behaviours, to examine as many scenarios as possible for analysis.
- **A tool for firmware certification:** In this chapter, we explore the approach to certifying firmware for IoT devices through the development of the manifest-producer tool. Specifically, we follow the path that led from preliminary analyses of ELF binaries to the process of identifying parameters useful for this certification.
- **Performance analysis:** In this chapter, we analyze the performance of the manifest-producer tool in evaluating several ELF files, including FFmpeg, OpenCV and xi-core. Each of the ELF files has been obtained from the building of the projects considered. Hyperfine and Heaptrack have been used to measure execution times and memory usage, offering a targeted overview of the tool's capabilities.
- **Conclusions:** In this final chapter, we summarize the key findings obtained from our research and explore avenues for the development and future developments of IoT firmware certification through the manifest-producer tool.

Chapter 2

Introduction to the IoT

2.1 Definitions

The **Internet of Things (IoT)** [1] is an emerging technological paradigm based on the interconnection of physical devices, objects and systems through Internet. This interconnection facilitates the collection, exchange of data and execution of automatic actions, helping to create a digital ecosystem (see Figure 2.1) in which physical objects become intelligent and capable of communicating without direct human intervention.

In this context, IoT is the key to creating an environment where every device, from the intelligent light bulb to complex industrial sensors, is equipped with Internet connectivity and data collection capabilities. This convergence paves the way for several practical applications, including remote monitoring, home automation, healthcare, smart agriculture and industry 4.0.

The key definition of IoT is based on **extending connectivity** and **intelligence** to physical devices. This extension is made possible by advanced technologies such as sensors, actuators, communication networks and cloud computing platforms, providing sophisticated functionality and constant connectivity. Combining these elements enables devices to gather data, communicate with each other, and use that data to make decisions.

However, the objective and concrete aspect of IoT raises important challenges, especially in terms of **security**, **privacy** and **data management**. Increasing interconnectedness makes it crucial to address these issues to prevent unauthorized access, cyber-attacks and privacy violations [2, 3].

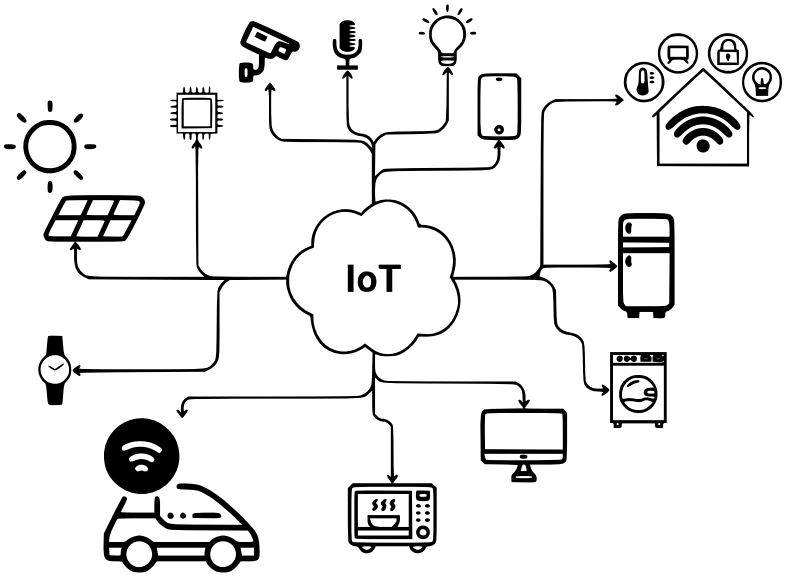

Figure 2.1: IoT ecosystem.

To sum up, the Internet of Things represents a milestone in the digital transformation of the physical world, bringing with it a series of benefits and complex challenges. Exploring this definition and its fundamental concept provides a solid starting point for examining the history and development of IoT. It will be interesting to analyze how this technological paradigm has taken shape over time, with particular attention to the crucial phases that have shaped the current IoT landscape.

2.1.1 History and evolution

The Internet of Things represents one of the most significant evolutions in the digital technology field. This sector has deep roots in past technological innovations and has undergone considerable evolution over the years. In this section, the history and evolution of the Internet of Things (IoT) will be explored, analyzing the key that contributed to its rapid diffusion and adoption as a technology.

The discourse surrounding the IoT took root earlier, notably in 1982, when the concept of a network of intelligent devices was addressed. A noteworthy instance occurred at Carnegie Mellon University, where a modified Coca-Cola vending machine became the inaugural ARPANET-connected appliance [4]. Additionally, the seminal paper on ubiquitous computing by Mark Weiser in 1991, titled The Computer for the 21st Century, played a pivotal role in shaping the contemporary vision of IoT [5].

In 1994, Reza Raji expounded upon the IoT concept in IEEE Spectrum, characterizing it as the transmission of small data packets to an extensive network of nodes, aiming to integrate and automate various facets ranging from household appliances to entire industrial complexes [6]. However, the true genesis of IoT can be attributed to the work of pioneers such as Kevin Ashton, who coined the term Internet of Things in 1999 [7]. Ashton, a researcher at MIT, anticipated the idea of connecting physical objects to the Internet to improve data collection and management.

In the early 2000s, with the advancement of wireless networks and sensor technologies, experimental IoT designs and applications began to emerge. Using RFID (Radio-Frequency Identification) and wearable sensors was a significant step towards the practical realization of IoT [8, 9]. Projects such as Ambient Intelligence and Smart Home began to demonstrate the potential of connecting everyday objects to a network [10, 11].

The past decade has seen an impressive surge in the Internet of Things (IoT) industry, with 2010 to 2020 being a particularly notable period. Thanks to the drop in sensor expenses, the increase in computing capabilities, and the emergence of specialized IoT networks, connecting more devices than ever has become a reality. A wide range of sectors, such as healthcare [12, 13], agriculture [14], manufacturing and logistics [15], have embraced IoT technology on a grand scale to optimize their operational efficiency and streamline their data management.

As the Internet of Things (IoT) accelerates, significant security, privacy and interoperability challenges emerge. Security is the primary concern in the process of adopting this technology, as there is anxiety that its rapid development will occur without thoughtful consideration of the complex security challenges involved and any necessary regulatory changes [16]. The rapid expansion of the IoT has facilitated the connection of billions of devices to the network, but has created security challenges attributable to the vast number of connected devices and limitations of communications security technology [17, 18]. The widespread diffusion of the Internet of Things (IoT) has generated an urgent need for standardization to avoid a stall and to aim at ensuring consistency in communication protocols and security requirements [19, 20]. The evolution of the IoT, with the integration of emerging technologies such as artificial intelligence and 5G, has outlined sophisticated applications such as smart cities and Industry 4.0, anticipating future expansion into new sectors [19]. However, rapid development has raised concerns about safety and lack of regulation, prompting standardization initiatives such as IoTSFW [19]. At the same time, actors such as the Open Interconnect Consortium and the Connected Home over IP group emerged, reflecting the need for a collaborative approach to overcome fragmentation in IoT standards [21, 22].

To sum up, IoT has gone through a remarkable evolution from its conceptual roots to its current omnipresence in digital society. Its history is marked by technological advances, overcoming challenges and a continuous commitment to creating an interconnected and intelligent ecosystem. In the future, IoT continues to represent a dynamic field of research and development, with the potential to further transform interactions between the digital and physical worlds.

2.1.2 IoT devices

As detailed by Sharu Bansal and Dilip Kumar in their article [23], the IoT ecosystem fundamentally relies on crucial components known as IoT devices, which form the cornerstone of its complex architecture. These devices play a fundamental role in all layers of the IoT framework. Characterized by limited internal resources, including storage, memory and computing capacity, IoT devices leverage various types of memory technologies. The broad classification (Class 0, Class 1, and Class 2) clarifies the different roles that these devices take on within the IoT environment, ranging from basic sensing and actuation capabilities to the advanced capabilities of the single-board computers that support artificial intelligence. The following sections will delve into the classification and exemplification of these devices, providing a comprehensive overview of their significance in shaping the IoT ecosystem landscape.

Main categories

- **Monitoring devices and sensors:** This category of devices includes instruments designed to collect environmental data or specific parameters, such as temperature, humidity, atmospheric pressure and motion sensors. They play a crucial role in applications such as smart agriculture, energy management and environmental surveillance. These sensors, supported by academic research, contribute significantly to the advancement of key sectors, providing essential data for informed decisions and efficient implementations in contemporary challenges related to sustainability and resource optimization [24, 25].
- **Wearable devices:** This category includes wearable devices such as smartwatches, fitness trackers and smart glasses, which collect data on physical activities, health and other personal parameters. This data collection plays a fundamental role in monitoring individual well-being. The use of such devices provides a detailed picture of daily habits, offering significant opportunities to improve health management and promote an active lifestyle [26, 27].
- **Smart Home devices:** These devices, known as home automation devices, are designed to optimize and simplify daily operations within homes. These

include smart thermostats, connected lamps, smart appliances and security cameras. Their interconnection allows users to manage and monitor them remotely through dedicated applications or voice commands. This technological integration not only increases energy efficiency and safety, but also represents a significant step towards the implementation of smart homes, in tune with the concept of the Internet of Things (IoT) [28, 29].

- **Industrial IoT devices:** Industries such as manufacturing, energy, and transportation use IoT devices to optimize operations. Machine monitoring sensors, logistics tracking devices and predictive maintenance systems fall into this category, helping to improve efficiency and reduce operating costs [30, 31, 32].
- **Connected Vehicles:** Vehicles such as cars, drones and other forms of transportation are increasingly integrating IoT technologies to enhance safety, navigation, and the driving experience. Through the connection, these vehicles can exchange data with each other and with the road infrastructure, optimizing traffic flow and preventing accidents. This implementation of connected devices not only promotes greater efficiency in the transportation sector but also offers fertile ground for academic research in the field of intelligent vehicular networks and the Internet of Things applied to mobility [33].

IoT Devices examples

- **Smart thermostat:**[1] This device embodies the transformative potential of smart home devices, showcasing how IoT enhances daily living experiences. Its ability to autonomously adjust to user preferences represents a tangible outcome of the interplay between connectivity, data analytics, and intelligent decision-making – key elements that define the broader landscape of the Internet of Things in smart homes.
- **Activity and health tracker:**[2] This wearable device, belonging to the category of connected objects, not only actively monitors physical activity, sleep quality and other health-related parameters through advanced sensors such as accelerometers and heart monitors, but also uses intelligent algorithms to analyze the collected data. Thanks to wireless connectivity, users can synchronize information on a dedicated app or cloud platform, obtaining detailed

[1]Smart thermostats are Wi-Fi thermostats responsible for controlling the heating, ventilation, and air conditioning inside the home.

[2]An activity tracker involves the practice of measuring and collecting data on an individual's physical activity to keep track and maintain documentation regarding their health.

analyses of their sleep habits and physical activity levels. Furthermore, the device encourages active user engagement by providing personalized suggestions to improve well-being, thus highlighting the active and interactive role of wearables in promoting a healthy lifestyle.

- **Smart agriculture sensors:**[3] These advanced devices allow detailed control of soil conditions, optimizing irrigation practices and driving significant yield increases in agricultural sectors. The ability to collect data in real time and adapt agricultural practices to specific environmental conditions contributes significantly to operational efficiency and sustainability in the agricultural sector.
- **Autonomous car:**[4] These vehicles use an advanced network of sensors and cameras to perceive their surroundings, analyzing visual and audio data both inside and outside the vehicle. The control system interprets this information to understand the context and subsequently makes autonomous navigation decisions, considering the route, road conditions, traffic signs and obstacles.
- **Smart grid sensors:** In the energy sector, they play a crucial role, constantly monitoring the electricity grid. This constant surveillance not only allows for more efficient management of energy resources but also the timely identification and resolution of any faults. The integration of these sensors contributes significantly to operational optimization, reflecting the broad scope of impact of IoT in transforming key industries.

In summary, the Internet of Things (IoT) has rapidly transformed from a conceptual idea to a ubiquitous reality, impacting diverse industries. The proliferation of IoT devices, ranging from wearables to industrial sensors, highlights the versatility and potential for optimizing daily activities. However, the accelerated growth of IoT necessitates a proactive approach to address security, privacy, and interoperability challenges, ensuring the continued success and widespread adoption of this transformative technology.

2.2 Smart Home integration

Contextualization of the Smart Home is configured as the process of integration and concrete application of the Internet of Things (IoT) within the domestic environment, aimed at enhancing and optimizing the quality of users' daily lives.

[3]Smart agriculture digitally collects, stores, analyzes, and shares electronic data and/or information in agriculture.

[4]Autonomous car, is a car that is capable of operating with reduced or no human input.

This technological convergence, characterized by the interconnection of devices and systems, takes on a crucial role in transforming living spaces into intelligent contexts that are responsive to the needs of the inhabitants.

During this analysis, some key sections of this contextualization will be explored, focusing on the tangible impacts that the synergy between the Internet of Things (IoT) and the home environment can generate on daily activities. In particular, how this interconnection facilitates the efficient management of resources, housing safety, and the personalization of experiences at home will be examined. From an academic perspective, the dynamics underlying this emerging paradigm will be outlined, highlighting how the Smart Home can become a significant fulcrum in the evolution of the way of conceiving and experiencing domestic space.

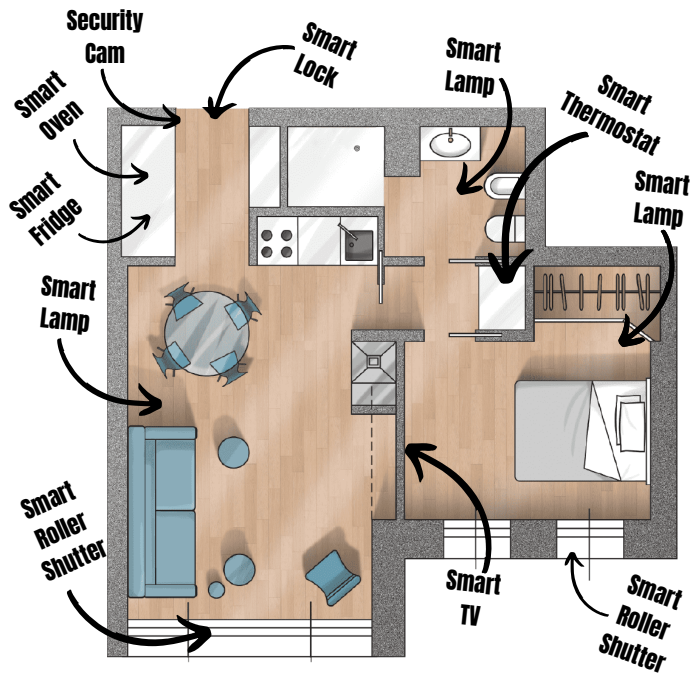

Figure 2.2: Smart Home environment.

2.2.1 Concept and features

The introduction of the **Smart Home** represents an evolved paradigm of the domestic environment, consisting of an intricate ecosystem in which a variety of household electronic devices interact synergistically. These devices are connected and controlled across a network, often leveraging Internet of Things (IoT) technologies.

The main objectives of this innovative residential configuration are the elevation of efficiency, safety and comfort in the daily activities of the inhabitants. This living scenario stands out for the use of advanced home automation systems that coordinate the functionality of interconnected devices, simplifying and optimizing home routines [34]. A fundamental element of the Smart Home are intelligent devices, as they not only proactively respond to user needs, but also personalize the home experience, creating an interactive and adaptive environment. The foundations of this emerging configuration are supported by academic studies. For example, research proposed an integrated model based on the technology acceptance model, the diffusion of innovation theory and consumer-perceived innovativeness, using Structural Equation Modeling to validate the model [34]. The findings of this study highlight the importance of factors such as compatibility, perceived usefulness, and perceived ease of use in the adoption of smart home technology.

Other insights [35] focus on the challenges of smart homes as a new living concept, underlining the importance of understanding users' needs and preferences to encourage acceptance of such changes in personal spaces. The authors connect users' preferences to their lifestyles, proposing a redefinition of the concept of housing that takes into account user-centred design and future lifestyles. Furthermore, [35] presents a multidimensional classification framework of the Smart Home, integrating Virtual Space, Intelligent Environmental Space and Physical Space. This new concept of the domestic environment, in which these spaces are integrated, is proposed as the key to better understanding the elements and spaces of this future living environment.

In summary, the Smart Home is configured as a cutting-edge living context, where the integration of intelligent devices, the personalization of experiences and the synergy between physical and virtual elements define a new frontier in the evolution of modern homes.

2.2.2 Impacts on daily life

The interconnection of devices in the Smart Home has profound impacts on daily life. Some of these impacts include:

- **Improved energy efficiency:** The seamless connection between electronic devices and energy efficiency is a crucial aspect of modern technology. Smart homes enable optimized communication between various devices, resulting in a more intelligent response to user needs [36]. Sophisticated sensors are integrated to provide real-time monitoring of both internal and external conditions. This feature not only enhances security by detecting and promptly responding to any anomalies but also facilitates efficient management of resources. [37].

Energy management is a fundamental pillar of the Smart Home and plays a critical role in reducing unnecessary energy consumption. The intelligent coordination of electronic devices in the smart home environment aims to achieve this goal. The importance of smart home technology in optimizing energy consumption in buildings is highlighted in [38], which thoroughly analyzes specific applications, such as energy management systems.

To put it briefly, the Smart Home aims to offer comfort, safety, and sustainable solutions through technological synergy and the intelligent integration of innovative devices and systems. The multidimensionality of this environment emerges in the joint analysis of the different papers, emphasizing its tangible impact on daily life.

- **Greater comfort and automation:** Automating daily activities, such as regulating the temperature and turning lights on/off, is crucial in offering greater comfort and time savings in Smart Homes. In this context, various academic research demonstrates how practical projects, such as the automatic lighting system with Arduino [39], can significantly improve convenience and energy efficiency. Using key components such as Arduino, a PIR sensor, and a relay module, this project illustrates lighting automation based on human presence. This concept is extendable to other scenarios, such as automatic bathroom flush valves, highlighting the effectiveness of simple technologies, such as the PIR sensor, in improving daily life [39].

  From the analysis of the usage of IoT-based Smart Home technology in Malaysia [40], six main uses emerge, including real-time remote control, surveillance, home automation and entertainment. IoT-SHT (Internet of Things Smart Home Technology) has proven to significantly contribute to saving time by improving aspects such as safety, environmental conditions and home convenience. The implementation of IoT technologies in this context also has positive consequences on a psychological level, improving image, company and the sense of family belonging [40].

  In the context of optimizing comfort and energy consumption in smart homes, an innovative approach is presented in the work that proposes a bat algorithm (BA) [41]. This algorithm considers three crucial parameters for occupant comfort: temperature, lighting and indoor air quality. Introducing an exponentially increasing inertia weight significantly improves performance, ensuring optimal comfort with minimal energy consumption. The application of this algorithm represents a significant contribution to achieving a more efficient and comfortable smart home environment [41].

  As demonstrated, these academic approaches offer innovative solutions to

improve daily life in Smart Homes, ensuring optimal comfort and maximizing energy efficiency. Technologies such as lighting automation, IoT-SHT and optimization algorithms represent significant steps towards a smart and connected living future.

- **Challenges and advances in Smart Home security:** Platform fragmentation and the lack of technical standards in the field of home automation represent significant challenges, with the diversity of home automation devices complicating the development of coherent applications across heterogeneous technological ecosystems [42].

  Security plays a crucial role in smart homes, where the interaction between remote users and devices takes place through gateways. User authentication is essential, but the vulnerability of schemes proposed over the years raises significant concerns. Meeting these challenges requires a thorough evaluation of existing approaches and the development of secure schemes [43].

  In parallel, unauthorized operation in IoT-based Smart Home Systems (SHS) presents additional security challenges. The review of advanced approaches to ensure the operational security of such systems reflects the importance of understanding security threats and taking appropriate measures [44].

  Voice interface attacks are another critical area for consumer Internet of Things (IoT) platforms for smart homes. Here, the need for effective countermeasures becomes evident, and the introduction of a voice-liveness detection system is an example of how research aims to protect privacy and improve the security of smart homes [45].

  Through an integrated approach, the cited studies contribute significantly to the advancement of security in smart homes, addressing crucial challenges related to authentication, operational security and privacy protection. An in-depth analysis of these aspects provides a complete and comprehensive view of the smart home security landscape, which is essential for ensuring a safe and reliable smart home environment.

