POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Smart Home Devices: Firmware Analysis and Certification

Supervisors

Prof. Luca ARDITO

Dr. Michele VALSESIA

Candidate

Marian Alexandru LEONTE

July 2024

Abstract

An Internet of Things (IoT) system is a network of interconnected devices. These devices range from simple sensors or actuators with limited computational capabilities to devices with higher processing power. The behaviour of a device is driven by the firmware, software which interacts with the device's hardware and performs some actions.

An example of an IoT system is the Smart Home, which is equipped with devices that enhance the quality of life of its inhabitants.

The rapid growth of IoT systems comes with challenges such as security, standardisation and a lack of proper certification.

Regarding IoT systems security, our research has mainly focused on firmware security. Many firmware are designed without respecting security and programming best practices, thus exposing devices to attacks. We concentrated on static and dynamic firmware analysis. We propose two static analysis tools that could be integrated into firmware analysis: weighted-code-coverage and complex-code-spotter.

We also tried to address the gap in firmware certification. Indeed, there is a lack of well-structured certification processes to validate devices in terms of behaviour and security and to inform users about their potential dangers. We have focused on Smart Home firmware devices' behaviour and defined the hazard concept. A hazard can indicate a potential safety, privacy, and financial risk associated with the execution of determined devices' actions within a house.

To partly address this issue, we have developed hazard-generator and code-certifier. These tools can be used in Smart Home firmware development and integrated into a broader certification process to inform and certify about the risks associated with devices' actions within a Smart Home.

The implemented tools have been highly tested and written in Rust, a new programming language we have chosen for its security, optimisation and performance aptitude.

In the first part of the thesis, we provide an overview of the state of the art of IoT systems. Afterwards, we discuss the most significant challenges and security threats, presenting potential solutions from the literature.

In the second part, we present the tools we have been working on. To address the firmware analysis problem, we have added some features to the static analysis tools mentioned previously. weighted-code-coverage implements three new software quality metrics. complex-code-spotter extracts overly complex snippets of code.

To address the gap in the description of Smart Home devices' behaviour, we have developed the hazard-generator. This tool receives a hazard ontology as input and generates the API necessary to describe device hazards. The Rust library used to create firmware, which incorporates this API, aims to enhance the classification of devices' behaviour while reducing firmware developer effort.

To partially fill the current gap in device behaviour certification, we have developed the code-certifier software, which provides two main functionalities. The first one extracts all public APIs from the Rust library used for Smart Home firmware development. The second functionality takes the source code of a Smart Home firmware as input and generates as output a manifest which lists all the devices contained in the firmware. This manifest provides a detailed description of each device, outlining its actions and associated hazards.

The thesis concludes with a performance analysis in terms of time and memory of some of the developed tools.

Acknowledgements

"Ai miei genitori,
per il loro costante supporto e
fondamentale presenza durante l'intero percorso."
Alex

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we have started by analysing the Internet of Things (IoT) ecosystem in order to identify the numerous challenges and problems that arised from the rapid evolution of this technology.

Specifically, we have focused on the Smart Home system, one of the most popular IoT application domains. We have found that the lack of proper certifications for Smart Home device firmware is a major contributing factor to their insecurity and unreliability. In fact, firmware is critical as it controls all the device's actions and defines its behaviour. Therefore, it is essential to establish rigorous processes for comprehensive firmware certification.

In this context, we first present two tools that can perform a static analysis of firmware to evaluate its software quality.

Subsequently, we introduce the concept of hazard, which refers to a danger that may arise from an action performed by a Smart Home Device. One of the tools we have implemented allows the integration of this concept into a Smart Home device firmware development library. This enables developers to explicitly define that an action may be characterised by one or more hazards. We then provide a tool that analyses a firmware created using the library and generates a comprehensive description of all the actions and hazards of a device.

This thesis is divided into various chapters according to the following structure:

- **IoT Evolution and Challenges**: in this initial chapter, we provide an overview of the Internet of Things concept, including how this system operates and the main events and technological advancements that have contributed to its development. We also discuss the primary application domains where

IoT principles and concepts are applied, with a focus on the Smart Home system. Furthermore, we list the major issues and challenges related to standardization, interoperability, reliability, and security within these systems. Regarding security, we outline some of the solutions presented in the literature, with a focus on firmware analysis through static and dynamic analysis. Finally, we highlight a significant problem within the IoT ecosystem: the absence of rigorous certification processes for the firmware device behaviour.