  It is crucial to underline that these works represent only one segment of a vast research, all committed to revealing the complex security issues related to the Internet of Things (IoT) expansion in Smart Homes. Attention to and resolution of these issues is critical to drive the development of better solutions and practices, ensuring user security and privacy in an increasingly interconnected and intelligent environment. Continued research and collaboration between industry sectors, academic communities and regulatory organizations remain critical to proactively address emerging challenges in the Smart Home and IoT landscape.

2.3 Development and Security in IoT Devices

In this section, the analysis is moved from the perspective of a consumer to that of a developer, with a particular emphasis on the complex and fundamental domain of software in Internet of Things (IoT) devices. The focus will be on understanding how Continuous Integration (CI)[5], when adopted effectively, can optimize the management of the complexities related to the development of firmware for IoT devices. This method aims to enhance code consistency, promptly identify errors, and streamline frequent updates, ultimately enhancing software reliability and security.

In parallel, the crucial role of binary analysis in revealing the correct functioning of the firmware of an IoT device will be examined. Although intrinsically complex, this methodology offers an in-depth analysis of the binary structure of the firmware, identifying potential vulnerabilities and ensuring a high standard of quality and security.

In summary, this section aims to offer an objective and academic approach to understanding software development processes for IoT devices, emphasizing the importance of continuous integration and binary analysis in the context of advanced and secure development.

2.3.1 Continuous Integration

The software development process relies heavily on **Continuous Integration (CI)** as a pivotal paradigm that bolsters efficiency and quality. Fowler and Foemmel first presented CI in 2006 [46], encompassing ten key practices that have since gained widespread acceptance across the industry. Empirical studies have attested to the positive influence of these practices on software quality.

The Continuous Integration (CI) approach revolves around the principle of constantly merging code alterations into the primary codebase, guaranteeing that the software remains consistently operational, tested and prepared for release. The objective is to expedite feedback mechanisms during the development phase, detect any potential issues at an early stage, and foster better teamwork among the development team.

A multi-centre analysis conducted on small to medium-sized companies investigated the implementation of Continuous Integration (CI) practices in the context of software development. Through a combined approach of in-depth interviews and

[5]Continuous Integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project.

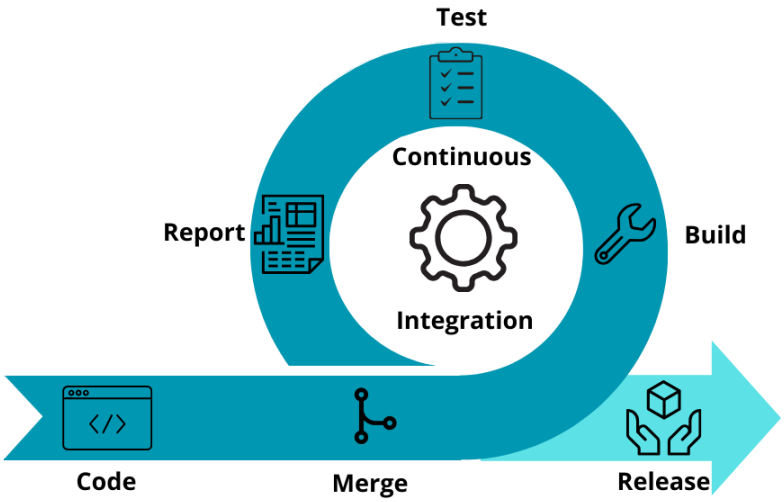

Figure 2.3: Continuous Integration steps

analysis of activity logs, a varied picture emerged on the adoption of such practices. The survey results indicated widespread adoption of CI practices within the companies examined. However, it became clear that there is considerable variation in how these practices are implemented. These divergences have been observed concerning the benefits perceived by development teams, the specific context of each project, and the specific CI tools used within organizations [47].

Furthermore, in the context of open-source development, detailed analysis has shown that when a version of a software project fails all tests during CI, often only small adjustments in pull requests are needed. This highlights the importance of handling pull requests carefully, as their size can greatly influence their success. This correlation highlights the need to adopt mature Continuous Integration practices to improve the efficiency of this process in open-source environments [48].

To sum up, Continuous Integration (CI) is a fundamental approach that revolutionizes the software development process, improving quality and accelerating delivery, with academic research and case studies highlighting its application and impact in diverse contexts.

2.3.2 IoT Firmware

IoT Firmware represents a fundamental component in Internet of Things (IoT) devices, defining itself as the software incorporated directly into the hardware device. Unlike other types of firmware that may be present in more generic devices, IoT Firmware is specifically designed to enable and manage IoT devices' connected and intelligent features. This type of firmware is closely linked to the peculiarities

of the IoT environment, ranging from advanced sensors to network interconnections, favouring the implementation of control logic and data collection.

The role of IoT Firmware is of utmost importance in the Internet of Things (IoT) ecosystem. It not only oversees the fundamental functions of devices but also manages data access and network communications. Since these devices often operate in sensitive environments like smart homes (section 2.2), where a lot of data is collected, ensuring the security of the firmware is crucial.
Vulnerabilities in firmware pose a significant threat to the security of the entire IoT system, bringing with them risks such as unauthorized access, exposure of sensitive data, and compromise of control functions. Effectively managing these vulnerabilities becomes imperative to counteract potential threats, including malicious code injection [49, 50] and communications interception [51, 52], that could compromise data integrity [53, 54, 55].

IoT firmware security differs significantly from that of traditional computing devices, where established standards such as UEFI [6] provide a stable foundation. In the IoT context, firmware developers often lack security awareness, and frequently connecting devices to the Internet increases the risk of exploiting vulnerabilities. The diversity of architectures introduces additional challenges in standardizing security solutions, while the limited resources of IoT devices require balanced trade-offs between performance and security [55]. The situation is made even more intricate due to the absence of standardized firmware update methods and the common practice of using a single firmware image for multiple devices. This intensifies the ongoing insecurity in the realm of IoT since vulnerabilities can be leveraged across a broad spectrum of devices [55].

Current solutions for assessing IoT security face challenges related to legacy analysis techniques and IoT-specific limitations, which hinder firmware reverse engineering and vulnerability analysis procedures. In this panorama, binary firmware analysis emerges as a crucial element to mitigate risks and ensure the safe operation of IoT devices in the era of global interconnection.

2.3.3 Binary Analysis

- **Executable binaries**
  Executable binaries are files that contain not only the code to be executed but also a wealth of information necessary for the proper functioning of a program.

[6] Unified Extensible Firmware Interface is a specification that defines the architecture of the platform firmware used for booting the computer hardware and its interface for interaction with the operating system.

These files encapsulate data, code, and additional management information, including details about memory allocation, symbols, and dynamic linking for libraries.

1. **Data:**
   - *Static/Dynamic Memory:* Binaries include static and dynamic memory usage information, specifying how memory is allocated and utilized during program execution.
   - *Fixed/Preallocated Values:* Certain values, whether fixed or preallocated, are stored within the executable to maintain consistency during runtime.
2. **Code:**
   - The core instructions that define the program's functionality are stored in the binary. This section contains the machine code that the processor executes to perform the desired operations.
3. **Management Informations:**
   - *Memory Allocation:* Details about how the program's memory is allocated and managed are included in the binary, ensuring the proper functioning of the application.
   - *Symbols:* Symbols represent named entities such as functions, variables, and libraries, providing a structured way to reference different elements within the program.
   - *Dynamic Linking:* Information related to dynamic linking enables the program to load shared libraries during runtime, enhancing flexibility but introducing additional complexity compared to static linking.

- **ELF Format**
  The Executable and Linkable Format (ELF) is a standard file format for executables, object code, shared libraries, and even core dumps on Unix-like operating systems, particularly Linux. The ELF format features a header table that provides information on how to create the memory image of the program. This table is divided into segments, each containing specific details about the executable.
  - *Symbols:* ELF includes a symbol table, where symbols represent named entities within the program, including functions, variables, and libraries.
  - *Dynamic Linking:* ELF supports dynamic linking, allowing the resolution of library dependencies during runtime.

- **Binary Analysis Objectives**

  - **Revealing Vulnerabilities**
    Analysis of binary firmware is a crucial technique for identifying latent weaknesses in code, which is vital in mitigating risks and ensuring device security. A vulnerability is a software flaw that, if exploited, can be used by an attacker to infiltrate the system. These flaws can stem from a variety of sources, including oversights in data management, programming errors, or issues with the software's design.
    The primary goal is to detect and comprehend potential vulnerabilities while preempting any hostile attacks. Binary analysis provides a means of inspecting firmware at the machine code level, scrutinizing instructions and data structures to pinpoint any potential weaknesses that may be exploited. As a result, specialists can create preventive measures, rectify vulnerabilities, and bolster the overall security of the firmware.

  - **Ensure Compliance**
    In addition to vulnerability disclosure, binary analysis of firmware plays a crucial role in ensuring firmware compliance with security standards and development policies. Standards and policies define the best practices and guidelines that firmware should follow to ensure a secure and reliable environment [56].
    Binary analysis is a method used to objectively evaluate whether firmware adheres to established standards. Through this process, any deviations from these standards can be identified, which may potentially compromise security. This process helps ensure that the firmware meets regulatory requirements and security specifications, providing a solid foundation for the reliable operation of IoT devices.
    For effective and objective binary analysis, consulting specialized online resources and communities of experts in the field of cybersecurity is recommended. These resources provide valuable insights and expertise to ensure accurate analysis and interpretation of firmware. Forums, official documentation, and how-to guides can provide useful guidance for addressing specific challenges during binary firmware analysis. Sharing knowledge online is essential to stay up to date on new threats and advanced analysis techniques. This way, the community can work together to continually improve security practices and mitigate vulnerabilities across the vast Internet of Things landscape.

- **Methodologies and tools**
  Binary analysis can be a complex and detailed part of the security process for IoT devices. This practice involves exploring the binary code derived from the

firmware compilation, offering a detailed view of the implemented structures and functionality. Some of the methodologies for carrying out this type of analysis will be presented below.

- **Reverse Engineering**
  - **Definition:** Reverse engineering is the process of analyzing a system to understand its operation or to identify the design and implementation of an object without having access to its source code.
  - **Application:** In the context of binaries, reverse engineering is often used to examine the behaviour of a program or firmware, identify vulnerabilities, or understand how an application works without access to the source code.
- **Control Flow Reconstruction**
  - **Definition:** Control flow analysis is a methodology focused on comprehending the sequence of instructions and execution paths within a program.
  - **Application:** Reconstructing the control flow is essential for understanding how the program executes instructions, identifying potential vulnerabilities, and pinpointing critical points within the codebase.
- **Disassembler**
  - **Definition:** A disassembler is a reverse engineering tool that statically converts a program's machine code into human-readable assembly code without the need to execute the program.
  - **Application:** It enables the examination of executable code at a lower level, facilitating static analysis of a program's behaviour, aiding in vulnerability research, malware analysis, and understanding program logic without running the code directly.
- **Decompiler**
  - **Definition:** A decompiler is a reverse engineering tool that converts machine code into a high-level programming language similar to the source code.
  - **Application:** Useful for obtaining a more understandable representation of the code, the decompiler helps to understand the logic of a program without necessarily having to understand the assembly.

- **Debugger**
  - **Definition:** A debugger or a debugging tool is a program that allows developers to run a target program in a controlled mode, monitoring various aspects of execution such as variables, instructions, and execution flow.
  - **Application:** Used to analyze program behaviour at run time, the debugger is essential for finding and fixing errors.
- **Tracing**
  - **Definition:** It is a sort of debugger, with the difference that it tries to better understand the behaviour of the target in the least intrusive way possible.
  - **Application:** Tracing is useful for understanding the flow of program execution, identifying any anomalies or unwanted behaviour, and collecting statistical data. For instance, using a tool like strace on a Unix-like system allows one to trace system calls made by a process, helping to diagnose issues such as file access problems or network communication errors.

Case Study

Analyzing the firmware binary in IoT devices is recognized as a critical practice in ensuring the security and reliability of such devices, given their growing integration into various aspects of daily life [57, 58, 49, 59, 53]. The analyzed case studies indicate how track analysis not only reveals hidden vulnerabilities but also contributes to improving the understanding of the environment in which these devices operate. Through the use of advanced methodologies, such as the attributed control flow graph (ACFG) combined with graph-embedding algorithms and deep neural networks [57], or hybrid analysis that combines machine learning techniques with dynamic binary analysis [58], it is possible to identify potential threats and ensure timely remediation. The challenges of patching IoT devices are addressed with innovative approaches, such as PATCHECKO, a framework that leverages hybrid analysis based on machine learning and dynamic analysis to check binary executables for patches [58]. Dynamic solutions such as IoTCID, which uses constrained models and feedback-based fuzzing, demonstrate effectiveness in detecting command injection vulnerabilities [49]. Furthermore, the need to address specific challenges related to direct execution on IoT devices has led to the development of tools such as AflIot, an on-device fuzzing framework that preserves peripheral compatibility [59]. Finally, projects like PROLEPSIS focus on analytics optimized for IoT platforms, identifying executable points at risk of control-flow hijacking

and applying targeted monitoring techniques [60]. These case studies highlight how binary analysis not only identifies vulnerabilities but extends to verifying the effectiveness of patches and mitigating threats, thus strengthening the overall security of IoT devices.

2.4 Final Remarks

This first chapter explored in detail the complex and interconnected world of Internet of Things (IoT) devices, focusing on two fundamental aspects: firmware development and related security issues. The discussion on Continuous Integration (CI) underscored its pivotal role in enhancing firmware development processes, aiding in maintaining code consistency, promptly identifying errors, and streamlining regular updates. Analysis of the firmware binary revealed its critical importance, highlighting the need to understand the internal structure and identify potential vulnerabilities that could compromise the security of the entire IoT system. These considerations are crucial in ensuring greater reliability and security of the software implemented in IoT devices.

Chapter 3

A tool for Continuous Integration

3.1 Introduction

Continuous integration (CI) is a software development methodology focused on enhancing efficiency and quality across the development lifecycle. CI entails the frequent and automated integration of code changes into a shared repository, as discussed in the 2.3.1 section. This practice ensures that any modification made by a contributor is merged with the existing codebase and subjected to a battery of automated tests to ascertain the overall integrity and functionality of the system.

3.1.1 CI benefits

The adoption of continuous integration yields some properties for software development teams:

- **Risk Reduction:** The regular integration of code changes coupled with automated testing aids in reducing identification time and resolving compatibility and integration issues, thereby mitigating the risk of software errors.
- **Enhancement of Code Quality:** Continuous integration fosters the cultivation of cleaner and more modular codebases, as team members are incentivized to integrate their modifications frequently and uphold a stringent standard of quality.
- **Automation:** Continuous integration practices allow organizations to release software by integrating and rigorously testing changes in an automated manner.

- **Feedback:** Automated testing mechanisms identify errors and anomalies, giving contributors feedback on their contributions to enable iteration and refinement.

One instrumental tool employed to streamline the practice of continuous integration is ci-generate. This tool automates the intricate processes of code generation and dependency management within a software project. In essence, ci-generate empowers development teams to economize on time and ensure the perpetuation of source code consistency by automating repetitive tasks, such as the creation of new components or the updating of project dependencies.

In summation, Continuous Integration stands as an indispensable practice in the realm of software development, fostering heightened efficiency, bolstered quality, and accelerated delivery cycles. The utilization of tools like ci-generate further amplifies the efficacy of continuous integration workflows, thereby facilitating the creation of robust and reliable software solutions.

3.2 ci-generate tool

In the context of software development, it is crucial to proactively identify and resolve any issues present in both the source code and binary files. The establishment of an efficient workflow for continuous integration, supported by specialized tools capable of identifying specific problems during the coding and binary code production phases, becomes an essential practice.

This workflow represents a synergy of different aspects of software maintainability and security, with particular attention to static analysis[1]. This crucial process carefully examines the source code for defects and maintainability issues, providing crucial information to improve not only the overall quality of the code but also the security itself.

Manually developing this process, however, can be laborious, distracting developers from their primary coding responsibility. Simplifying and automating these essential controls is the main function of the tool in question: ci-generate.

[1]In computer science, static program analysis (or static analysis) is the analysis of computer programs performed without executing them, in contrast with dynamic program analysis, which is performed on programs during their execution.

3.2.1 Rust language

In software development, the choice of programming language is of enormous importance, influencing both the functionality and security of the final product. **Rust**[2], a relatively new language, has emerged as an attractive option due to its unique blend of security and performance.

- **Prioritize safety:** One of Rust's defining features is its focus on security. Unlike many other programming languages, Rust is designed from the ground up to prevent common mistakes that can lead to security vulnerabilities. How is this achieved? Rust's type system and ownership model work together to ensure memory safety, eliminating risks like buffer overflows and null pointer dereferences that affect other languages, such as C/C++.
  Rust keeps an eye on how memory is used in a program, preventing the kinds of errors that hackers often exploit. This proactive approach to security sets Rust apart and makes it an ideal choice for projects where security is paramount, like ci-generate.

- **Provide high performance:** But Rust does not just stop at security; it also focuses on performance. Conventional wisdom might suggest that prioritizing security would come at the expense of speed, but Rust achieves both goals through its innovative approach to concurrency and resource management.
  In many programming languages, handling multiple tasks at the same time (concurrency) can be complex and error-prone. Rust's ownership model simplifies this process, allowing developers to confidently write concurrent code. Additionally, Rust's zero-cost abstractions ensure that high-level code translates into efficient machine code, maximizing performance without sacrificing security.

- **Differences with other languages:** When comparing Rust to more established languages like C or C++, the differences become even more pronounced. While C and C++ offer a high degree of control over system resources, they also expose developers to several potential vulnerabilities, especially when it comes to memory management. Rust, on the other hand, provides similar control but with built-in protections to prevent common pitfalls.

By leveraging Rust's innovative features, ci-generate ensures the reliability and security of its code base and aims for good performance, demonstrating a commitment to excellence in software engineering.

[2]Rust is a multi-paradigm, general-purpose programming language that emphasizes performance, type safety, and concurrency.

3.2.2 Features and purpose

Leveraging the power of templates and Continuous Integration (CI) [61] through GitHub Actions[3], ci-generate simplifies the intricate project creation process by producing a range of must-have files, ranging from source code to build systems, YAML files, shells and Dockerfiles. Crucially, this tool dynamically integrates information provided by developers at runtime, completing project templates.

- **Versatile support for projects and configurations:** ci-generate boasts support for a wide range of project types, each closely associated with a core programming language, specific build tools and package managers. The adaptability of its continuous integration system, orchestrated through GitHub Actions, enables the orchestration of workflows perfectly tailored to the unique requirements of individual projects. These workflows are developed to reduce developer wait times and accelerate error detection through in-depth analysis and testing phases.
- **Toolchain and API integration overview:** ci-generate takes a structured approach to its toolchain, adapting to various build systems and programming languages, such as Python, Rust, Java, etc. Additionally, the availability of APIs enables developers to conveniently generate and tailor projects by inputting project-specific details like name, license, and path. This systematic approach ensures a streamlined and efficient setup process for users. Figure 3.1 provides a visual representation of a directory containing a generated project example, with its build configuration files.
- **Purpose and contextual meaning:** The primary objective of ci-generate is to reduce the cognitive load involved in setting up GitHub Actions projects and workflows. This refers to the mental effort developers need to manually configure these operations. Its ability to streamline project startup or enhance existing ones with the integration of GitHub Actions highlights its role in modern software development practices, allowing developers to focus their attention on core development efforts.

Thus, ci-generate offers a complete solution for project initialization and continuous integration configuration. By automating mundane tasks and providing flexible infrastructure, ci-generate enables developers to accelerate project setup and simplify workflow integration. Its integration with GitHub Actions ensures a cohesive and efficient development experience, affirming its status in modern software engineering efforts.

[3]GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that enables automation of the build, test, and deployment pipeline.

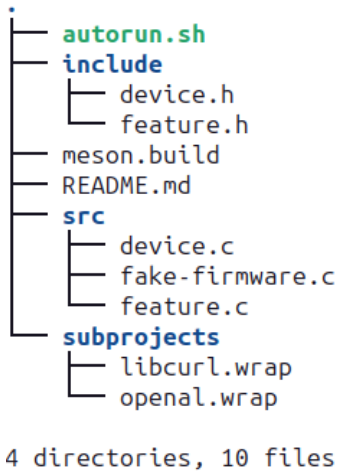

Figure 3.1: libdevice-C project with its build configuration files

3.2.3 Workflow

The CI workflow of the ci-generate tool is designed to automate and orchestrate the project development process, ensuring that every code change is verified and integrated into the repository consistently and reliably.

The workflow is structured into different levels, each of which performs specific tasks within the software development process. The levels are:

- Legal and Format Layer
- Builds and Docs Layer
- Code Coverage Layer
- Dependency Layer
- Unsafe Checks Level

The initial steps involve checking for legal compliance and formatting using tools such as Clippy[4] and Rustfmt[5]. Clippy analyzes the code, offering suggestions for common errors, while Rustfmt formats the code according to style guidelines. Subsequently, static code analysis is performed to detect potential quality and security issues, employing the rust-code-analysis action. This is followed by the steps of compiling the project and generating documentation, ensuring that the

[4]Clippy documentation

[5]GitHub repository

code is compileable and well-documented across different platforms.

The workflow proceeds with the computation of code coverage by executing tests, employing grcov to gather and consolidate code coverage data from multiple source files. This information is then uploaded to Codecov, providing insight into which lines of code were executed by the test suite. Additionally, advanced security checks and code analysis are conducted to identify potential vulnerabilities, utilizing tools like Valgrind and Address Sanitizer. Valgrind is employed for memory debugging, memory leak detection, and profiling, while Address Sanitizer detects bugs manifested as undefined or suspicious behaviour.

To sum up, the CI workflow implemented with GitHub Actions for the ci-generate tool offers a robust automation and verification infrastructure, ensuring code quality, security and compliance at every stage of the development process. Due to GitHub Actions, all this happens automatically and transparently, allowing developers to focus on software development without worrying about manual and repetitive processes.

3.3 Code refactoring

In the evolution of any software project, the refactoring process stands out as a fundamental phase that transcends mere code restructuring. It represents a strategic revisiting of the code base, not only to fix bugs or improve performance but to substantially improve its architecture, usability and maintainability. In the case of the ci-generate tool, refactoring serves as a transformation path to improve its capabilities and developer experience.

Refactoring in software development is the process of restructuring existing code without changing its external behaviour, aiming to enhance the design, structure, and implementation of the software while maintaining its functionality. It's not just about cleaning up the code; it is a conscious effort to make the codebase more readable, adaptable, and scalable. The process involves optimizing existing structures, improving interfaces, and introducing new abstractions to improve the overall quality of the software.

3.3.1 Objectives

The refactoring initiative for the ci-generate tool has been driven by a series of multi-faceted goals aimed at making it more robust and easier to use:

1. **Modular architecture for intuitive use:** Library includes various modules, such as error handling, command execution, and filters used in templates. By maintaining a modular architecture, the refactoring initiative preserves the logical organization of features, making it easier for developers to navigate and understand the structure. This modular approach promotes code maintainability and allows developers to selectively use or extend specific components based on project requirements.

2. **Efficient and clear execution flow:** Restructuring efforts prioritize optimizing the execution flow, making it more efficient and transparent. The library's refined interface ensures that developers can seamlessly integrate CI configurations and project templates into their workflows without unnecessary complexity. The clearer execution flow contributes to a more consistent development process and facilitates collaboration between team members working on different aspects of a project.

3. **Improving Error Handling:** In the process of renovating the ci-generate library, one of the key areas of focus was refining error handling. This section explores not only the technical changes made but also the broader meaning of custom error handling in the context of a Rust project.

   - **Understand the role of error handling:** Effective error handling is a critical aspect of reliable software development. Custom error handling allows developers to express and handle errors to align with specific library requirements.

   - **Meaning of custom error definitions:** The library incorporates a dedicated error module. This deliberate choice allows for the definition of custom error types, as shown in the code 3.1), each of which conveys specific details about potential problems that may occur when running the library. These custom errors provide descriptive insights into the nature of the errors, helping developers with rapid diagnostics and resolutions.

Listing 3.1: Custom code to handle directory not found error.

```
1         /// Unable to retrieve the home directory
2         #[error("Home directory failure")]
3         HomeDir,
4
```

- **Using the *Result⟨T⟩* type:** Central to Rust's error handling paradigm is the Result⟨T⟩type. The Result⟨T⟩type explicitly communicates the possibility of errors in specific operations, providing a clear and structured way to handle potential errors. By using Result⟨T⟩, the library signals developers that certain operations may cause errors, prompting them to consider and address these scenarios in their code.
- **Incorporation of external errors:** A notable aspect of the error-handling strategy is the inclusion of errors from other libraries. This practice enriches the error structure with specific information, contributing to more informed diagnostics and simplified troubleshooting. By encompassing a broader spectrum of potential failures, the library provides a comprehensive and detailed view of potential failure scenarios.
- **Advantages of specialized error handling:** The error-handling approach not only improves the readability and maintainability of the code but also provides developers with significant information about potential pitfalls. The use of specialized error types and the Result⟨T⟩type promotes a structured and disciplined coding style, in line with Rust's overall goal of building reliable and secure software.

4. **Increased flexibility with TemplateData:** Before delving into the TemplateData implementation, it is important to understand two fundamental concepts in Rust: borrow and owned.
   - **Borrow:** It refers to the temporary acquisition of a reference to a value. It grants access to a value without assuming ownership, enabling passing references to data without transferring ownership of the data itself.
   - **Owned:** It refers to complete possession of a value. When a variable owns a value, it manages its memory, including modifying the value or deallocating it when necessary. Ownership grants full control over the value's lifetime and memory usage.

   TemplateData is a strategic addition to the library, representing a **data structure** that consolidates essential parameters for project and CI configuration. This includes information such as the project license, git branch, project name, and project path. The structure was designed to accept various data, providing developers with a versatile means of passing parameters.

- **Importance of Cow⟨'a, str⟩:** At the heart of the improvement is the adoption of Cow⟨'a, str⟩for specific fields within TemplateData. This choice makes sense in the context of Rust's ownership model. Unlike String, which implies ownership and requires memory allocation, Cow⟨'a, str⟩(short for Clone On Write[6]) provides a flexible approach. Allows the structure to handle both references (&str) and owned data (String) without incurring unnecessary memory allocation costs.
- **Handling &str and String:** In Rust, &str represents a portion of a string, a reference to a sequence of UTF-8 bytes, while String is an expandable string, allocated on the heap, where the objects dynamically allocated during program execution are stored. The distinction is fundamental for efficient memory management. By allowing TemplateData to accept both types via Cow⟨'a, str⟩, the library gains the ability to accommodate different inputs seamlessly. This is especially beneficial when handling scenarios where the overhead of memory allocation for String is undesirable.
- **Implementation Details:** The TemplateData structure comes with methods like license, branch and name, each of which takes input that implements Into⟨Cow⟨'a, str⟩⟩. This allows developers to provide a reference or property string, demonstrating the flexibility of the library to adapt to the developer's preferred data representation.

In essence, the introduction of TemplateData not only simplifies the interface for developers but also addresses the complex challenge of efficiently managing different string representations. The refactoring process emphasizes the importance of adaptability and resource-aware design, ensuring that the ci-generate library can integrate seamlessly into a variety of project contexts.

5. **ProjectOutput Structure:** One of the fundamental architectural refinements introduced during the refactoring process is the integration of the ProjectOutput structure. This addition plays a pivotal role in the enhancement of clarity, separation of concerns, and the extensibility of the ci-generate library.
   - **Clarity and Separation of Concerns:** The ProjectOutput structure acts as a container for the output of the definition of a project model, ensuring that it is encapsulated in a defined, and appropriately organized structure given the presence of different elements that characterize it.

[6]Cow can encapsulate and offer immutable access to borrowed data, cloning the data only when mutation or ownership is needed.

- **Trait in Rust. Defining BuildTemplate for Extensibility:** In Rust, a trait [62] is a fundamental concept, akin to an interface in other programming languages. It defines a set of methods that a type must implement if it adopts the trait. Specifically, in the context of ci-generate, the BuildTemplate trait outlines the methods required for building a template. Traits are essential for creating flexible and reusable code components, allowing types to conform to a shared interface.
- **Understanding the BuildTemplate Trait:** The BuildTemplate trait, therefore, acts as a contract that any template builder within the ci-generate library must adhere to. It declares methods responsible for defining how templates are constructed, providing a consistent interface for different template implementations. By adopting the trait, various components within the library can interact seamlessly, ensuring that any changes or additions to template-related functionalities will not disrupt the overarching structure.
- **Purpose of ProjectOutput within BuildTemplate:** The ProjectOutput structure is intricately linked to the BuildTemplate trait. It represents the standardized output that any template adhering to the trait should produce. This clear definition of what constitutes the output of a template enhances the readability and flexibility of the code. So, the ProjectOutput structure acts as a concrete manifestation of the abstract template-building process outlined by the BuildTemplate trait. In essence, the ProjectOutput structure, in tandem with the BuildTemplate trait, embodies the principle of separation of concerns and lays the groundwork for a flexible and extensible template-building framework within the ci-generate library.

To sum up, the refactoring is a deliberate and strategic effort to fortify the tool, ensuring it aligns with the best practices of software development while providing a foundation for future enhancements. The benefits extend beyond the confines of the codebase, positively impacting the entire development workflow. As the specific changes made are examined, the depth and thoughtfulness of the refactoring process will become more apparent.

3.3.2 Path validation function

In the context of the ci-generate tool, the path validation function is a crucial component for ensuring the validity and consistency of project paths, being a mandatory parameter to be included. The various features of the function will be reviewed, with special emphasis on the rationale for the implementation choices.

- **Accept directory only:** The path_validation function is designed to accept

only directory paths and reject file paths. This choice is motivated by the fact that the tool primarily handles the creation of projects organized in directories. Avoiding the acceptance of file paths is critical since the generated project must be placed in a directory.

- **Verification of UTF-8 character correctness:** Verifying the correctness of UTF-8 characters is essential to avoid encoding problems at runtime. If a path contains invalid UTF-8 characters, unexpected errors may occur during path manipulation. Ensuring correct encoding contributes to greater stability and predictability of the tool.

- **Handling of the Special Case of the Point:** It is important to resolve the special case where the path is a dot, as users can enter '.' to indicate creation of the project in the current directory. Handling this case is important to prevent ambiguity and ensure that the tool behaves according to user expectations.

- **Adaptability to Different Home Director Prefixes:** The function is designed with a wide adaptability to handle the different prefixes used in home directors on various platforms. This is especially relevant when considering the tilde symbol (~), which can be interpreted differently on Unix and Windows systems. On Unix systems, the tilde (~) is commonly used to represent the current user's home directory. For example, if the user is user1, the path ~/Desktop will translate to /home/user1/Desktop. In this case, the function is designed to recognize the tilde as the home directory reference symbol and construct the full path accordingly. On Windows systems, the tilde character may not be directly recognized as representing the user's home directory. In the past, Windows systems used the tilde to summarize long file or directory names[7]. For example, the name TextFile.Mine.txt might be abbreviated to TEXTFI 1.TXT. Therefore, the tilde does not have a specific function as in Unix systems, creating potential ambiguities.
  In this context, the function must be able to interpret the tilde and convert it to the correct path. The flexibility of the function allows it to handle this diversity in home director prefixes between operating systems.

- **Path canonicalization:** Path canonicalization is a key step in defining a complete and correct path. In practical terms, canonicalization resolves any relative paths and normalizes them. For example, if the path contains references to parent directories (".."), canonicalization will replace them with the names of the corresponding directories, thus obtaining the correct absolute path. Next, the function takes care of the creation of any missing directories,

[7] 8.3 filename notation

ensuring that the project structure is cohesive and complete.

Different approach

The different approach to the implementation of the path_validation function in the context of the ci-generate tool reflects a process of continuous improvement aimed at simplifying the code, improving maintainability and ensuring a greater consistency between different implementations intended for different operating systems. This section will examine the modifications to the functionality, highlighting the underlying reasons driving these changes.

- **Code unification**
  One of the major changes was to unify the code for Windows and Unix/MacOS operating systems into a single function. This approach eliminated code duplication and made it much easier to manage differences between platforms. Additionally, this unification made the code more consistent and easier to understand, contributing to better maintainability in the long term.
- **Use of cross-platform libraries**
  Another key point was using the home library to get the home directory on all platforms. This change greatly simplified home directory management, eliminating the need to use different libraries for different operating systems. It also made the code more readable and contributed to greater consistency across implementations.
- **UTF-8 encoding**
  Simplifying the handling of UTF-8 encoding was another significant change. In the previous version, there were additional checks to verify the correctness of the UTF-8 encoding, while in the final version a simpler condition was used for this purpose. This helped reduce the complexity of the code and make it clearer and more concise.

In summary, the changes made to the path validation feature reflect an ongoing effort to improve code quality and ensure a better development experience. Adopting simpler and more uniform approaches has contributed to greater coherence, readability and maintainability of the code, preparing it for any future updates and ensuring greater robustness in addressing challenges that may arise during software development and maintenance.

3.3.3 Cargo feature

Before this approach, ci-generate could only create a cargo project with files necessary for continuous integration. However, with the integration of a new

method following the cargo init command flow, it is possible to get a complete Cargo project. This approach enables them to create both a library and an executable. The functionality is accessible through the command line or the library API.

Change of Perspective

Originally, the command evolution was centred around using cargo new to create new Rust projects. However, a shift towards the more flexible cargo init command was later embraced. This command allows initializing a Rust project within an existing directory, facilitating integration into contexts with predefined project structures. This flexibility proves beneficial for embedding Cargo build templates into contexts with existing CI configuration files and vice versa.

Generation of library, binary, and CI workflow

The key component in implementing Cargo project generation is the cargo.rs file. It defines a pre-defined project structure, including essential configuration files for CI, Docker, and testing. The file introduces two options for the Cargo project structure: creating projects for libraries or executables, both with Continuous Integration files included. A check is implemented to prevent the simultaneous creation of both types of projects, ensuring clarity and coherence in the development process. Additionally, the inclusion of proptest.rs in the Cargo project marks an evolution towards best practices and refined approaches for code validation.

Final considerations

The cargo.rs file is a central component in advancing Cargo project generation within ci-generate. It provides a predefined project structure adaptable to various project types. Its flexibility allows developers to tailor project generation to specific needs, making it a key contributor to the creation of sophisticated Cargo projects aligned with current quality standards and development methodologies.

3.4 Tests implementation

Test suite, characterized by unit tests and integration tests, constitutes a pivotal element in the software development lifecycle. Within the established paradigm of software engineering, validation and verification emerge as essential, undertakings crucial for the progression of the development process [63].
The early identification and rectification of errors through systematic testing significantly contribute to efficient error management and heightened code maintainability.

This proactive approach mitigates the accumulation of defects, thereby streamlining subsequent maintenance activities and augmenting the longevity of the software. Moreover, the methodical execution of the test suite functions as a linchpin for ensuring the overarching consistency and reliability of the entire system. By affirming the accurate execution of functions within each module and meticulously validating interactions between modules, the robustness and coherence of the application are assured.

Furthermore, the strategic implementation of testing in the nascent stages of development yields profound cost reductions. This preemptive measure facilitates the timely detection and correction of defects, mitigating the expenses associated with addressing more intricate and critical errors later in the developmental cycle. Consequently, this practice contributes substantively to the optimization of resources deployed throughout the process.

Code Coverage

Code coverage, measured as the percentage of code covered by a test suite, is a critical metric. It is computed using profiling instruments that detect executed code lines. Code coverage is associated with a traffic-light system model:

- **Red:** Code coverage below 60%, indicating insufficient coverage, leading to a necessary workflow stop.
- **Orange:** Code coverage between 60% and 80%, considered acceptable but not desirable.
- **Green:** Code coverage above 80%, signifying comprehensive test coverage.

This general approach aligns with the concept of code certification required, ensuring reliability, security, and well-tested code. The adoption of thresholds as an integral part of the workflow execution adds an active role to the code coverage percentage.

In essence, the incorporation of unit and integration tests, along with code coverage analysis, reflects a commitment to quality assurance, minimizing defects, and enhancing the overall stability and maintainability of the ci-generate tool. With the addition of the tests explained in the following sections, the coverage achieved for the tool is 98%.

3.4.1 Unit tests

As part of software testing, the unit tests for the path_validation function in the ci-generate library examine the behaviour of this critical component. Unit tests are designed to reproduce specific scenarios, revealing nuances in the function's

behaviour. The tests created for the path_validation feature are no exception, covering a spectrum of situations to ensure the works under different conditions.

- **File path validation:** The unit test test_invalid_path_file assesses the behaviour of the path validation function when provided with a file path. This test validates that the function correctly identifies and rejects file paths. This verification reinforces the function's role in enforcing strict path validation policies, critical for maintaining the integrity of project structures.

- **Directory path validation:** Conversely, the test_valid_path_folder test evaluates the function's response to valid directory paths. By providing a valid directory path, this test verifies that the function accepts such paths without errors. This validation is essential in ensuring that the function behaves as expected when presented with typical directory paths.

- **Current directory shortcut:** The unit test test_current_path examines the behaviour of the function when presented with the special case of the current directory shortcut ('.'). This test verifies that the function appropriately handles this scenario by returning a successful result without any errors. By validating this aspect, the test confirms the function's ability to interpret and process directory shortcuts accurately, contributing to its overall robustness and versatility across various input scenarios.

- **Windows-specific home directory:** The test_invalid_home_directory test evaluates the function's behaviour concerning the representation of the home directory on Windows systems. By providing a path with a tilde ('~') prefix, typical in Unix systems but potentially ambiguous on Windows, this test ensures that the function correctly identifies and manage such cases. Specifically, it validates that the function appropriately raises an error when encountering an invalid home directory representation on Windows, thus ensuring consistent behaviour across different operating environments.

- **UTF-8 character encoding:** The test_invalid_utf8_path test scrutinizes the function's capability to manage paths containing invalid UTF-8 characters. By providing a path with malformed UTF-8 encoding, this test verifies that the function accurately detects and rejects such paths, preventing potential runtime errors stemming from incorrect character encoding. This validation underscores the function's robustness in maintaining data integrity and reliability, particularly in diverse language environments where character encoding may vary.

Test design pros and cons

Achieving a balanced approach between specificity and generality in testing is essential for effective quality assurance. While comprehensive tests are necessary to identify potential weaknesses in software, an overly detailed test suite may become unmanageable and impede development progress. Therefore, the right balance ensures that tests remain a valuable resource, providing meaningful feedback to enhance software quality without causing undue burden on the development process.

3.4.2 Property testing

When it comes to validating the main APIs in the ci-generate library, such as define_license, the approach taken leans towards using the proptest[8] library. This decision is motivated by the goal of conducting a comprehensive analysis of the inputs of these functions, ensuring their correctness and robustness through property testing.

Property testing allows developers to define fundamental software properties as declarative rules. The library's automatic input generator produces random data to test specified properties. This methodology differs from the traditional unit tests as it explores a large input space, revealing unexpected behaviour and potential errors that might elude a conventional test suite. Applying proptest to these functions aligns with the need to carefully examine various usage scenarios, including inputs that go beyond standard use cases. This approach is particularly effective at identifying critical limitations and potential edge case errors and maintaining the overall consistency and reliability of the analyzed APIs.

define_license proptest

- **Overview:** The define_license function manages the definition of the license associated with a project. It takes a string representing the license as an input parameter and returns an object. The function ensures the correct management of project license information, providing a robust mechanism for defining and validating this information.
- **Test operation:** This property test explores and tests the behaviour of the define_license function in response to various license-related inputs. The test defines a static array, containing some default valid licenses. A LicenseTest

[8]Proptest is a property testing framework. This testing method enables specific property verifications within the code for arbitrary inputs. In failure, it automatically identifies the minimal test case necessary to reproduce the issue.

structure is defined with a field of type string, used as an input parameter for the test. The test block contains assertions for different scenarios where the define_license function might be called, covering cases where the license string is empty, valid, or invalid.

These property tests, supported by proptest, play a critical role in ensuring the correctness and robustness of the function. By covering a wide range of input cases, they contribute significantly to the overall quality of the ci-generate tool.