- **Firmware Analysis Tools**: in this chapter, we introduce two static analysis tools we have contributed to: `weighted-code-coverage` and `complex-code-spotter`. The first tool implements three new metrics for evaluating software quality, which combine the concepts of code coverage and complexity. The second one, instead, extracts complex code snippets based on specific thresholds, helping developers identify code that may be challenging to maintain. These tools represent an example of static analysis tools that can be used to analyse the firmware of an IoT device. In particular, we list the modifications and new features that we have implemented in these tools, highlighting the improvements compared to their initial version.

- **Hazar Generator**: in this chapter, we introduce the first tool we have developed, with the objective of addressing the lack of certification of a Smart Home device's behaviour. We named this tool `hazard-generator`, and starting from an ontology of hazards, it generates an API that can be integrated within a library for firmware development in order to assign hazards to the actions of a Smart Home device.

- **Code Certifier**: in this chapter, we present `code-certifier`, our second tool, which consists of two libraries: `pub-api` and `hazard-analyzer`. `pub-api` enables the extraction of all public API from two libraries, which can be used for the development of a Smart Home device firmware. `hazard-analyzer`, on the other hand, analyzes those kinds of firmware implemented using these two libraries with the goal of extracting a manifest that lists all device actions and associated hazards. We then highlight the fact that a firmware manifest, produced by this second library, could be integrated within a broader certification process in order to help delineate the behaviour and hazards of a Smart Home device.

- **Performance Analysis**: in this chapter, we present the performance analysis in terms of execution time and memory usage that we have conducted on `weighted-code-coverage`, `complex-code-spotter` and `hazard-analyzer` binaries.

- **Conclusions**: in the last chapter of our thesis, we have summarized the results and final considerations about the analysis and implementations carried out during this entire thesis, also highlighting possible future developments.

Chapter 2

IoT Evolution and Challenges

This chapter introduces the Internet of Things and Smart Home concepts, outlining their evolution and some of their main applications. It also highlights challenges faced by these systems, such as reliability and interoperability, with a detailed focus on security aspects.

2.1 Internet of Things

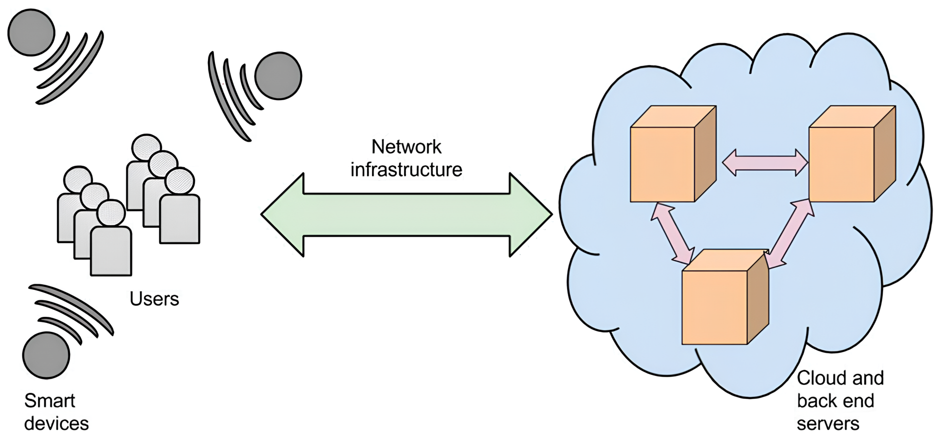


Figure 2.1: Internet of Things structure.

The term **Internet of Things** (**IoT**) refers to a system consisting of smart devices capable of independently interacting with the physical world around them [1]. These devices are interconnected to each other and can communicate with additional components present in the system. They can exchange information or perform specific actions in response to external feedback or commands issued by a customer. An example could be a smart thermostat that adjusts the temperature in a house based on current weather conditions and time of day. Additionally, smart devices can share their data with external services, such as a Cloud[1] service, for processing or storage purposes.