Observations

Through unit tests, various scenarios were carefully examined to evaluate function behaviour. Each test, designed to address specific situations, ensures the correctness and reliability of the implementation, providing a complete overview of the feature capabilities.

The adoption of the proptest library for property testing reflects a shift towards a broader testing strategy aimed at conducting comprehensive analyses of inputs to APIs such as define_license. These tests allow the declarative definition of fundamental properties, exploring a large input space and revealing unexpected behaviour and potential errors that conventional test suites might miss.

In summary, unit and property testing set the stage for a more comprehensive evaluation of software behaviour in integrated scenarios. The subsequent transition to integration testing marks a crucial phase in validation, significantly contributing to the overall quality and robustness of the module under consideration.

3.4.3 Integration tests

In this context, integration tests take on a fundamental role, as they are dedicated to ensuring the correct interoperability of all the different components of the system, ensuring the efficient management of various scenarios and inputs. Unlike unit tests, which focus on validating individual units of code, integration tests explore the joint behaviour of multiple units, evaluating the cooperation and cohesion between them. This section will explore the implementation of such tests, delimiting the evolutionary path into two distinctive phases.

First Phase: Test with Fixed Output

In the initial phase of the implementation of integration tests, attention was focused on defining specific scenarios in which a known input gave rise to a predefined output. To illustrate this approach, the practical process of creating a Cargo project within the context of ci-generate can be considered.

During this phase, the integration tests orchestrated the creation of a Cargo project structure through the use of library functions, using predefined data (i.e. license, name and branch defined by default). This procedure led to the generation of a set of files intended for project configuration and continuous integration.
Subsequently, through the implementation of a specific function that allows the use of the sha2 library, the computation of the hexadecimal string representing the SHA-256 hash of each file within the aforementioned folder was carried out.

The verification process consisted of comparing the hexadecimal string of each generated file with the corresponding SHA-256 hash previously saved in an array containing all expected hashes. However, this approach was static as the analysis was limited to fixed versions of the same project. Creating a new project with different parameters would have resulted in the generation of a divergent hash since the new hash produced would have had no correspondence with the expected hash previously recorded in the array of expected hashes.

This static nature not only characterized the system but, through a simple comparison of SHA-256 strings, made it impossible to discern specific differences between files. As a result, the necessity to enhance integration tests to be dynamic and adaptable has influenced the evolution of the testing methodology.

Second Phase: Adoption of Insta Tests

The second phase of the testing process represents a significant evolution compared to the initial approach, abandoning the previously described solution in favour of Insta[9]: a library for snapshot testing in Rust. Insta tests stand out for their ability to detect content-level differences between two versions of the same file.
Inside the project, the folder called tests (figure 3.2) hosts the .rs files dedicated to each generatable project (yarn, poetry, meson, maven and cargo). Each .rs file contains the code relating to the definition of specific integration tests, aimed at verifying distinct aspects of the project generation. In parallel, common folder contains mod.rs file, designed to group parts of code used by all projects in the comparison process. Another directory named repositories/snapshot is specifically designated for storing the snapshots generated from the comparison operation, preserving the original path provided by the user for each project.

The core of these integration tests is represented by the compare_template function. Its main responsibility is to compare the snapshots dynamically generated during the test execution with those stored previously. The primary objective of this

[9]Insta is a snapshot testing tool for Rust. Snapshot tests are tests that assert values against a reference value (the snapshot).

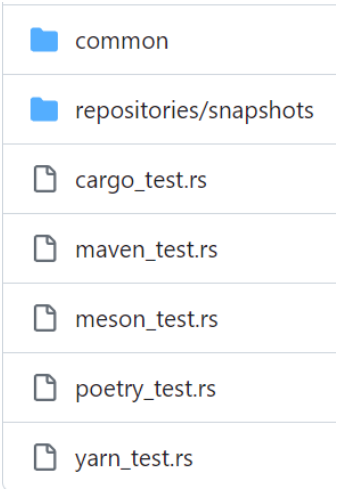

Figure 3.2: Contents of the tests folder

function is to ensure consistency between the expected results and those obtained during the testing process.
To understand how compare_template works, the function inspects the files within each directory of the specified path. It extracts the contents of each file and compares it with the contents of the corresponding snapshots, which represent stored versions of the files in repositories/snapshot folder.
The comparison between contents is done through the use of the assert_snapshot! assertion. This assertion allows to make detailed comparisons between the current contents of the file and the expected version, defined in the snapshots. Ultimately, the compare_template function checks whether the contents of the files during the test execution coincide exactly with those expected in the snapshots, thus ensuring the integrity of the tested system.

This process leads to a comparative analysis between a new version of the snapshot, dynamically generated during the current test execution, and the previous instance, which represents the original, accepted version of the snapshot. Insta's setup is tailored appropriately to customize the storage location of the snapshots, providing a structured organization. This adaptation constitutes a crucial element in ensuring precise and orderly management of snapshots, facilitating the process of reviewing and comparing the different recorded versions (see Figure 3.3).

3.4.4 Observations

Implementing a comprehensive testing strategy for ci-generate has brought significant benefits to the quality and reliability of the software. Unit tests, focused on

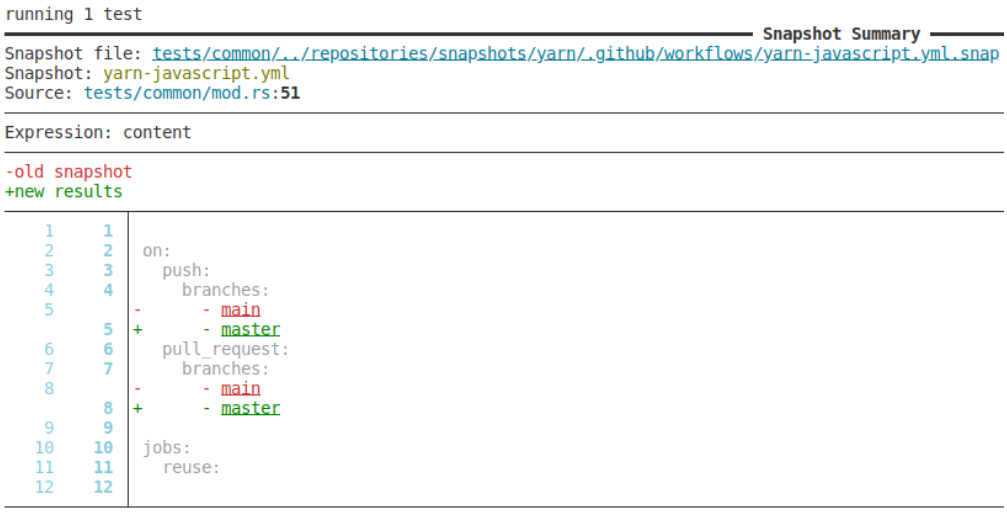

Figure 3.3: Comparison between different recorded versions of Yarn

individual features, provided a solid foundation, identifying and fixing potential errors in specific scenarios. The precision of these tests ensured that each critical part of the code functions as intended, establishing fundamental confidence in the correctness of the individual components.

The introduction of property tests with proptest has broadened this coverage, allowing for deeper analysis of inputs provided to APIs, such as define_license. This methodology led to the discovery of unexpected behaviours and critical constraints, providing a more complete understanding of the capabilities and limitations of the tested functions.

The integration tests represented a crucial step, especially with the adoption of snapshot tests via the Insta library. This evolution not only revealed content-level differences between file versions but also introduced flexibility in managing projects with different parameters. The ability to dynamically recognize changes made the testing process more adaptable and allowed better management of future software evolutions.

Overall, the phased and complementary approach of unit, property, and integration testing was a major contributor to ensuring the consistency, reliability, and overall quality of the ci-generate tool. This testing strategy not only identified and corrected errors but also provided a solid foundation for future development, improving the maintainability and sustainability of the software under diverse usage scenarios.

3.5 Final Remarks

The second chapter of this work comprehensively outlined the various changes of ci-generate tool library, focusing on the restructuring of its architecture and the implementation of new features. The incorporation of new data structures, error handling mechanisms, and the implementation of initialization functionality for Cargo projects has significantly enhanced the overall organization of the code, interface clarity, and flexibility of the entire library. Furthermore, a comprehensive test suite has been established to guarantee the dependability and accuracy of the library's functionality.

Chapter 4

A dummy firmware for IoT

4.1 Introduction

In the era of the Internet of Things (IoT), where connectivity and digital integration increasingly permeate both our domestic and professional environments (as discussed in 2.2.2), IoT devices play a crucial role in enhancing the efficiency, security, and comfort of our lives. At the heart of these devices lies their **firmware** (see Section 2.3.2), which acts as a focal point regulating and coordinating, the various functions and interactions among the system's components.

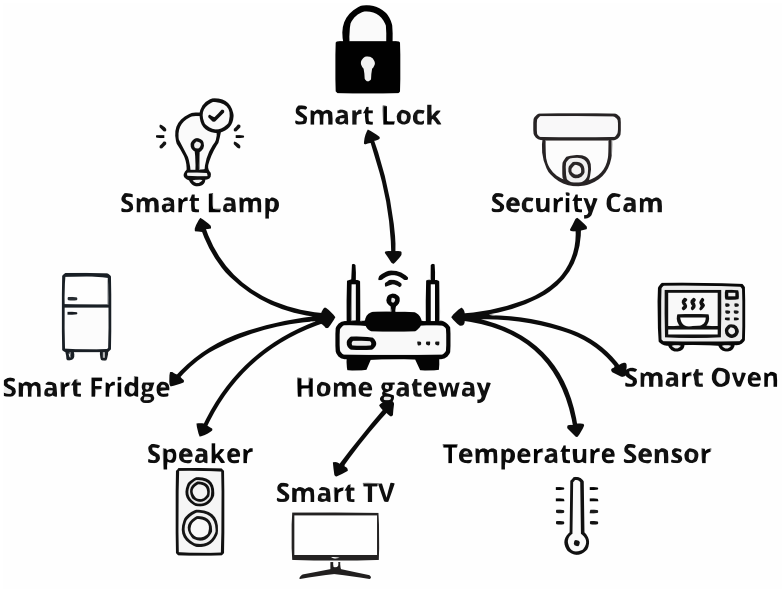

Figure 4.1: A smart home IoT architecture system.

As shown in Figure 4.1 in a typical home environment, the home gateway often acts as the central hub of the home network, managing several actuators scattered throughout the premises. These actuators include, among other components, smart lamps, audio/video recording devices such as security cameras and speakers, as well as temperature sensors. Each actuator requires dedicated firmware to govern behaviour and interaction with the rest of the system to perform its functions.

4.1.1 Purpose of dummy firmware

In the context of IoT environments, the accuracy and reliability of the firmware implementation are fundamental elements to ensure the correct functioning and interoperability of IoT devices. However, firmware development and testing can be complex and resource-intensive, often requiring access to physical devices for experimentation and functionality verification.
A **dummy firmware-device**[1] implementation has been devised to address these challenges and advance research in this field. Within this implementation, several library variants named **libdevice** have been created. From a general standpoint, this library essentially comprises APIs that simulate the behaviour of IoT firmware. Therefore, the concept of **dummy firmware** refers to the different binaries or versions of firmware that emerge from compiling the variants of the libdevice library within the dummy-firmware-device project. These binaries are termed *dummy firmware* because they are simulated and do not correspond to real firmware intended for actual IoT devices. However, they are useful for analysis and evaluation during development, as in the case of the **manifest-producer** tool in Chapter 5.

To develop the dummy firmware, we opted for the C, C++, and Rust programming languages. Each language has been chosen based on its specific advantages and suitability for computer simulation firmware.

- **C** has been selected for its low-level functionalities, making it ideal for programming embedded devices. It offers direct control over hardware and memory management.
- **Rust** has been chosen for its robustness and safety features. Its advanced memory management mechanisms and static type system help prevent common errors during programming, ensuring more reliable code. See Section 3.2.1 for further information.
- **C++** provides a flexible and balanced approach compared to C and Rust. It supports both imperative and object-oriented programming paradigms,

[1]GitHub repository: https://github.com/SoftengPoliTo/dummy-firmware-device

allowing for more complex and modular code then C, without the added complexity of concepts introduced in Rust (for example borrow and owned seen in section 4).

Each programming language contributes to developing variants of the libdevice used in generating different dummy firmware.

4.1.2 Importance of different programming languages

The importance of developing this firmware in three different programming languages - C, C++, and Rust - lies in the need to comparatively evaluate the performance and characteristics of these languages within the Internet of Things (IoT) domain. Given the complexity and diversity of IoT applications, it is essential to identify the most suitable programming language for developing efficient, secure, and reliable firmware.

- Firstly, the choice of programming language can significantly impact firmware **performance** in terms of execution speed, resource consumption, and system responsiveness. Each language has its characteristics in terms of memory management, code optimization, and low-level operations support which can directly affect the performance of a device. So, developing firmware in different languages allows for an empirical comparison of each language's performance and the identification of any advantages or limitations that may influence design and implementation choices.
- Secondly, **security** is a critical factor in an IoT domain (a reference to the Section 2.2.2), considering the growing concern for data protection and the vulnerability of connected devices. Programming languages differ in their ability to prevent common vulnerabilities such as buffer overflow, code injection and other security threats. Developing firmware written in C, C++, and Rust allows the evaluation of each language's robustness and resilience to attacks, providing valuable insights to enhance the security of IoT devices.
- Lastly, code **scalability** and **maintainability** are crucial factors in IoT firmware development, considering the complexity and the rapid evolution of applications and technologies. Each language offers different features and programming paradigms that can influence ease of development, code modularity, and long-term maintainability.

4.2 Design and Structure

The library variants have been written in C, C++ and Rust, they all share the same set of APIs and core structures. This means that, despite the syntactic differences

between the languages, the features offered and the operations performed by the APIs remain consistent across all variants of libdevice. The focus is therefore on describing the goals and structure common to all libraries, rather than on the implementation specifics of each programming language.

The architecture of the libdevice delineates a conceptual framework designed to simulate the behaviour of an IoT device firmware. While not intended for practical implementation, this simulated environment serves as a valuable tool for deriving fictitious firmware behaviours within an IoT domain.

4.2.1 Architecture

This section elucidates the overarching structure of the project, shedding light on its components and the role they play in facilitating simulated firmware functionalities.

Modules

The libdevice structure revolves around components and APIs that imitate the expected functionality of an IoT firmware, specifically for use in a smart home setting. These APIs are contained within files.

1. **Features:** APIs, outlined in the feature module, extend the capabilities of the simulated firmware beyond basic device control. From writing data to drives to accessing network resources and interacting with peripherals like webcams, these APIs simulate a wide range of functionality commonly associated with IoT devices, allowing the exploration of various scenarios and use cases.
2. **Device:** The device module contains the APIs that simulate device control, for example, those dedicated to managing the lamps.

APIs Features

- **Write on Drive:** The write on drive function is an essential process in storing data on permanent storage devices, such as hard drives or flash memory. This operation is crucial in various contexts and use cases. When a user saves a file on their computer, its content is recorded on the hard drive or an equivalent permanent storage device. In a smart-home environment, a user can set up a personal profile by entering details such as name, preferences, and personalized settings, which then the IoT device stores and manages. Whenever a new smart device is added to the IoT ecosystem, its data, such as a unique identifier, configuration information, and access permissions, must be saved to the permanent storage device to ensure consistency and persistence of information over time.

- **Network Access:** In the implementation, the function simply makes a GET request to an Internet page to get its content. While this action may seem basic, it exemplifies the interaction that IoT devices could have with the external environment through the network. The ability to access and retrieve data from external resources is essential for the operation of IoT devices that must dynamically adapt to surrounding conditions and user needs.
- **Access to the Webcam:** The simulation function for accessing the webcam holds significant importance across a diverse spectrum of application scenarios for IoT devices, encompassing surveillance systems and audiovisual communication devices. A notable use case emerges within the realm of home video surveillance systems. Considering ownership of a security-focused IoT device for home protection when away, accessing the webcam enables environmental monitoring. Specifically, the device can be programmed to monitor its surroundings, detecting any suspicious movements or unauthorized activities within the home. When such anomalies are detected, the device can send push notifications to a smartphone, allowing the user to stay informed and take timely action.
- **Accessing the Audio Driver:** This function aims to simulate the use of an audio device, typically used to reproduce a wide range of auditory signals, ranging from simple tones to more complex voice commands. In the context of practical applications, a frequent use case is represented by the use of acoustic signals to notify significant events, such as the receipt of messages or system notifications, thus contributing to enriching the user experience and facilitating interaction with the device.
  Within the library, this feature provides a direct but effective method to evaluate the device's sound output capabilities through a computer's speaker system. This is achieved by generating a short and distinctive acoustic signal, lasting a few seconds, to provide a perceptible feedback to the user.
  This implementation underscores the significance of auditory feedback in device interaction. Transmitting audio signals plays an important role in enhancing engagement and enabling intuitive communication with the user.
- **Turning a Lamp On and Off:** These functions have been designed to provide an intuitive and direct mechanism to manage the status of the lamp, represented by on for activation and off for deactivation.

Binary

Within libdevice, there is an additional file dedicated to the creation of dummy firmware. This file consolidates various APIs from previous libdevice modules, producing an ELF binary that emulates the behaviours of an IoT device. Through

the orchestration of device behaviours and feature interactions, the fake-firmware file offers an environment for experimentation and assessment of diverse firmware implementations.

4.2.2 Compilation and Deployment

Compilation and deployment are fundamental processes in software development:

- **Compilation** transforms source code into an executable format.
- **Deployment** distributes an executable into an operating environment for use.

These processes are essential for delivering functional software.

The focus is now on compiling the different variants of the libdevice. Additionally, the implementation of a GitHub workflow is explored to automate all steps of building and deploying firmware. This systematic approach aims to manage the firmware development cycle efficiently, simplifying the compilation and deployment of libdevice using standardized tools and procedures.

Compilation of the libraries

The compilation of different variants begins with project configuration through a script called **autorun.sh**.

- In C/C++ this script, once executed, starts the project configuration and compilation process, making use of the **Meson** compilation tool in tandem with the Clang compiler for the C language version, and Clang++ for the C++ language version. Meson is a build system that reduces the difficulty in setting build parameters and retrieving dependencies.
- Compiling the Rust version is simplified by using **Cargo**, which acts as both a package manager and a compiler for Rust. The *build* command has been used to compile the project, resulting in a version with and a respective version without debug symbols, using the *--release* option. The release option, in addition to removing some debug symbols, optimizes the final binary ensuring a more efficient implementation than the simple debug compilation.

This compilation methodology provides a comprehensive and organized overview, managing the dependencies and ensuring that each libdevice variant is correctly configured and compiled according to a series of specifications, whether the language is in C, C++, or Rust.

GitHub actions workflow

A GitHub workflow (introduced in this Section 3.2.2) has been established to automate the compilation and distribution process for various variants of the dummy firmware. This workflow is detailed in two YAML files: **build.yml** and **deploy.yml**. Upon a push event, these files outline a sequence of operations to be performed one after the other.

The build.yml workflow primarily focuses on verifying the correct compilation of the main variants of the dummy firmware. It is divided into three distinct steps: *build-cpp*, *build-c*, and *build-rust*. Each of these steps is responsible for compiling the C++, C, and Rust variants of the libdevice, respectively. Each phase includes standard operations:

1. Cloning the code from the repository.
2. Installing necessary dependencies, including the Meson compiler and supporting libraries.
3. Configuring and compiling the project by executing the language-specific autorun.sh script.

In addition to the previously mentioned compilation steps, deploy.yml includes operations to compile all remaining variants:

- Statically and dynamically linked external dependencies.
- Stripped binaries.
- Non-compliant API behaviour.

These steps precede a crucial phase within deploy.yml, where the results of the previous compilations, namely the Elf binaries, are distributed and released on GitHub.
In summary, the implementation of these workflows aims to simplify and automate the compilation and deployment process of the various variants of the dummy firmware.

4.3 Dummy Firmware Variants

The implementation of variants within firmware satisfies a wide range of needs and usage contexts. These variants, which encompass the management of external dependencies, the presence or absence of debug symbols, and non-compliant behaviours in APIs, are useful for an analysis of possible use cases. Furthermore,

we will explore the importance of each of these variants and their impact on the construction and operation of firmware.

4.3.1 External dependencies management

Two main variants have been developed: **statically** and **dynamically** linked. This fundamental difference affects how external dependencies have been managed during the compilation process.

- In contexts prioritizing portability and independence from external libraries, such as embedded environments or self-contained software distributions, statically linked libraries are preferred. They integrate all dependencies directly into the executable file during compilation, ensuring complete self-sufficiency.
- Conversely, in scenarios requiring flexible dependency management or reduction of executable file sizes, like resource-constrained systems or large-scale application distributions, dynamically linked libraries offer a more suitable alternative. They require access to external libraries during program execution, providing flexibility and aiding in reducing the size of the executable file.

In the specific context of C and C++ variants, significant challenges have emerged in producing a statically linked version, mainly caused by retrieving and compiling dependencies during the building phase. In particular, difficulties have been encountered with libraries involved in interfacing with audio devices and accessing the Internet. These libraries presented only dynamic dependencies to internally used libraries, making it impossible to obtain a static version of the dependencies during compilation. As a result, we have reduced the implementation of static libdevice variants written in C and C++, eliminating the use of those dependencies and limiting the libraries to a minimalist variant. These contain only the features which do not require compiling external dependencies: write on a drive and access the webcam. Figure 4.2 illustrates the cases of static variants of libdevice implemented in Rust and the dynamic case implemented in C.

4.3.2 Stripped binaries

During the compilation and deployment, another considered variant provides the absence of debug symbols. Debug symbols are additional information embedded into the source code during compilation for analysis and debugging. These include variable names, function names, and code structure information. Obtaining versions of the code without these symbols is crucial to emphasize their significance in the analysis. Their absence renders detailed analysis of the binary impractical, as compiled programs lose references essential for identifying and understanding specific code segments during execution. Consequently, a **stripped binary** has

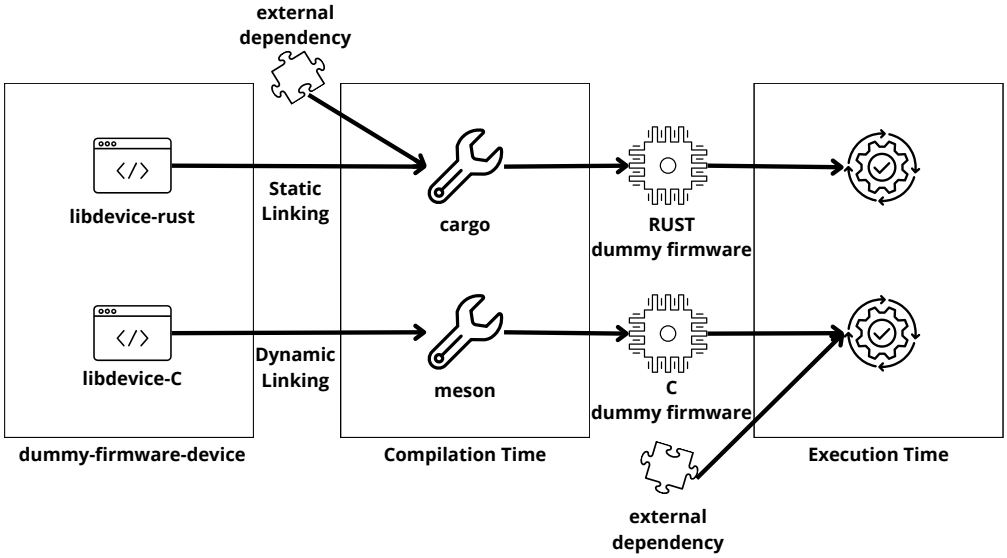

Figure 4.2: Example of dummy firmware workflow

been produced, where debug symbols and non-essential information are removed. This process notably reduces the size of the executable file, enhancing efficiency and security. Eliminating this information helps safeguard sensitive data within the source code, such as debug strings, access tokens, or authentication information, which could be exploited if present in publicly distributed binaries.

On the other hand, retaining debug symbols, although it increases the size of the binary, offers several significant advantages. This detailed information can be leveraged when analyzing software for a better understanding of its structure and internal workings. In this context, the **DWARF format** provides an organized structure for storing useful data, such as variable names, data types, and other program structure information. This format is used in the field of debugging to represent detailed information in executable files.

4.3.3 Non-compliant API behaviour

A final variant of the firmware has been created to provide APIs that deviate from the expected behaviours defined in the main variants. This branch focuses on generating situations where functions, while executing their intended operations, exhibit unexpected or undesirable behaviour. For instance, functions responsible for file writing may initiate network operations, such as sending files to remote platforms. Similarly, a webcam access function might redirect image streams to unexpected network destinations. These scenarios may represent anomalies in the

API's behaviour, potentially indicating vulnerabilities or programming errors.

4.4 Final Remarks

In conclusion, the development of the Dummy-Firmware-Device represents an important element for the future analysis of IoT firmware. Its implementation, through the definition of the libdevice, offers a reliable and accessible means of replicating firmware behaviour across different platforms and satisfying specific application needs. By creating variants of the libdevice in C, C++, and Rust, a complete framework for IoT emulation has been established. Providing dummy firmware variants with statically and dynamically linked libraries, along with variants for including or removing debug symbols, improves versatility in analyzing the resulting ELF binaries. Furthermore, the introduction of a variant that shows non-compliant API behaviour highlights the importance of evaluating firmware in potentially vulnerable scenarios.

Chapter 5

A tool for firmware certification

5.1 Introduction

The landscape of IoT, despite its advantages in terms of efficiency and convenience, is often plagued by concerns regarding the security and reliability of connected devices and systems. Particularly, the lack of standardization represents a significant gap in this context [19]. This deficit creates an environment where the security and reliability of IoT products can be compromised, as there is no formal guarantee regarding the quality and compliance of the firmware used.

Certifying firmware for IoT devices could address this issue, offering numerous advantages.

- Firstly, certification would allow consumers to easily identify products that meet certain security and reliability standards. This would help ensure greater peace of mind for end-users, while simultaneously reducing the risk of vulnerabilities and security breaches associated with uncertified firmware.
- Moreover, certification would positively impact the practice of IoT software development. Developers would be encouraged to allocate more resources to secure coding and code quality verification. This would enhance the robustness and resilience of firmware, reducing the likelihood of critical errors or security vulnerabilities.

- Lastly, firmware certification could contribute to promoting greater transparency and accountability in the IoT ecosystem. Developers would be required to accurately document the functionalities and behaviours of their firmware, enabling better understanding and evaluation by end-users and regulatory bodies.

In summary, firmware certification is crucial for creating a safer, more reliable, and responsible IoT environment. The implementation of certified standards and processes could improve the quality of IoT products.

5.1.1 ELF binary analysis

In the context of firmware certification for IoT devices, a important aspect is the analysis of **ELF** (Executable and Linkable Format) binary, as previously explained (see Section 2.3.3). This section aims at introducing such analysis, highlighting its advantages and disadvantages, as well as the rationale behind its consideration.

As mentioned in the introductory chapter (see Section 2.3.2), IoT firmware represents a fundamental component for Internet of Things (IoT) devices, defining itself as the software incorporated directly into the hardware device. This firmware, closely linked to the peculiarities of the IoT environment, requires particular attention to ensure security and reliability, given the extensive interconnection and data collection involved.

Motivations for analyzing ELF binary

Analyzing ELF binary files offers numerous advantages in the context of IoT firmware certification.

- Firstly, this format is widely used in Unix-like operating systems, particularly Linux, making it a natural choice given the widespread adoption of such systems in the IoT ecosystem. Figure 5.1[1] shows the great dominance of the Linux operating system as a preference in this context.

[1] *Copyright (c)2018, Eclipse Foundation, Inc. | Made available under a Creative Commons Attribution 4.0 International License (CC BY 4.0)*

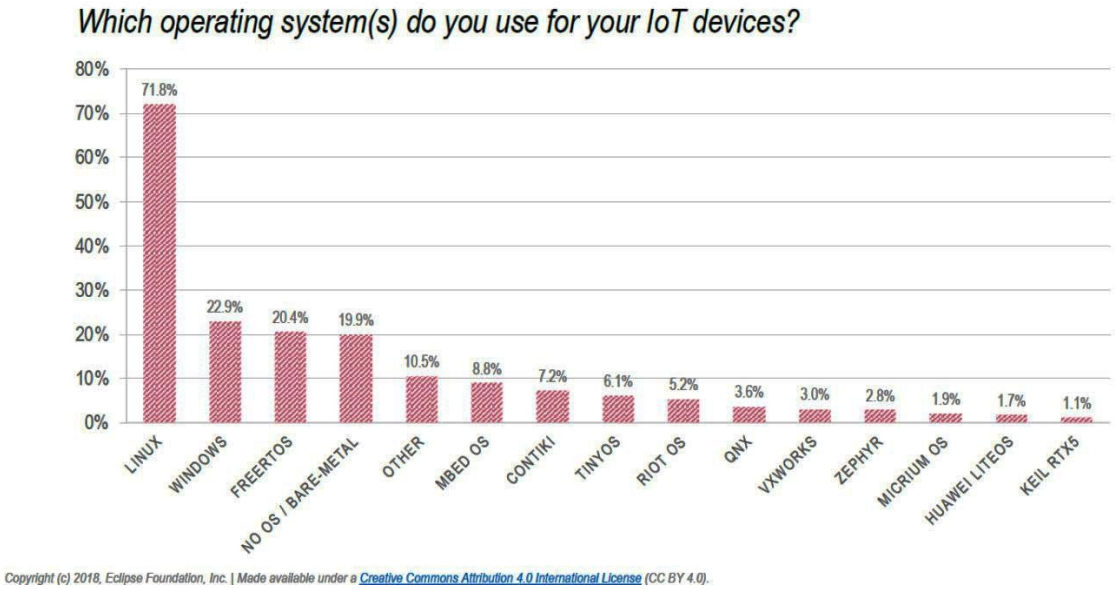

Figure 5.1: Linux dominates the scene with 71.8% of users preferring it.

- Additionally, ELF binary files contain detailed information about program structures and functionality, providing a comprehensive overview of the firmware and potential points to examine.
  - **Advantages:** ELF binary analysis enables the examination of firmware at a lower level, providing a detailed view of program instructions and data structures. This approach allows for the identification of potential security vulnerabilities, understanding firmware behaviour, and ensuring compliance with security standards and development policies. Furthermore, ELF binary analysis can facilitate the creation of preventive measures and vulnerability correction, thereby improving the overall security of IoT firmware.
  - **Disadvantages:** However, ELF binary analysis may present some disadvantages, including the complexity of program structures and the need for specialized skills to conduct a thorough analysis. Additionally, ELF binary analysis may not reveal all vulnerabilities present in the firmware, necessitating the adoption of complementary approaches to ensure a comprehensive security assessment.

5.1.2 Preliminary approaches

The developmental trajectory of the manifest-producer tool started with a preliminary analysis aimed at probing the inherent challenges associated with analyzing ELF binaries within the context of certifying firmware for IoT devices. In this phase, the use of tools such as **radare2**[2] and **objdump**[3], which enabled an exhaustive analysis of the dummy firmware variants, was crucial. This preliminary analysis method facilitated comprehension of the file structure, enabling the identification of areas relevant to firmware certification. Specifically, the analysis, initiated through **code disassembly**[4](see Section 2.3.3), focused on system calls, deemed crucial in clarifying the firmware's authentic behaviour. Indeed, system calls allow user programs to access functionality that requires access to operating system privileges, such as file and memory management, communication with I/O devices, and many other operations. However, despite the granular control offered by the analysis with the cited tools, the imperative need to adopt an automated solution has emerged, given the laboriousness and impracticality associated with this approach. Furthermore, it is important to note that since Rust has been chosen as the programming language for the development of the manifest-producer tool (for reasons related to the characteristics of the language explained in Section 3.2.1), the tools radare2 and objdump do not offer adequately supported crates for direct integration within a Rust program. Consequently, it was not possible to implement the manifest-producer directly using the workflow of these tools to obtain references to the various system calls during the analysis of the binaries.
After a comprehensive preliminary analysis, two primary approaches for ELF binary analysis have been delineated: static analysis and dynamic analysis.

Static analysis

Static analysis entailed the exploration of two distinct methodologies.

1. The first method conceived involved the use of **hexadecimal patterns**: they make it possible to identify and compare particular byte sequences in a hexadecimal representation, which is useful in this context for identifying specific behaviour representing system call instructions. However, this strategy was immediately recognized as complex and onerous in terms of computational resources, as it required the generation and management of a large corpus

[2]Radare2 is a complete framework for reverse-engineering (ref.2.3.3) and analyzing binaries.

[3]objdump is a program for displaying various information about object files on Unix-like operating systems.

[4]A disassembler is a computer program that translates machine language into assembly language.

of hexadecimal models to cover the multiple architectures supported, as not all share the same syscall patterns. Moreover, the requirement to keep these models constantly updated, to adapt them to changing architectures, would involve considerable effort. For example, considering the syscall write in x86-64, we could identify a possible hexadecimal pattern as **0xB801 0F05**, where:

- **B8 01** corresponds to *mov eax, 0x01*, where the assembly instruction mov loads the syscall write, represented by the number 01, into the eax register. By convention, the system associates a positive integer with each syscall.
- **0F 05** represents the syscall instruction, which may differ depending on the architecture.

However, recognition of this pattern alone may not be sufficient to reliably identify the syscall, as there may be other instructions in the code between the one that loads the syscall number into the appropriate register and the one that invokes it, shown in the example above. Therefore, analysis based on hexadecimal patterns requires careful consideration of context and may be prone to errors if not implemented with attention and a thorough understanding of the binary code.

2. The second method was to create a system call **mapping table**, which is a systematic approach to correlate system call numbers with their respective names. As explained in the previous point, by convention the operating system uses positive integers as identifiers for the various syscalls. Pseudo-code 5.1 shows an example mapping table implementation.
   This methodology inherently exploits the insights from the previous approach by focusing on the .text section of the ELF file where the executable code is contained. Through an analysis of this section, the mapping process establishes a consistent association between the numerical identifiers of system calls and their semantic representations. Compared with the use of hexadecimal patterns, this approach significantly improves code readability and generalizability, as it can be applied to different architectures (such as x86, x86-64, ARM, ...)

Listing 5.1: Pseudo-code that maps the syscall number to its name.

```
1        create a new list called syscall_names
2        add ("write", 0x01) to the syscall_names list
3        add("sendto", 0x44) to the syscall_names list
4        add ("recvfrom", 0x45) to the syscall_names list
5        // Add more system calls
6
```

without requiring substantial modification or adaptation. However, despite the inherent advantages, it is important to note that the possible absence of a system call in the mapping table could result in incomplete categorization, compromising the integrity of the analysis.

Dynamic analysis

Dynamic analysis is characterized by its ability to provide a probable and contextualized representation of program behaviour by focusing on **tracking** system calls during firmware execution using the **strace** tool[5]. However, the effectiveness of this approach is closely related to the availability and functionality of strace in the target system. The presence of strace and its ability to provide interpretable output play a crucial role in determining the accuracy and usefulness of the dynamic analysis.
Another significant aspect is that dynamic analysis records the execution flow of a single firmware instance. Therefore, it omits consideration of every possible alternative path that other instances might take in different contexts or with different inputs. This implies that although dynamic analysis provides realistic and immediate data, its coverage is inherently limited. To obtain a complete and thorough understanding of firmware behaviour, it may be necessary to run many instances to explore all possible combinations of scenarios and input configurations.

5.1.3 Definitive analysis

Preliminary analyses aimed at comprehending the structure of ELF files and configuring an analysis to identify suitable parameters for certification highlighted the necessity for a more precise and targeted methodology. In particular, greater emphasis has been placed on the implementation of individual public APIs rather than only relying on the entire firmware execution. This approach allows the analysis to focus on specific **code blocks**, thereby enhancing the granularity and precision of the evaluation, and enabling the division of firmware functionalities among different APIs. Given their significant contribution to the analysis, they have thus been recognized as important *parameters* for firmware certification.
This orientation of the analysis towards a more static view suggests not only the identification of the main functions but also the identification of their memory addresses, thus enabling the correct disassembly of the code and a detailed analysis of system calls. Furthermore, in the context of dynamically compiled firmware (see section 4.3.1), the importance of identifying library function calls has been also recognised.

[5]strace is a diagnostic and debugging utility for Linux.

5.2 How manifest-producer works

The manifest-producer tool, developed in Rust, is therefore designed to perform the firmware certification process through ELF binaries analysis. Its primary objective is to ensure firmware integrity and compliance through two key steps:

1. **API Detection:** Firmware developers must provide the ELF binary together with a list of public APIs used in it. This list forms the basis for the analysis, as each API is independently examined to assess its adherence to the intended behaviour.

2. **Behavioral Analysis:** Once the APIs provided by the developer are identified, the tool disassembles its code and searches for system calls and external library functions, evaluating whether the APIs align with the expected behaviour or exhibit undesired characteristics.

Through this process, the manifest-producer tool enables validation of firmware compliance with security, reliability, and performance standards. Ultimately, it generates three distinct manifests in JSON format, which encapsulates the extracted and processed information.

In essence, the manifest-producer serves as an instrument in binary firmware certification, offering a systematic approach to validate aspects of firmware behaviour and foster a safer IoT ecosystem.

5.2.1 API Detection

The first point of the analysis aims to carefully examine the public APIs provided by a firmware developer, in order to assess their adherence to specifications and ensure their integrity and compliance. Initially, the tool checks for the presence of the debug sections within the ELF file. The debug sections contain useful data for analysis purposes, including symbols representing variables, functions and other code entities. This information is essential for identifying and understanding the structure of a firmware.

Once these sections have been confirmed, the tool proceeds to retrieve the list of APIs provided by the firmware developer. This list, consisting of a set of strings representing the names of the APIs, is essential for guiding the search process within the symbol table[6] of the ELF file. The symbol table in a binary file contains information on variables, functions and other code entities, along

[6]symbol table holds information needed to locate and relocate a program's symbolic definitions and references.

with their memory addresses. This information is used by the operating system to link program symbols to data and executable code during program execution. Therefore, examining the symbol table makes it possible to identify functions and variables in the firmware.

The tool then scans the symbol table, examining each symbol to determine whether it represents a function and is associated with a valid code section. These criteria are crucial for distinguishing valid functions within the firmware. For each symbol that meets these criteria, the tool checks if the function name matches one in the API list. If there is a match, it is an API to be analysed and then the tool obtains the starting address and size of the associated code block.

At the end of this operation, an appropriate data structure contains the information for each API in the list:

- Name
- Starting address
- End address

In addition, the structure provides the possibility of recording each system call associated with an API, thus offering a broader context for evaluating the behaviour and API usage in the context of a firmware.

In conclusion, the process above represents the first fundamental step in firmware certification process, allowing the identification of public functions within the code, and thus providing a solid base for the subsequent stages of firmware analysis.

5.2.2 Behavioural analysis

The analysis process continues using the previously collected information stored in the dedicated data structure. This process is mainly divided into two phases:

1. **Code Disassembly:** During this phase, the process focuses on analyzing the executed instructions to understand the operations performed by a function. This step translates the machine code into readable and understandable instructions, i.e. assembly code. This translation simplifies the analysis of the program execution flow and facilitates the identification of system calls.

   In particular, attention is focused on the identification of two specific instructions: **call** and **lea**.

   - call instruction receives a single piece of information: the address of the function to invoke. This can happen in two ways: either by directly passing the address or by loading it into a register. For example, we can

express a call instruction as call 0x1352 or call rax, where the function's address is either directly specified (0x1352) or has been previously loaded into the RAX register.

- lea instruction, short for **load effective address**, is used within the code to load the address of a function which will be engraved by the program into a specific register. For example, a lea statement might have the following syntax: lea 0x6452(%rip), %rax. In this context, lea instruction loads into the destination operand, the rax register, an offset arithmetically added to the rip register.

These two instructions are fundamental in the analysis of the disassembled code, since they allow to identify all system calls made by a function, providing a fundamental overview of the operations performed by an API and its interactions with system libraries and the operating system.

2. **System Call Identification:** During this phase of the analysis, system calls occurring within a function are detected and logged. These calls are significant for the analysis as they offer insights into the API's behaviour. For instance, the detection of the sendto system call implies potential involvement in network operations, as sendto is typically used for transmitting data within a network environment.