Figure 2.1 shows the essential components of an Internet of Things system [2]. These components include smart devices equipped with sensors and actuators to gather information from the surroundings and interact with it, a communication network, typically based on Internet protocols, and a Cloud platform.

The term *Things* contained in the Internet of Things noun refers to smart devices. Despite their importance in the system, there is no universally accepted definition of a smart device. Nevertheless, we can still identify some fundamental properties that these devices must have [3]:

- **Autonomy**: after an eventual initial configuration, a smart device should be able to perform its tasks independently, without continuous user intervention.
- **Connectivity**: in order to interact with other entities within an IoT system, a device must be able to connect to the communication network. Although wired connections are possible, wireless connections are becoming more and more prevalent.
- **Context-awareness**: a device must have self-awareness of the context and the environment in which it is installed, enabling it to gather and utilize data accordingly.

Other characteristics which are often associated with smart devices are *User Interaction* and *Portability*. However, these properties primarily pertain to devices such as smartphones, computers, or smartwatches. These devices represent only a small part of an IoT system, the one related to user control. Therefore, they are not considered fundamental characteristics, since many IoT devices are neither portable nor designed for direct user interaction.

Starting from the *Connectivity* property, we can gain a deeper understanding of the role of the other term that composes the name Internet of Things: Internet. The Internet is a global telecommunications infrastructure that facilitates data and

[1]A Cloud service consists of an online platform that provides computing resources like storage or processing power, allowing users to access and use these resources without needing to own or manage physical hardware.

information exchange among its members. This concept of a vast, shared network for communication and digital services was first theorized by MIT professor J.C.R. Licklider as the *Galactic Network* [4]. The Internet is crucial for the IoT ecosystem, as it serves as the primary communication channel among all of its elements. It enables devices to interact with each other and with users, also allowing a remote control from outside the local network. Additionally, it allows an IoT subsystem to take advantage of network services and interact with Cloud platforms.

2.1.1 History

In 1999, during a presentation at Procter & Gamble, Kevin Ashton coined the term Internet of Things [5]. He used this term to predict a shift in the interaction between humans and machines. Ashton foresaw a future where machines would interact more directly between them and with the surrounding environment, rather than relying on human intervention [6].

This new type of interaction, later referred to as *Machine-to-Machine* (M2M) communication [7], is a fundamental principle of the Internet of Things system. M2M communication involves creating a network that enables communication between diverse devices through standardized rules and protocols.

Until then, humans acted as the conduit between machines and the physical world. Machines lacked direct knowledge of their surroundings, relying on data provided by humans. This information could be transmitted via audio, video, images or through data collected by other tools and shared with machines. Rather than observing the physical world first-hand, machines received descriptions of it through human input.

Ashton highlighted the limitations of this approach. He pointed out the cost and speed advantages, in terms of collection, processing, and transmission, of an alternative where devices themselves directly gather and exchange the information.

Humans cannot compete with the efficiency, accuracy, and consistency of operation of a sensor, nor can they react to feedback as quickly as an actuator. Even then, the direction was clear: to create an ecosystem that minimizes human intervention and the errors that result from it, promoting automation and the continuous technological evolution of devices.

[8] The Internet of Things initially saw its initial growth and adoption in the enterprise and industrial sectors. In these areas, it quickly demonstrated its potential to accelerate and monitor business processes efficiently and consistently. Over time, IoT technology became popular with the general public by improving the quality of life for individuals.

Beyond Ashton's coining of the term Internet of Things, numerous innovations over the years have contributed to the development of the IoT ecosystem as we know it today. These innovations include tools and projects that emerged to meet specific needs, as well as breakthroughs that have permanently changed the telecommunications field.