   In this context, it is essential to acknowledge the potential scenario where certain system calls are not identified. This could occur when API operations are conducted within functions called from external libraries. Even in these situations, the tool can obtain the name of the function associated with the external library called by the API. Once the addresses have been obtained from lea or call instructions, it becomes crucial to consider how the external dependencies are managed a building process. This analysis highlights two alternatives:

   - For **static linking**, identifying the name of the called function associated with the address is a relatively simple process. This is because, as highlighted in section 4.3.1, the code of the functions is contained within the .text section of the binary. This structure simplifies the access to the various function allocations, allowing the tool to directly consult the symbol table. From there, it is possible to retrieve the index corresponding to the string table, providing the exact name of the function invoked by an API. This process is clearly illustrated in Figure 5.2, which details its operational flow.
   - In the **dynamic linking** case, the process is theoretically more cumbersome. As described in section 4.3.1, during dynamic linking, not all

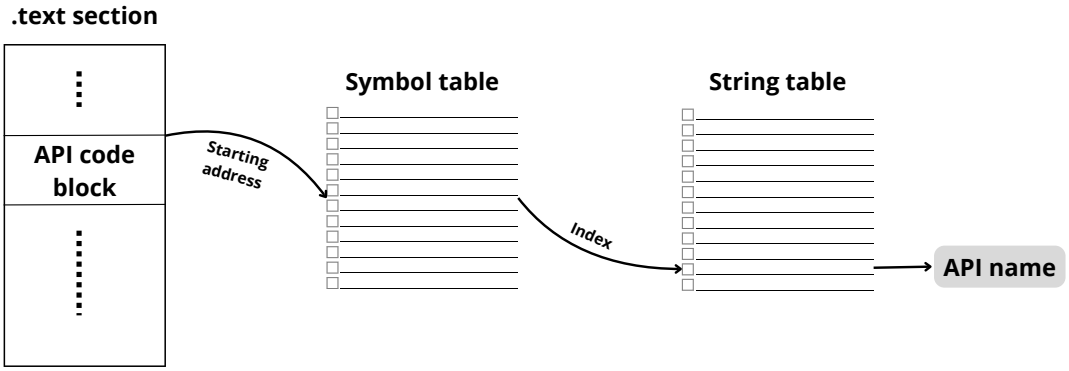

Figure 5.2: Example of function name extraction.

addresses are resolved at compile time. This necessitates accessing the **Procedure Linkage Table (PLT)**[7] to retrieve the names of functions from external libraries. These addresses are dynamically resolved at runtime, making the process of identifying function names more intricate and dynamic. To simplify this process, the tool adopts a different strategy. First, identify the .plt section containing the PLT table. Next, it loads all the addresses associated with their names into a hash table. This design choice significantly speeds up the search because the tool can perform a simple query against the hash table rather than performing more complex operations in terms of time and number of operations.

5.2.3 Observations

The following observations emerge from the analysis of the different firmware produced by the dummy-firmware variants (described in chapter 4).

- **Compiler Interpretation Variants:** Compiler behaviour varies significantly depending on the programming language used to create the firmware. In particular, a clear distinction in behaviour was noted between firmware written in Rust and those in C/C++. In the case of Rust firmware, the compiler tends to predominantly use the lea instruction for function calls. On the other hand, in the case of C/C++ firmware, the compiler tends to perform some optimizations, such as function inlining, that directly embed a function address in the call instruction.
- **External libraries functions:** As previously discussed, programs often

[7]PLT is a table used to manage calls to functions present in dynamically linked libraries.

use functions from external libraries to accomplish specific tasks. While the manifest-producer can detect these calls, the analysis primarily centres on the names of the external library functions, without delving deeply into their implementation to identify system calls. This limitation is driven by the potential complexity that may arise from a detailed analysis of library functions. For instance, these might call additional functions, leading to increased analysis depth.

- **Code Mangling:** Code mangling is a technique used by compilers to distinguish between functions and variables with the same name within a program. When a compiler encounters a function declaration, it generates a unique name for that function incorporating information about the type and number of function parameters, as well as the return type and other characteristics. For example, in C++, the function name writeOnDrive could be changed to _Z11writeOnDrivev, allowing a distinction between various writeOnDrive functions. For analysis purposes, the APIs provided must not be mangled to facilitate their identification. Additionally, the tool can demangle the calls identified during code disassembly, enabling a clean flow of syscalls and functions to be obtained.

5.3 Manifest Generation

The generation of JSON manifests represents the last phase in the binary analysis, as it allows the essential information obtained from previous firmware analysis steps to be represented in a structured way. These manifests provide an important overview of the salient features of the analyzed ELF file and its interactions with system calls and library functions.

5.3.1 Manifest for basic information

Manifest for basic information, is a starting point for understanding the firmware. It provides general information about the ELF file, such as its file name, its programming language used, its target architecture, and its dependency linking type, static or dynamic. Additionally, it lists all public APIs identified in the code, providing a preliminary indication of the functionalities offered by the firmware. Figure 5.2 shows what base information has been reported within the manifest during an analysis.

Listing 5.2: Some basic information obtained from the analysis.

```
{
  "APIs found": [
    "accessWebcam",
    "accessNetwork",
    "writeOnDrive",
    "turnLampOff",
    "turnLampOn"
  ],
  "architecture": "x86-64",
  "endianness": "Little",
  "entry_point": "0x1b0f0",
  "file_name": "fake-firmware-c-dynamic",
  "file_type": "Dynamic Library",
  "header_size": 64,
  "link": "dynamically linked",
  "programming language": "C99"
}

```

5.3.2 Manifest for syscall flow

Manifest for syscall flow, provides a detailed overview of the system calls and library functions associated with each API identified in the firmware. This document provides a structured sequence of the operations performed during the execution of the various public functions. The peculiarity of this static analysis lies in its ability to comprehensively capture the API interactions with the system libraries and operating system, considering all possible execution paths that may not be explored whether a single dynamic instance of the program is used.

By representing this information, this manifest contributes to a detailed and comprehensive understanding of the API's behaviour and its impact on the execution environment. Such static analysis is essential for revealing dependencies and interactions of the API with the underlying system, providing a solid base for evaluating the security and performance of a firmware. Figure 5.3 shows which system calls and functions have been identified in the code of accessNetwork API during an analysis.

Listing 5.3: Call flow obtained for the accessNetwork function.

```
{
  "name": "accessNetwork",
  "syscalls": [
    "curl_global_init",
    "curl_easy_init",
    "curl_easy_setopt",
    "curl_easy_perform",
    "curl_easy_strerror",
    "fprintf",
    "curl_easy_cleanup",
    "fprintf",
    "curl_global_cleanup"
  ]
}

```

5.3.3 Manifest for features

Manifest for features, classifies APIs according to their functionality offering a structured overview of firmware's capabilities. This categorization occurs through a systematic process that evaluates the system calls and library functions associated with each API, identifying the tasks performed and grouping them into meaningful categories.

The categorization process is based on a predefined set of functional categories, such as file manipulation, network access, device management, encryption. Each category is associated with a set of keywords or substrates that indicate the presence of specific functionality within system calls and library functions.

Once a match has been found, APIs are categorized based on their features. For example, when an API contains file manipulation system calls, it will be categorized under File Manipulation. This approach helps to categorize APIs based on what they can do, giving a clear picture of the firmware's main features. Figure 5.4 shows how APIs have been categorized during an analysis.

Listing 5.4: Categorizations for the various functions identified.

```
{
  "accessNetwork": [
    "Network Access"
  ],
  "accessWebcam": [
    "Device Access"
  ],
  "writeOnDrive": [
    "File Manipulation",
    "Device Access"
  ]
}
```

5.4 Structure

manifest-producer is the result of a development process aimed at ensuring the robustness and scalability of the tool. Due to the use of the ci-generate, it was possible to start the initial project configuration, with a modular structure that made the code maintainable, with a built-in Continuous Integration flow.

5.4.1 Modular approach

The heart of the manifest-producer lies in its modular architecture, which divides the system into independent components, each of which plays a specific role within the ELF file analysis process. This approach allows for a separation of concerns[8], making it code understanding and changes. Furthermore, modularity promotes the reusability of components, allowing new functions to be implemented with minimal effort.

Each module is responsible for a set of features. The main modules include:

- **manifest_creation:** This module deals with manifest creation, i.e. a structured representation of the information extracted from the analysis of ELF files. These manifests are essential to understand the behaviour and dependencies of the analyzed software.
- **api_detection:** This module deals with the detection of APIs within the analyzed ELF files. The identified APIs are fundamental to understand what

[8]Separation of Concerns is a software design principle that suggests dividing a system into separate modules, each of which deals with a single responsibility.

are the features offered or used by the software.

- **cleanup:** This module manages data cleaning and analysis of system call flows, thus helping to guarantee the correctness and integrity of the extracted information.
- **elf_utils:** This module provides utility functions for manipulating ELF files, such as reading and parsing its metadata.
- **dwarf_analysis:** This module deals with the analysis of DWARF data within ELF files, which provides detailed information on software structure and functionalities.
- **code_section_handler:** This module handles the extraction and disassembly of API code sections within ELF files, taking care of both static and dynamic linking.
- **plt_mapping:** This module returns a mapping of PLT function addresses to their respective names.

In short, each module contains a series of components that work together to perform specific tasks related to their specialized domain within the ELF file analysis process. This modular approach allows for better organization and structure of the code, facilitating the maintenance and scalability of the manifest-producer.

5.4.2 Continuous Integration

The manifest-producer project implements a Continuous Integration (CI) system to evaluate the quality of software and facilitate collaborative development.

The CI process is triggered whenever there are pushes on the main branch or when pull requests are opened. The system uses GitHub Actions to perform a series of predefined steps, which include:

1. Static analysis of the code using Clippy and Rustfmt to ensure compliance with formatting standards and catch common mistakes.
2. Compiling the code to verify that there are no compilation errors and generating documentation to provide clear guidance on the use of the various software components.
3. Running automated tests and measuring code coverage to identify untested parts of the code and ensure good test coverage.
4. Verifying project dependencies for being up-to-date and secure, because dependencies may be subject to change over time.

5. Using tools to detect and fix potential security problems such as memory leaks, race conditions, and memory access errors.

Using a CI ensures software quality in a continuous and automated way, facilitating the development process and reducing the risk of introducing errors within the code. This process automatically runs a series of tests whenever changes are made to the source code, ensuring that the software remains stable and functional.

5.4.3 Test Development

The manifest-producer project uses unit testing and snapshot testing as part of the software development process. Unit tests check the behaviour of individual units of code, while snapshot tests, which are a form of integration testing, test the behaviour of the entire system by integrating and testing different units of code together.

Using tests in manifest-producer, as we did for ci-generate (reference here 3.4), proves extremely useful for several reasons:

- **Ensure Software Correctness:** Testing allows to identify and correct errors in code promptly, ensuring that the software works as intended.
- **Facilitate Refactoring:** The presence of a complete set of tests allows to make changes to the code with greater confidence, as it is possible to quickly check whether the changes introduce errors or break existing functionality.
- **Ensure Code Quality:** Unit tests and snapshot tests provide immediate feedback on code quality, encouraging high-quality development practices and writing robust and maintainable code.
- **Support Project Continuity:** Due to testing, software reliability can be maintained over time, enabling the integration of new features and enhancements.

5.5 Final Remarks

The manifest-producer tool represents an alternative in the certification process of firmware for IoT devices, offering a systematic approach to analyze ELF binaries and generate comprehensive manifests which encapsulate information. Through API detection and behavioural analysis, the tool ensures firmware integrity, and identification of potential vulnerabilities. The modular architecture and continuous integration process ensure code quality, maintainability, and reliability.

The analysis of ELF binaries and the generation of manifests provide valuable insights into firmware functionality, system dependencies, and interactions with the underlying environment.

Chapter 6

Performance analysis

6.1 Introduction

The performance analysis of the manifest-producer tool aims to evaluate its effectiveness in analyzing a variety of ELF files. This evaluation has been conducted by running the tool on a diverse selection of binaries, including dummy-firmware variants (introduced in Chapter 4) and other known projects such as FFmpeg, xi-core and OpenCV. The tools mentioned are open-source, meaning their source code is publicly available, enabling access to APIs for analysis through the manifest-producer.

- **FFmpeg**[1] has been chosen for its broad utility in digital media manipulation. The complexity of the source code, primarily written in C with some critical parts optimized in assembly, provides an opportunity to assess the tool's performance in scenarios where complexity may impact the analysis of ELF files, making it a relevant study subject to evaluate the performance of the manifest-producer tool in practical contexts.
- **OpenCV**[2] has been included in the analysis to examine the performance of the manifest-producer on ELF files involving complex computational calculations and intensive processing.
- The **xi-core**[3] project, being the core of the Xi text editor, represents an opportunity to evaluate the tool's capabilities in analyzing ELF binaries from

[1]Github repository:https://github.com/FFmpeg/FFmpeg

[2]GitHub repository: https://github.com/opencv/opencv

[3]GitHub repository: https://github.com/xi-editor/xi-editor

projects that require optimal performance and efficient management of system resources.

As regards the binaries obtained from the dummy-firmware variants, the static and dynamic versions of each of the programming languages used have been taken into consideration:

- **C-static** and **C-dynamic** for the respective versions written in C
- **Cpp-static** and **Cpp-dynamic** for the respective versions written in C++
- **rust-static** and **rust-dynamic** for the respective versions written in Rust

This broad range of ELF binaries provides a comprehensive methodology for evaluating the tool's performance in real-world contexts, allowing for a detailed understanding of its strengths and possible areas for improvement.

Selected tools

Two performance analysis tools, **Hyperfine**[4] and **Heaptrack**[5], have been used to conduct a thorough analysis.

- **Hyperfine**, a benchmarking tool[6], plays a role in analyzing the performance of the manifest-producer. It measures the execution times of programs, providing valuable insights into the duration of the analysis for each considered ELF binary file. Hyperfine's repeated benchmark runs offer an overview of the manifest-producer tool's performance, particularly regarding its speed and responsiveness.
  Hyperfine comes with a default configuration, including a warmup of 100 iterations followed by 1000 actual runs. This setup ensures a stable execution environment, minimizing the impact of any initial performance variations due to initialization processes or caching. Consequently, the data obtained through Hyperfine offers a dependable understanding of the manifest-producer tool's performance, effectively eliminating disturbances and providing a solid base for comparative analysis of execution times among different ELF binaries.
- **Heaptrack** is a performance analysis tool designed to provide an insight into memory usage during program execution. This tool plays a role in the performance analysis of the manifest-producer tool. It enables monitoring and

[4]GitHub repository: https://github.com/sharkdp/hyperfine

[5]GitHub repository: https://github.com/KDE/heaptrack

[6]A benchmark is a method used to evaluate the performance of a computer system or a software.

evaluating memory allocation in the software under examination by recording information about memory consumption peaks, temporary allocations, and any memory leaks. This ability to identify memory management issues is essential for accurately and thoroughly assessing the efficiency of memory allocation in a software.

6.2 Data Analysis

Analysis of data collected from benchmarks using Hyperfine and memory allocation statistics obtained via Heaptrack provides a basis for understanding the performance and memory usage of the manifest-producer tool on considered binaries.

6.2.1 Time Efficiency

In analyzing the performance of the manifest producer tool, an investigation has been conducted to explore the correlation between the size of the analyzed binaries and the time required by the tool to process those binaries. This comparison intends to provide a more comprehensive and detailed view of the tool's behaviour in response to binaries of various sizes, divided according to the different programming languages used.

C Language

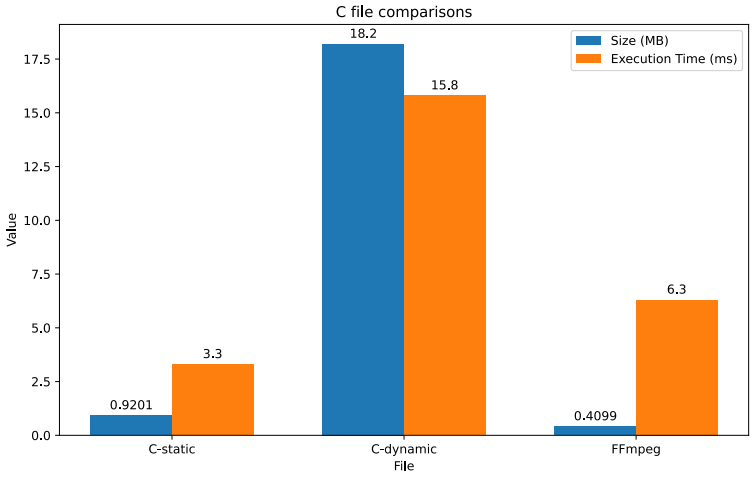

Figure 6.1: Comparison between binaries written in C.

As can be seen from the graph in Figure 6.1, a significant variation in file dimension

can be observed when analysing ELF files written in C language. However, the analysis of the data in the chart, with particular reference to FFmpeg and C-static, reveals that file size does not necessarily determine a longer execution time. For example, when comparing FFmpeg, which is 409.9 kB in size, with C-static, which is 920.1 kB, we notice that FFmpeg takes almost twice as long to execute compared to C-static, despite its smaller size. They are around 6.3ms for FFmpeg and 3.3ms for C-static. This behaviour can be attributed to several factors including, for example, the complexity of the data structures present in the binary, which require more in-depth analysis operations and therefore longer execution times by the manifest-producer. On the other hand, despite its large size of 18.2MB, the C-dynamic file demonstrates a relatively modest execution time, hovering around 16ms, as depicted in the graph.

C++ Language

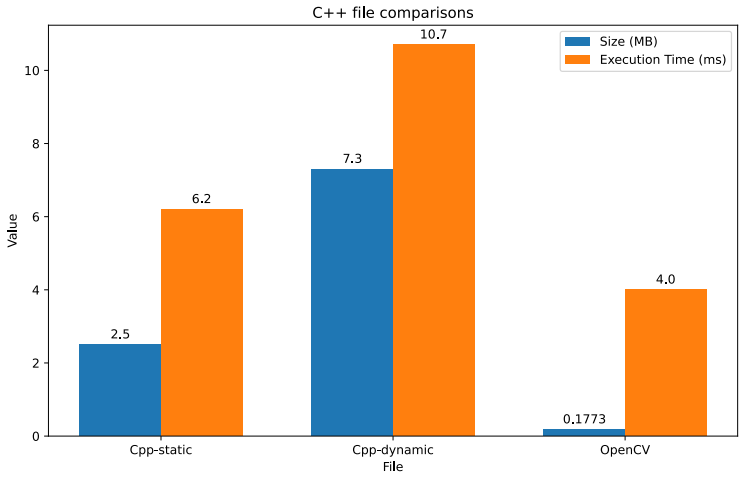

Figure 6.2: Comparison between binaries written in C++.

In the context of C++ binaries, from the graph in Figure 6.2, a correlation between file size and execution time is noted, albeit less pronounced than for C binaries. The tool can process C++ binaries uniformly despite their size. However, it is noteworthy that the OpenCV project, despite its small size likely attributed to code optimization, exhibits a considerable execution time. This emphasizes the importance of taking into account the practical circumstances or situations in which the tool is used, rather than only focusing on theoretical or ideal conditions. Even moderately sized binaries can require significant parsing time, especially if they contain complex, optimized code.

Rust Language

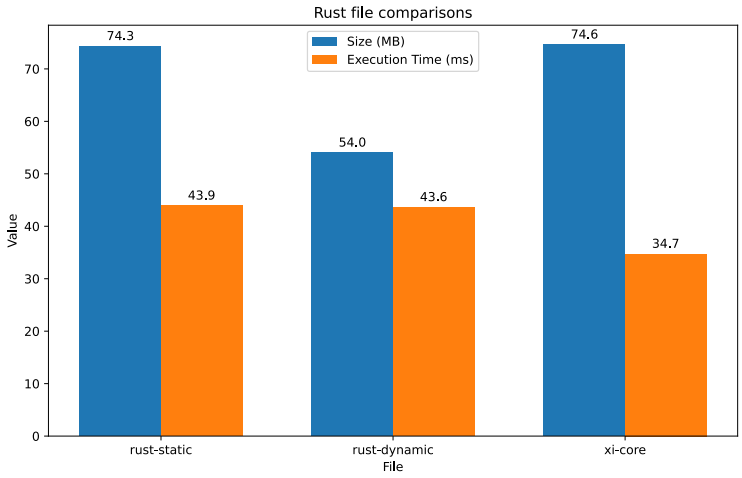

Figure 6.3: Comparison between binaries written in Rust.

In the context of analyzing Rust binaries, an interesting dynamic emerges between file size and execution time. As can be seen from the graph in Figure 6.3, the relationship does not follow a linear model, contrary to C/C++ observations. Although Rust binaries have larger dimensions (all above 50MB), there is no direct correlation between these and the average time required for the analysis. This phenomenon is evident in the collected data, as Rust binaries do not show a proportional increase in execution time relative to their size.

As an example, the results of the analysis of the rust-static and xi-core files are particularly significant: despite the similar size of these two files, the execution times are comparable. This consistency highlights the tool's ability to manage analysis operations efficiently, regardless of variations in the Rust binaries.

6.2.2 Memory Evaluation

In performance analysis, the evaluation of memory allocation offers an investigation into the behaviour of the tool in different operational scenarios. Through the use of tools like heaptrack, it is possible to capture relevant data on four key metrics:

- **Allocations** denotes the total number of memory allocations during program execution. This measurement reveals the workload related to dynamic memory management, providing insight into the complexity of allocation and deallocation operations within the code.

- **Temporary allocations**. These allocations are created and released during program execution in a short amount of time. Monitoring this parameter enables the identification of inefficiencies in temporary resource allocation, highlighting potential memory waste or suboptimal practices.
- **Peak heap memory consumption**. The peak memory allocated to the heap during program execution is a useful metric for evaluating a program of maximum memory requirements. This information can help evaluate how efficiently the program manages resources and adjusts to changing workload demands.
- **Peak RSS (Resident Set Size)**. It represents the maximum amount of physical memory (RAM) used by the program during execution, including data, stack and heap. This parameter provides a general assessment of how much memory the program is using, helping to understand its overall impact on available resources.

C Language

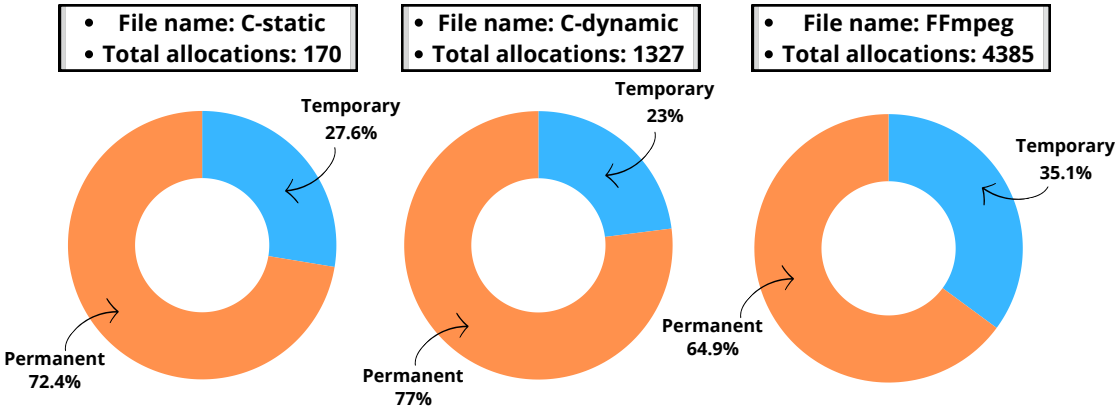

Figure 6.4: C binaries allocations.

In the realm of C programming language, a discernible variance in performance becomes evident across the diverse files subjected to analysis, shown in Figure 6.4. Notably, the FFmpeg file exhibits a pronounced disparity in allocations, registering a substantial total of 4385 allocations in contrast to 1327 for C-dynamic and 170 for C-static. This data unveils a distinct footprint characterized by pronounced resource utilization and memory management activities throughout program execution. Additionally, the percentage of temporary allocations relative to total allocations within FFmpeg stands notably higher at 35.1%, contrasting with C-dynamic's 23.0% and C-static's 27.6%. Such a metric suggests a heightened

frequency in the generation and release of temporary resources within the program's execution flow.

File	Size	Memory Peak	Peak (RSS)	Allocations	Temporary Allocations
C-static	920.1 kB	1.1 MB	9.8 MB	170	47
C-dynamic	18.2 MB	19.1 MB	28.7 MB	1327	305
FFmpeg	409.9 kB	1.8 MB	11.7 MB	4385	1538

Table 6.1: Allocation and memory peak data of binaries written in C.

Regarding memory peak metrics, in Table 6.1 it is possible to observe a series of data collected. FFmpeg exhibits the lowest peak at 1.8MB, despite its elevated allocation count. This observation might indicate proficient memory management practices in the program, despite the vigorous allocation activity. However, FFmpeg's peak RSS presents a notably higher figure, reaching 11.7MB, indicative of a more robust utilization of system resources. In comparison, despite its relatively larger size, C-static showcases lower memory and RSS peaks compared to FFmpeg, with respective values of 1.1MB and 9.8MB. On the contrary, C-dynamic, the largest file in the analyzed group, exhibits the highest peak memory and peak RSS, reaching 19.1MB and 28.7MB, respectively. This aligns with its file size of 18.2MB and the number of allocations made (1327), indicating a significant utilization of system resources during execution.

C++ Language

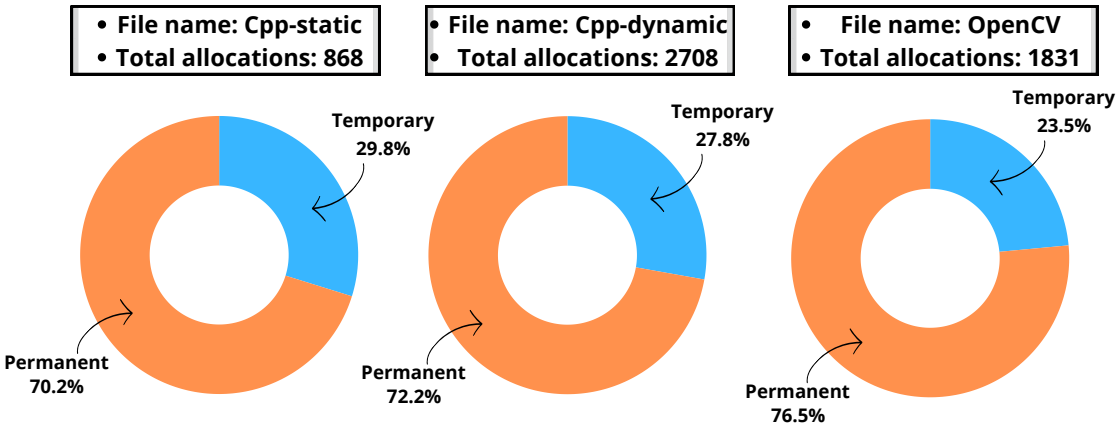

Figure 6.5: C++ binaries allocations.

In the context of C++ programming, Figure 6.5 shows a significant variation in performance between the different analyzed files. In particular, Cpp-dynamic

stands out for the highest number of allocations, totalling 2708, compared to 868 for Cpp-static and 1831 for OpenCV. These data suggest marked memory management activity and intense resource utilization during program execution. Furthermore, both files, C-dynamic and OpenCV, show a significant percentage of temporary allocations compared to the total, about 23% and 35.1% respectively. That denotes a frequent cycle of generation and release of temporary resources in the analysis execution.

File	Size	Memory Peak	Peak (RSS)	Allocations	Temporary Allocations
Cpp-static	2.5 MB	3.0 MB	10.5 MB	868	259
Cpp-dynamic	7.3 MB	8.0 MB	17.9 MB	2708	753
OpenCV	177.3 kB	670.5 kB	10.7 MB	1831	406

Table 6.2: Allocation and Memory Peak data of binaries written in C++.

In Table 6.2 it is possible to observe data relating to memory peaks. Cpp-dynamic file emerges with the highest peak at 8.0 MB, followed by Cpp-static at 3.0 MB and OpenCV at 670.5 kB. This pattern may indicate a more aggressive approach to memory management within Cpp-dynamic, likely due to its larger number of allocations and relatively larger effective file size. However, it's worth noting that although the peak RSS of Cpp-dynamic is the highest at 17.9 MB, OpenCV exhibits a peak RSS value of approximately 10.7 MB, which is significantly high relative to its file size. This observation suggests a substantial consumption of system resources during program execution.

Rust Language

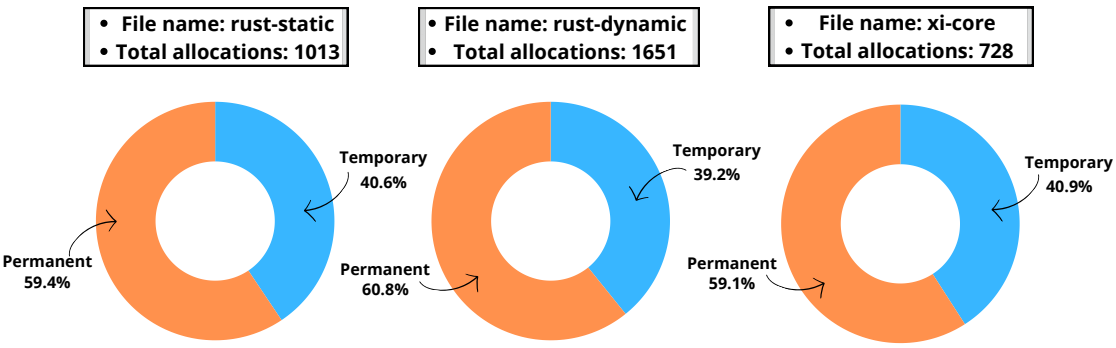

Figure 6.6: Rust binaries allocations.

In the Rust language landscape, there is a significant trend towards generating a large number of temporary allocations relative to total allocations, as visible from the data provided by the chart in Figure 6.6. This inclination suggests a

dynamic and security-oriented memory management model, which may require a greater number of temporary allocations to guarantee program correctness. In particular, rust-dynamic stands out for the highest percentage of temporary allocations, followed by rust-static and xi-core.

File	Size	Memory Peak	Peak (RSS)	Allocations	Temporary Allocations
rust-static	74.3 MB	76.4 MB	82.4 MB	1013	411
rust-dynamic	54.0 MB	55.3 MB	64.7 MB	1651	647
xi-core	74.6 MB	76.7 MB	85.9 MB	728	298

Table 6.3: Allocation and Memory Peak data of binaries written in Rust.

Among the Rust files considered, Table 6.3 highlights how rust-dynamic stands out for the highest peak memory, recording a value of 55.3MB. Rust-static and xi-core follow with memory peaks of 76.4MB and 76.7MB respectively. However, it is interesting to note that rust-static and xi-core exhibit significantly higher RSS peaks than rust-dynamic, reaching 82.4MB and 85.9MB respectively. According to the data, Rust-Static and xi-Core seem to utilize system resources more intensively, which could be attributed to the distinct management approach of the tool towards statically linked programs.

6.2.3 Programming language comparisons

- In the direct comparison among different programming languages, the dynamically linked firmware of libdevice have been examined, as they exhibit the same implemented features across all variants. The comparison between rust-dynamic, C-dynamic, and Cpp-dynamic files reveals significant differences. rust-dynamic shows the largest size at 54.0 MB, followed by C-dynamic at 18.2 MB and Cpp-dynamic at 7.3 MB. This variation may indicate differences in code optimization among the programming languages. Regarding execution times, Cpp-dynamic is the fastest at 10.7 ms, followed by C-dynamic at 15.8 ms and rust-dynamic at 43.6 ms. Interestingly, there is no direct correlation between file size and execution times. While the file size-to-execution time ratio in the case of the Cpp version may suggest increasing times with larger file sizes, this is not confirmed in the versions written in C and Rust, which exhibit accessible execution times despite their larger sizes. This suggests that factors such as code complexity and resource management significantly influence performance. Figure 6.7 shows the relationship between file size and execution time just described.

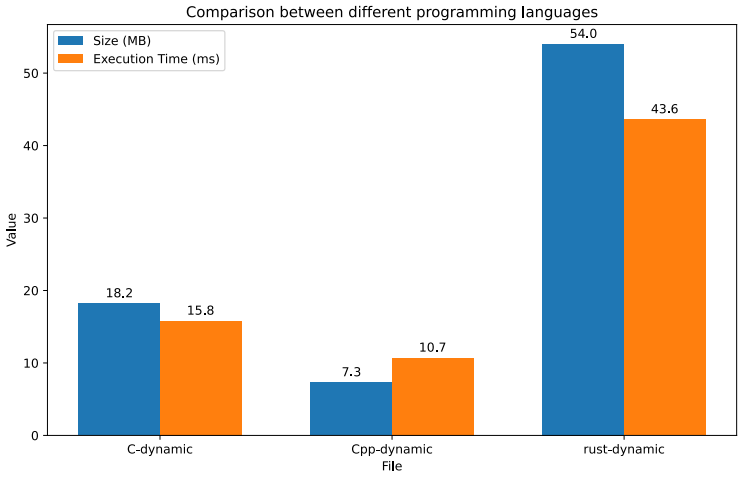

Figure 6.7: Comparison between libdevice variants

- The comparison between FFmpeg, OpenCV, and xi-core files aims to contrast their different implementations and functionalities, along with their programming languages. FFmpeg, with a file size of 409.9 kB, exhibits an execution time of 6.3 ms, while OpenCV, with a smaller file size of 177.3 kB, shows a faster execution time of 4.0 ms. In contrast, xi-core stands out with a significantly larger file size of 74.6 MB, resulting in a longer execution time of 34.7 ms. This disparity suggests that larger file sizes generally correspond

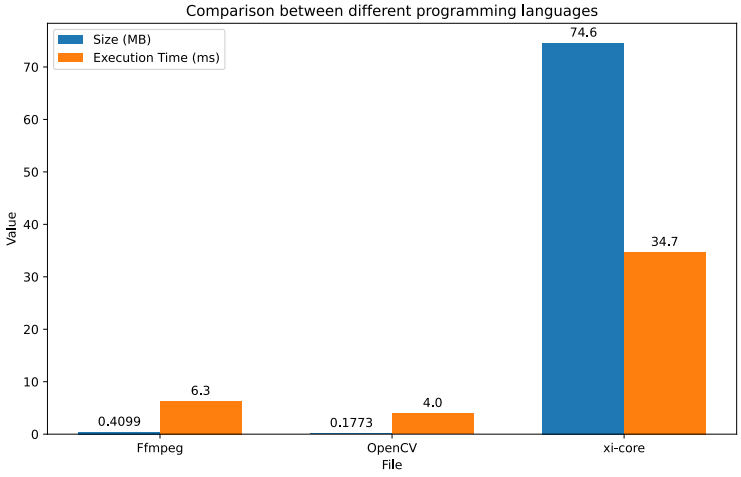

Figure 6.8: Comparison between real programs

to longer execution times. However, it is interesting to note that, similar to previous analyses, there is no significant increase in execution times as file sizes increase, contrary to the trend observed for FFmpeg and OpenCV files. The graph in Figure 6.8 shows this comparison.

- From the point of view of memory usage, it is interesting to note the trend represented by the data collected through the memory peak and peak RSS parameters. Despite the obvious differences in file sizes, a consistent pattern emerges showing uniform growth ratios in both heap memory usage and maximum physical memory usage. This trend is precisely illustrated in the graph in Figure 6.9. For example, although FFmpeg and OpenCV are relatively compact file sizes, their memory usage growth ratio is approximately equal to that of the largest file size, xi-core. This suggests that, regardless of file size, the tool tends to use memory consistently and predictably.

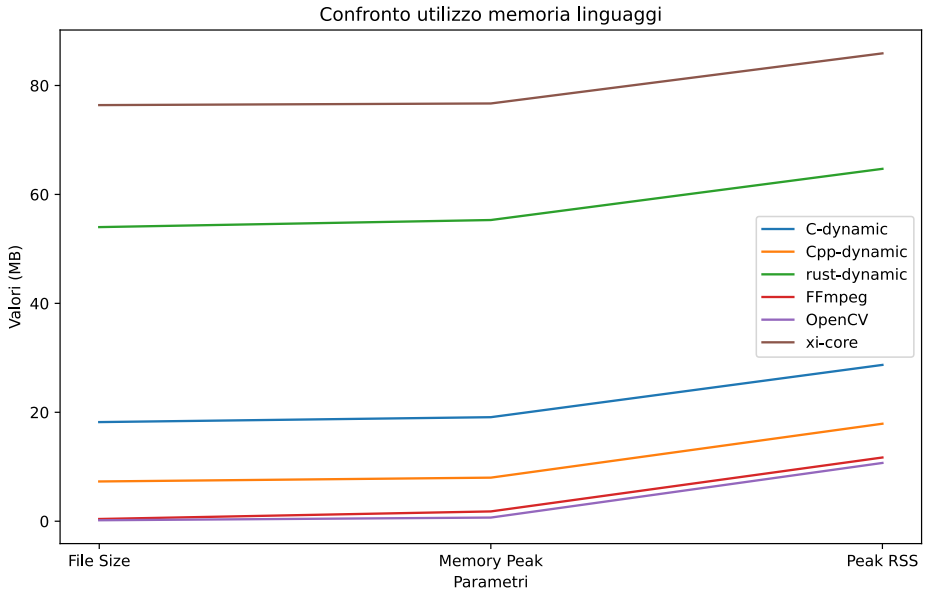

Figure 6.9: Memory usage comparison.

6.3 Final Remarks

The performance analysis of the manifest-producer tool has provided a comprehensive overview of its capabilities in analyzing a range of ELF files. The collected data has demonstrated that file sizes are not the determining factor of program execution times. For instance, in the case of files written in the C language, a significant variability in execution times has been observed, which was not directly proportional to file sizes.

Additionally, memory allocation analysis has revealed distinct resource utilisation patterns among different types of ELF files. For example, when comparing files written in C, C++, and Rust, it was observed that some exhibited a more efficient resource management than others. However, it was interesting to note that despite differences in programming languages and file sizes, the tool demonstrated uniformity in performance across various contexts. This suggests a high level of adaptability and robustness of the tool, capable of effectively handling various types of ELF files regardless of their specific characteristics.

Chapter 7

Conclusions

This thesis has focused on the intricate challenges faced by developers in the realm of IoT software development, with a particular emphasis on firmware for IoT devices. It has examined the criteria necessary to ensure the security and reliability of such software through certification processes.

A significant aspect highlighted by this research revolves around the paradigm of Continuous Integration (CI) in software development. The introduction of the ci-generate tool has shown new perspectives from developers' point of view, offering them the capabilities of automation, simplified configuration, and enhanced control over their software projects.

Moreover, firmware certification assumes a role in validating the assertions made by developers and guaranteeing the reliability of the software. It serves as a quality assurance mechanism, verifying that the firmware operates as intended.

Central to this endeavour is the manifest-producer, which stands as an instrument for conducting thorough evaluations of IoT firmware. A comprehensive analysis of the essential characteristics embedded within binary files, enables us to analyze firmware components. Through the utilization of static analysis techniques applied to ELF binaries, coupled with the generation of JSON format manifests, the manifest-producer provides a clear and comprehensive representation of the information required for firmware certification.

7.1 Future developments

To guide future work related to improving the manifest-producer tool, some significant tasks can be defined:

- **Integrate dynamic analysis** with the current static analysis of the tool to enrich the retrieved information and enhance the analysis's accuracy. A concrete example could be integrating strace functionalities to obtain the list of syscalls belonging to a concrete instance of the binary being examined.
- To enhance the tool's analytical capabilities, expanding its ability to understand and analyse a broad spectrum of **hardware architectures commonly used** in the context of IoT firmware is a possible way forward. This advancement would allow the tool to also examine and interpret binaries compiled for architectures other than those considered during its development, i.e. x86 and x86-64, thus ensuring a more comprehensive coverage among IoT devices.
- The tool has been tested to analyse firmware mainly written in languages such as C, C++, and Rust. However, it is important to note that there are **other programming languages** used in the creation of firmware for IoT devices, including Python, JavaScript, and Java. Therefore, to fully evaluate the current capabilities of the tool and identify any gaps or limitations in the handling of these languages, it is crucial to analyze firmware written in these languages. This would enable further development of the tool, improving its ability to analyse and understand a wider range of binaries, and thus making it more effective in the IoT firmware certification process.
- **Vulnerability analysis** targeting specific threats identified in IoT devices is a crucial approach to ensuring the security and resilience of such systems. This type of analysis requires a special focus on potential vulnerabilities related to firmware, which is one of the fundamental pillars of IoT devices. To develop effective research in this direction, it is essential to fully understand the common threats present in the IoT environment, as well as the risks associated with firmware vulnerabilities. This approach will enrich the analysis conducted by the manifest-producer tool, enabling the generation of a detailed report highlighting the various vulnerabilities identified and their associated risks.