We have therefore reconstructed a timeline of the key events [9] [10]:

- **1982**: in Pittsburgh, Pennsylvania, at Carnegie Mellon University, some researchers connected a Coke machine to the university's network. This was later named *The Only Coke Machine on the Internet* [11]. The purpose was to allow individuals to check the availability of Coke before heading to the room where the machine was located, ensuring they would not find it empty. This machine is considered to be the first non-computer device connected to the Internet [12].
- **1989**: at CERN in Switzerland, British computer scientist Tim Berners-Lee proposed developing software, standards, and protocols to improve and streamline the communication and sharing of scientific material within the organization and with other institutes using the Internet [13]. This suggestion laid the foundation for the World Wide Web (WWW) [14]. In the context of IoT, the *Web of Things* concept (WoT) [15] refers to a seamless integration of IoT-related technologies into the WWW, allowing direct interaction and control of smart devices through web interfaces and applications [16].
- **1990**: While working on the SNMP protocol, computer scientist John Romkey used it to control a toaster, making it the first toaster connected to the Internet [17]. SNMP (Simple Network Management Protocol) manages devices over an IP network, mainly to control and monitor their status [18]. This experiment demonstrated that protocols previously adopted only for computing and networking could also be applied to control Zphysical devices, a groundbreaking concept at the time [19].
- **1993**: Researchers at the University of Cambridge in England created the *Trojan Room Coffee Pot* [20] to allow those who were far from the coffee pot's location to check the coffee level status. The system included a camera, a server, and a client, and it provided employees with images with an update rate of approximately 20 seconds. This is presumably the first example of a webcam [21].
- **1997**: technology forecaster Paul Saffo predicted that the sensors' integration with the Internet would have been the next revolution in information technology, with a similar impact to the birth of microprocessors and the Internet itself [22]. This prediction is proving accurate, as data collection through

device sensors and sharing via the Internet are core concepts of the Internet of Things.

- **2000**: LG launched the first Internet-connected refrigerator [23]. Despite offering innovative services like memo writing and monitoring product freshness, it was not very successful. Customers found the features impractical for daily use and the price unjustifiably high [24]. This highlighted the need that technologies must address real customers' needs to succeed in the market.

- **2003**: RFID technology began to be adopted by the U.S. Department of Defense [25]. RFID (Radio Frequency Identification) allows objects to be identified, monitored, and controlled via radio waves by reading a tag with a reader [26]. The tag acts as a transponder, emitting a signal in response to a received signal [27]. RFID technology, giving objects a digital and unique identity, is a precursor to the Internet of Things [28].

- **2005**: the International Telecommunication Union (ITU) released its first report on the Internet of Things [29]. The report highlighted that the evolution of technologies such as RFID and sensors, along with network advancements, would lead to a world where all objects are connected, simplifying data processing and encouraging data sharing [30]. It also addressed the social and economic impacts and challenges of IoT [31].

- **2008**: Zurich, Switzerland, hosted the first international conference dedicated to the Internet of Things [32]. The conference aimed to encourage knowledge exchange and present the current state of IoT technologies and their applications [33]. The same year, the Internet Protocol for Smart Objects (IPSO) Alliance [34] was founded to promote the use of the IP protocol for smart device communication, advocating for its adoption in a simplified version suitable for environments with battery and power consumption limitations [35, 36].

- **2009**: Cisco suggests that IoT was born between 2008 and 2009. This period marked the surpassing of Internet-connected devices over connected people, reaching 12.5 billion connected devices by 2010 [37]. Meanwhile, the European Commission released an action plan emphasizing the importance of security and privacy to increase public trust in IoT and promote its adoption within the European community [38].

- **2012**: the World IPv6 Launch Day followed the previous year's World IPv6 Day [39]. These events aimed to test and promote IPv6 adoption as the successor to IPv4, with major companies like Google and Facebook making their sites accessible via IPv6 [40]. These two events were both successful, accelerating IPv6 adoption by even more sites [41]. IPv6 is crucial for IoT as it addresses the issue of limited IP addresses in IPv4, necessary for the growing number of smart devices [42]. The transition to IPv6 will be gradual, ensuring compatibility and interoperability with IPv4 systems [43].

- **2015 - 2016**: Cloud platforms such as Amazon's AWS IoT [44] and Microsoft's Azure IoT [45] emerged, providing services for building and managing IoT systems. These platforms facilitate device connections, data storage and data processing. They play a crucial role in the IoT ecosystem by ensuring security and continuous monitoring, as well as supporting devices that cannot handle the vast amount of data autonomously [46].
- **2020**: for the first time, IoT connections to the Internet surpassed non-IoT connections. A study by IoT Analytics [47] indicated that in 2020, IoT connections reached 11.7 billion, compared to 9.9 billion non-IoT connections, accounting for 54% of the total amount of Internet connections.
- **Future**: The Internet of Things is expected to continue growing. According to Transforma Insights [48], IoT connections will increase from an estimated 18.2 billion by the end of this year to 32.5 billion by 2030 and nearly 40 billion by 2033. Another report from GSMA [49] suggests an even faster growth, predicting more than 38 billion IoT connections by 2030.

2.1.2 Application Domains

The emergence of the Internet of Things has sparked significant interest and curiosity about its potential applications, leading to its expansion into various sectors.

This expansion aims to enhance people's lives by creating devices that simplify daily tasks and activities, as well as streamline industrial and business processes. The technology is leveraged to enhance customer experience, provide better services and products, and, at the same time, increase companies' economic returns.

It is thus possible to state that the growth of the Internet of Things market is unlikely to halt in the short term; instead, the opposite is happening [50]. Investments are increasing exponentially, driven by the healthcare, lifestyle, and energy sectors. However, there are several obstacles to this expansion. For example, communication technologies may not be as effective over long distances, and smart devices have power and energy consumption limitations, leading to reduced computing capacity and data storage.

The following list offers an overview of the main Internet of Things application domains, excluding the Smart Home, which will be discussed in more detail later [51] [52]:

- **Smart cities**: as urban populations continue to grow, the current infrastructure, often outdated and inefficient, will face increasing challenges in meeting the needs of residents. It is imperative for cities, both present and future, to leverage IoT technologies to enhance infrastructure efficiency while reducing costs. For instance, implementing a smart water delivery system can minimize leakage and enable citizens to monitor their water consumption easily. Similarly, deploying an intelligent lighting system that utilizes sensors to adjust lighting according to natural light levels can significantly contribute to energy conservation. Moreover, the adoption of sensor-based waste management systems to signal when bins are full can optimize waste collection processes, while sensors monitoring air and water quality can swiftly detect contamination incidents, thus mitigating the environmental impact.

- **Healthcare**: in this context, we can talk about a real ecosystem known as the Internet of Healthcare Things (IoHT). This encompasses patients, the devices used by patients, and the interfaces and platforms used to control those devices. It advances beyond past efforts of digitizing patient data. In fact, instead of merely digitizing information, it involves implementing tangible devices to assist healthcare providers in streamlining their tasks and effectively managing the escalating patient volumes. An illustrative application of the IoT paradigm in the healthcare domain involves using devices to monitor the health status of patients with specific diseases. These devices can transmit real-time notifications to designated caretakers in emergency situations and gather data to facilitate diagnostic procedures. These smart solutions can also significantly reduce healthcare costs by minimizing the need for hospitalizations and visits, thus making healthcare more accessible.

- **Energy**: the energy domain is another application area where IoT has achieved widespread use, known as the Energy Internet of Things (EIoT). Internet of Things (IoT) technologies enable us to meet various needs in the energy sector. This includes intelligently combining renewable and fossil energies, prioritizing renewable sources whenever possible and using fossil fuels only when necessary. Additionally, IoT allows for the efficient delivery of energy, responding promptly to demand while minimizing waste. Moreover, analogical monitoring devices can be replaced with smart devices, enabling remote consumption detection and real-time fault reporting. This speeds up repair times and enhances the reliability of the entire system.

- **Transportation and Logistics**: smart transportation vehicles equipped with actuators and sensors can make transportation more efficient and safer by using the data they collect. The vehicles can connect to each other and to the internet, allowing them to adjust dynamically to traffic conditions.

Implementing a smart traffic light management system can further improve this. In logistics, instead, IoT serves as a valuable tool for monitoring critical stages, encompassing storage, distribution, sales, transportation, customer service, and returns. Furthermore, during the transport phase, systems must monitor the condition of food products, ensuring specific environmental conditions during transfer.