Bibliography

[1] *What is Internet of Things (IoT)?* URL: https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT (cit. on p. 3).

[2] Basem Ibrahim Mukhtar, Mahmoud Said Elsayed, Anca D. Jurcut, and Marianne A. Azer. «IoT Vulnerabilities and Attacks: SILEX Malware Case Study». In: *Symmetry* 15.11 (2023). ISSN: 2073-8994. DOI: 10.3390/sym15111978. URL: https://www.mdpi.com/2073-8994/15/11/1978 (cit. on p. 3).

[3] Resul Das and Muhammed Gündüz. «Analysis of cyber-attacks in IoT-based critical infrastructures». In: *International Journal of Information Security* 8 (Dec. 2019), pp. 122–133 (cit. on p. 3).

[4] *The little-known story of the first IoT device.* URL: https://www.ibm.com/blog/little-known-story-first-iot-device/ (cit. on p. 4).

[5] *The Computer for the 21st Century.* URL: https://web.archive.org/web/20150311220327/http://web.media.mit.edu/~anjchang/ti01/weiser-sciam91-ubicomp.pdf (cit. on p. 4).

[6] R.S. Raji. *Smart networks for control.* eng. New York, 1994 (cit. on p. 5).

[7] *That 'Internet of Things' Thing.* URL: https://www.rfidjournal.com/that-internet-of-things-things (cit. on p. 5).

[8] Yvan Duroc. «From Identification to Sensing: RFID Is One of the Key Technologies in the IoT Field». eng. In: *Sensors (Basel, Switzerland)* 22.19 (2022), pp. 7523–. ISSN: 1424-8220 (cit. on p. 5).

[9] Shivayogi Hiremath, Geng Yang, and Kunal Mankodiya. «Wearable Internet of Things: Concept, architectural components and promises for person-centered healthcare». eng. In: *2014 4th International Conference on Wireless Mobile Communication and Healthcare - Transforming Healthcare Through Innovations in Mobile and Wireless Technologies (MOBIHEALTH)*. ICST, 2014, pp. 304–307. ISBN: 9781631900143 (cit. on p. 5).

[10] Hakilo Sabit, Peter Han Joo Chong, and Jeff Kilby. «Ambient Intelligence for Smart Home using The Internet of Things». eng. In: *2019 29th International Telecommunication Networks and Applications Conference (ITNAC)*. IEEE, 2019, pp. 1–3. ISBN: 1728136733 (cit. on p. 5).

[11] David Camacho and Paulo Novais. «Innovations and practical applications of intelligent systems in ambient intelligence and humanized computing». In: *Journal of Ambient Intelligence and Humanized Computing* 8.2 (Apr. 2017), pp. 155–156. ISSN: 1868-5145. DOI: 10.1007/s12652-017-0454-z. URL: https://doi.org/10.1007/s12652-017-0454-z (cit. on p. 5).

[12] Mrinai M. Dhanvijay and Shailaja C. Patil. «Internet of Things: A survey of enabling technologies in healthcare and its applications». eng. In: *Computer networks (Amsterdam, Netherlands : 1999)* 153 (2019), pp. 113–131. ISSN: 1389-1286 (cit. on p. 5).

[13] Ravneet K. Sidhu. «An Overview of IoT for Smart Healthcare Technologies». eng. In: *2023 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES)*. IEEE, 2023, pp. 1–7 (cit. on p. 5).

[14] Rahul Dagar, Subhranil Som, and Sunil Kumar Khatri. «Smart Farming – IoT in Agriculture». In: *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*. 2018, pp. 1052–1056. DOI: 10.1109/ICIRCA.2018.8597264 (cit. on p. 5).

[15] Ravi Ramakrishnan and Loveleen Gaur. *Internet of things : approach and applicability in manufacturing*. eng. Boca Raton: Taylor & Francis, a CRC title, part of the Taylor & Francis imprint, a member of the Taylor & Francis Group, the academic division of T&F Informa, plc, 2019. Chap. Smart Manufacturing, Logistics Optimization. ISBN: 0-429-48659-6 (cit. on p. 5).

[16] Christopher Clearfield. *Rethinking Security for the Internet of Things*. URL: https://hbr.org/2013/06/rethinking-security-for-the-in (cit. on p. 5).

[17] Warren Detres, Md Minhaz Chowdhury, and Nafiz Rifat. «IoT Security and Privacy». In: *2022 IEEE International Conference on Electro Information Technology (eIT)*. 2022, pp. 498–503. DOI: 10.1109/eIT53891.2022.9813933 (cit. on p. 5).

[18] Maryam Farsi, Alireza Daneshkhah, Amin Hosseinian-Far, and Hamid Jahankhani. «IoT Security, Privacy, Safety and Ethics». eng. In: *Digital Twin Technologies and Smart Cities*. Internet of Things. Switzerland: Springer International Publishing AG, 2020, pp. 123–149. ISBN: 3030187314 (cit. on p. 5).

[19] Jibran Saleem, Mohammad Hammoudeh, Umar Raza, Bamidele Adebisi, and Ruth Ande. «IoT standardisation-Challenges, perspectives and solution». eng. In: *ACM International Conference Proceeding Series*. 2018. ISBN: 1450364284 (cit. on pp. 5, 52).

[20] Alex Wood. *The internet of things is revolutionising our lives, but standards are a must*. 2015. URL: https://www.theguardian.com/media-network/2015/mar/31/the-internet-of-things-is-revolutionising-our-lives-but-standards-are-a-must (cit. on p. 5).

[21] Aaron Ardiri. *Will fragmentation of standards only hinder the true potential of the IoT industry?* 2014. URL: https://web.archive.org/web/20210227182106/https://evothings.com/will-fragmentation-of-standards-only-hinder-the-true-potential-of-the-iot-industry/ (cit. on p. 5).

[22] Carrie Mihalcik. *Apple, Amazon, Google, and others want to create a new standard for smart home tech.* 2019. URL: https://www.cnet.com/home/smart-home/apple-amazon-google-and-others-want-to-create-a-new-standard-for-smart-home-tech/ (cit. on p. 5).

[23] Sharu Bansal and Dilip Kumar. «IoT Ecosystem: A Survey on Devices, Gateways, Operating Systems, Middleware and Communication». eng. In: *International journal of wireless information networks* 27.3 (2020), pp. 340–364. ISSN: 1068-9605 (cit. on p. 6).

[24] S. R. Prathibha, Anupama Hongal, and M. P. Jyothi. «IOT Based Monitoring System in Smart Agriculture». eng. In: *2017 International Conference on Recent Advances in Electronics and Communication Technology (ICRAECT)*. IEEE, 2017, pp. 81–84. ISBN: 9781509067015 (cit. on p. 6).

[25] Vangelis Marinakis and Haris Doukas. «An Advanced IoT-based System for Intelligent Energy Management in Buildings». In: *Sensors* 18.2 (2018). ISSN: 1424-8220. DOI: 10.3390/s18020610. URL: https://www.mdpi.com/1424-8220/18/2/610 (cit. on p. 6).

[26] Wang Huifeng, Seifedine Nimer Kadry, and Ebin Deni Raj. «Continuous health monitoring of sportsperson using IoT devices based wearable technology». eng. In: *Computer communications* 160 (2020), pp. 588–595. ISSN: 0140-3664 (cit. on p. 6).

[27] Baichen Li, R. Scott Downen, Quan Dong, Nam Tran, Maxine LeSaux, Andrew C. Meltzer, and Zhenyu Li. «A Discreet Wearable IoT Sensor for Continuous Transdermal Alcohol Monitoring-Challenges and Opportunities». eng. In: *IEEE sensors journal* 21.4 (2021), pp. 5322–5330. ISSN: 1530-437X (cit. on p. 6).

[28] Siying Qian. «IoT Application with Tortoise Smart Home». eng. In: *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 2021, pp. 541–547. ISBN: 1665421746 (cit. on p. 7).

[29] Mustafa A. Omran, Bashar J. Hamza, and Wasan K. Saad. «The design and fulfillment of a Smart Home (SH) material powered by the IoT using the Blynk app». In: *Materials Today: Proceedings* 60 (2022). International Conference on Latest Developments in Materials & Manufacturing, pp. 1199–1212. ISSN: 2214-7853. DOI: https://doi.org/10.1016/j.matpr.2021.08.038. URL: https:

//www.sciencedirect.com/science/article/pii/S2214785321054663 (cit. on p. 7).

[30] Daniel Rodrigues, Paulo Carvalho, Solange Rito Lima, Emanuel Lima, and Nuno Vasco Lopes. «An IoT platform for production monitoring in the aerospace manufacturing industry». In: *Journal of Cleaner Production* 368 (2022), p. 133264. ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2022.133264. URL: https://www.sciencedirect.com/science/article/pii/S0959652622028487 (cit. on p. 7).

[31] Wei Chen. «Intelligent manufacturing production line data monitoring system for industrial internet of things». In: *Computer Communications* 151 (2020), pp. 31–41. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2019.12.035. URL: https://www.sciencedirect.com/science/article/pii/S0140366419315518 (cit. on p. 7).

[32] S. Sakena Benazer, M. Sheik Dawood, Sulochanan Karthick Ramanathan, and G. Saranya. «Efficient model for IoT based railway crack detection system». In: *Materials Today: Proceedings* 45 (2021). International Conference on Advances in Materials Research - 2019, pp. 2789–2792. ISSN: 2214-7853. DOI: https://doi.org/10.1016/j.matpr.2020.11.743. URL: https://www.sciencedirect.com/science/article/pii/S2214785320394190 (cit. on p. 7).

[33] Lubna, Naveed Mufti, Sadiq Ullah, Abubakar Sharif, Muhammad Waqas Nawaz, Ahmed Alkhayyat, Muhammad Ali Imran, and Qammer H. Abbasi. «IoT enabled vehicle recognition system using inkjet-printed windshield tag and 5G cloud network». In: *Internet of Things* 23 (2023), p. 100873. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2023.100873. URL: https://www.sciencedirect.com/science/article/pii/S2542660523001968 (cit. on p. 7).

[34] Shahrokh Nikou. «Factors driving the adoption of smart home technology: An empirical assessment». In: *Telematics and Informatics* 45 (2019), p. 101283. ISSN: 0736-5853. DOI: https://doi.org/10.1016/j.tele.2019.101283. URL: https://www.sciencedirect.com/science/article/pii/S0736585319307750 (cit. on p. 10).

[35] Erfaneh Allameh, Mohammadali Heidari Jozam, Bauke de Vries, Harry JP Timmermans, and Jakob Beetz. «Smart Home as a smart real estate, A state of the art review». eng. In: *IDEAS Working Paper Series from RePEc* (2011) (cit. on p. 10).

[36] Ehsan Kamel and Ali M Memari. «State-of-the-Art Review of Energy Smart Homes». eng. In: *Journal of architectural engineering* 25.1 (2019). ISSN: 1076-0431 (cit. on p. 10).

[37] T Serrenho and P Bertoldi. «Smart home and appliances: state of the art : energy, communications, protocols, standards». eng. In: 29750 (2019). ISSN: 1831-9424 (cit. on p. 10).

[38] Mehmet Buyuk, Ercan Avşar, and Mustafa İnci. «Overview of smart home concepts through energy management systems, numerical research, and future perspective». eng. In: *Energy sources. Part A, Recovery, utilization, and environmental effects* ahead-of-print.ahead-of-print (2022), pp. 1–26. ISSN: 1556-7036 (cit. on p. 11).

[39] Jyoti Neeli, Smithu B S, Ravi Teja S, and Layan N. «Comparative Study Using IoT to Automate Your Home». In: *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. 2023, pp. 1–7. DOI: 10.1109/CSITSS60515.2023.10334211 (cit. on p. 11).

[40] Leong Yee Rock, Farzana Parveen Tajudeen, and Yeong Wai Chung. «Usage and impact of the internet-of-things-based smart home technology: a quality-of-life perspective». In: *Universal Access in the Information Society* (Nov. 2022). ISSN: 1615-5297. DOI: 10.1007/s10209-022-00937-0. URL: https://doi.org/10.1007/s10209-022-00937-0 (cit. on p. 11).

[41] Mohamad Razwan Abdul Malek, Nor Azlina Ab. Aziz, Salem Alelyani, Mohamed Mohana, Farah Nur Arina Baharudin, and Zuwairie Ibrahim. «Comfort and energy consumption optimization in smart homes using bat algorithm with inertia weight». In: *Journal of Building Engineering* 47 (2022), p. 103848. ISSN: 2352-7102. DOI: https://doi.org/10.1016/j.jobe.2021.103848. URL: https://www.sciencedirect.com/science/article/pii/S235271022101706X (cit. on p. 11).

[42] Brown Eric. *Who Needs the Internet of Things?* 2016. URL: https://www.linux.com/news/who-needs-internet-things/ (cit. on p. 12).

[43] Iqra Asghar, Muhammad Ayaz Khan, Tahir Ahmad, Subhan Ullah, Khwaja Mansoor ul Hassan, and Attaullah Buriro. «Fortifying Smart Home Security: A Robust and Efficient User-Authentication Scheme to Counter Node Capture Attacks». eng. In: *Sensors (Basel, Switzerland)* 23.16 (2023), pp. 7268–. ISSN: 1424-8220 (cit. on p. 12).

[44] Noureddine Amraoui and Belhassen Zouari. «Securing the operation of Smart Home Systems: a literature review». eng. In: *Journal of reliable intelligent environments* 8.1 (2022), pp. 67–74. ISSN: 2199-4668 (cit. on p. 12).

[45] Yan Meng, Wei Zhang, Haojin Zhu, and Xuemin Sherman Shen. «Securing Consumer IoT in the Smart Home: Architecture, Challenges, and Countermeasures». eng. In: *IEEE wireless communications* 25.6 (2018), pp. 53–59. ISSN: 1536-1284 (cit. on p. 12).

[46] Martin Fowler and Matthew Foemmel. *Continuous integration*. 2006 (cit. on p. 13).

[47] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. «Uncovering the Benefits and Challenges of Continuous Integration Practices». In: *IEEE Transactions on Software Engineering* 48.7 (2022), pp. 2570–2583. DOI: 10.1109/TSE.2021.3064953 (cit. on p. 14).

[48] Michal R. Wróbel, Jaroslaw Szymukowicz, and Pawel Weichbroth. «Using Continuous Integration Techniques in Open Source Projects—An Exploratory Study». In: *IEEE Access* 11 (2023), pp. 113848–113863. DOI: 10.1109/ACCESS.2023.3324536 (cit. on p. 14).

[49] Hao Chen, Jinxin Ma, Baojiang Cui, and Junsong Fu. «IoTCID: A Dynamic Detection Technology for Command Injection Vulnerabilities in IoT Devices». eng. In: *International journal of advanced computer science & applications* 13.10 (2022), pp. 7–14. ISSN: 2158-107X (cit. on pp. 15, 19).

[50] Haitham Ameen Noman and Osama M. F. Abu-Sharkh. «Code Injection Attacks in Wireless-Based Internet of Things (IoT): A Comprehensive Review and Practical Implementations». In: *Sensors* 23.13 (2023). ISSN: 1424-8220. DOI: 10.3390/s23136067. URL: https://www.mdpi.com/1424-8220/23/13/6067 (cit. on p. 15).

[51] James Jin Kang, Kiran Fahd, Sitalakshmi Venkatraman, Rolando Trujillo-Rasua, and Paul Haskell-Dowland. «Hybrid Routing for Man-in-the-Middle (MITM) Attack Detection in IoT Networks». In: *2019 29th International Telecommunication Networks and Applications Conference (ITNAC)*. 2019, pp. 1–6. DOI: 10.1109/ITNAC46935.2019.9077977 (cit. on p. 15).

[52] Alessandra Alvarez Olazabal, Jasmeet Kaur, and Abel Yeboah-Ofori. «Deploying Man-In-the-Middle Attack on IoT Devices Connected to Long Range Wide Area Networks (LoRaWAN)». In: *2022 IEEE International Smart Cities Conference (ISC2)*. 2022, pp. 1–7. DOI: 10.1109/ISC255366.2022.9922377 (cit. on p. 15).

[53] Wei Xie, Yikun Jiang, Yong Tang, Ning Ding, and Yuanming Gao. «Vulnerability Detection in IoT Firmware: A Survey». In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. 2017, pp. 769–772. DOI: 10.1109/ICPADS.2017.00104 (cit. on pp. 15, 19).

[54] Xiaotao Feng, Xiaogang Zhu, Qing-Long Han, Wei Zhou, Sheng Wen, and Yang Xiang. «Detecting Vulnerability on IoT Device Firmware: A Survey». In: *IEEE/CAA Journal of Automatica Sinica* 10.1 (2023), pp. 25–41. DOI: 10.1109/JAS.2022.105860 (cit. on p. 15).

[55] Ibrahim Nadir, Haroon Mahmood, and Ghalib Asadullah. «A taxonomy of IoT firmware security and principal firmware analysis techniques». In: *International Journal of Critical Infrastructure Protection* 38 (2022), p. 100552. ISSN: 1874-5482. DOI: https://doi.org/10.1016/j.ijcip.2022.100552. URL: https://www.sciencedirect.com/science/article/pii/S1874548222000373 (cit. on p. 15).

[56] «IEEE Standard for Intelligent Electronic Devices Cyber Security Capabilities - Redline». In: *IEEE Std 1686-2013 (Revision of IEEE Std 1686-2007) - Redline* (2014), pp. 1–49 (cit. on p. 17).

[57] Yuan Cheng, Baojiang Cui, Chen Chen, Thar Baker, and Tao Qi. «Static vulnerability mining of IoT devices based on control flow graph construction and graph embedding network». eng. In: *Computer communications* 197 (2023), pp. 267–275. ISSN: 0140-3664 (cit. on p. 19).

[58] Pengfei Sun, Luis Garcia, Gabriel Salles-Loustau, and Saman Zonouz. «Hybrid Firmware Analysis for Known Mobile and IoT Security Vulnerabilities». In: *2020 50th Annual IEEEIFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 373–384. DOI: 10.1109/DSN48063.2020.00053 (cit. on p. 19).

[59] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. «AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation». In: *Computers & Security* 122 (2022), p. 102889. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102889. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002838 (cit. on p. 19).

[60] Valentina Forte, Nicolò Maunero, Paolo Prinetto, and Gianluca Roascio. «PROLEPSIS: Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity». In: *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. 2021, pp. 1–6. DOI: 10.1109/ICECCME52200.2021.9591080 (cit. on p. 20).

[61] *Continuous Integration*. URL: https://en.wikipedia.org/wiki/Continuous_integration#:~:text=In%20software%20engineering%2C%20continuous%20integration%20(CI)%20is%20the%20practice%20of%20merging%20all%20developers%27%20working%20copies%20to%20a%20shared%20mainline%20several%20times%20a%20day.%5B1%5D%20Nowadays%20it%20is%20typically%20implemented%20in%20such%20a%20way%20that%20it%20triggers%20an%20automated%20build%20with%20testing. (cit. on p. 24).

[62] *Traits: Defining Shared Behaviour*. URL: https://doc.rust-lang.org/book/ch10-02-traits.html (cit. on p. 30).

[63] Atica Mohammed, Rasha Alsarraj, and Asmaa Albayati. «VERIFICATION AND VALIDATION OF A SOFTWARE: A REVIEW OF THE LITERATURE». In: *Iraqi Journal for Computers and Informatics* 46 (June 2020), pp. 40–47. DOI: 10.25195/ijci.v46i1.249 (cit. on p. 33).