- **Agriculture**: in this field, devices and systems have been purposefully designed to automate essential processes such as fertilizer distribution and irrigation. These innovations utilize actuators that respond to data collected by sensors and then processed to match weather and soil conditions. This technological advancement has significantly streamlined farm operations by minimizing human error and increasing precision. Additionally, the development of autonomous tractor models has drastically reduced the need for farmer intervention. Moreover, the strategic use of drones in agriculture has transformed traditional tasks, making them more efficient and less resource-intensive.
- **Smart environments**: the most popular domains where the Internet of Things has expanded are those designed to enhance our quality of life by simplifying and assisting in our daily activities. In addition to Smart Homes which we will discuss later, there are instances of Smart Offices to monitor and improve working conditions, as well as Smart Gyms and Smart Museums, where IoT is increasingly utilized for entertainment purposes.

When discussing Internet of Things applications, it is essential to distinguish between industrial environments and sectors such as healthcare or transport, as opposed to other IoT application domains. The former category falls under the Industrial Internet of Things (IIoT) [53], wherein failures and errors can result in critical safety and security issues on a large scale. Although similar situations may occur in non-IIoT systems, these tend to be more specific and narrowly defined scenarios.

2.2 Smart Home

As we previously stated, the interest in the Internet of Things is growing steadily. This continuous evolution has led to the integration of IoT technologies into everyday life, with the Smart Home serving as a representative example.

Cisco estimated that by 2023, approximately half of all Internet-connected devices will be Internet of Things devices [54]. Smart Home devices represent the majority of those IoT devices. A study [55] from 2021 stated that, at that time, 822.6

million Smart Home devices had been installed installed worldwide. This is nearly double the number of devices in the second-ranking application domain, Smart Cities, which had 450 million devices. Therefore, we can reasonably say that the Smart Home is certainly one of the most popular and successful applications of the Internet of Things paradigm [56].

There are multiple definitions of a Smart Home. A general definition is that a Smart Home is an Internet of Things ecosystem where household processes, such as adjusting a room temperature or monitoring home surveillance, are automated through the use of smart devices [57] [58]. Some of the goals of a Smart Home include simplifying those processes, enhancing their safety, reducing their energy consumption and environmental impact, improving home security, and enhancing residents' quality of life [59].

Connectivity is a key aspect for these definitions, as Smart Home devices should be connected to each other and to the Internet in order to be controlled from outside a home and collaborate appropriately [60]. However, there are vendors who label their products as Smart Home devices even if they lack connectivity. Those devices only enhance certain aspects and features of traditional appliances, without creating a true Smart Home network. Such definitions are inaccurate and are often used for marketing purposes. In fact, as we mentioned earlier when listing the characteristics of smart devices, connectivity is one of the fundamental traits that a smart device must have.

2.2.1 History

The evolution of Smart Homes can be divided into three distinct stages [61]:

- **1990s**: this decade saw the widespread adoption of broadband Internet, which facilitated the development of the first home automation projects. This involved the concept of setting up networks to control and automate various home activities and operations.

- **2000s**: this period marked the advent of smartphones and the proliferation of applications. These technological advancements added a new dimension to home automation, enabling remote control and monitoring of household processes.

- **2010s**: the 2010s witnessed the emergence of the Internet of Things. This led to the development of smart devices capable of perceiving the surrounding environment and dynamically adapting to customer needs.

2.2.2 Applications

The implementations of the Internet of Things paradigm in the Smart Homes context are diverse and constantly evolving. Vendors are regularly introducing new devices and increasingly innovating systems. Below is an overview of the most prominent smart home applications [62] [63] [64]:

- **Energy**: a Smart Home, using smart appliances and devices, can operate in a way that minimizes energy waste. Smart lighting systems are among the most well-known applications of the Internet of Things technologies in collective imagery. Additionally, these systems can be designed to automatically adjust device behaviour based on whether someone is present in the home. For instance, in an empty house, forgotten electronic devices can switch off automatically or minimize energy consumption. Despite the initial installation cost, residents benefit financially from the energy savings over time.
- **Environment**: this application is closely related to the previous one. In fact, an intelligent energy system also significantly reduces the environmental impact. Furthermore, beyond water leakage prevention, air quality monitoring, and climate control, sensors can detect carbon monoxide, gas leaks, or fire hazards, potentially saving lives.
- **Security**: integrating home security and surveillance systems into the Smart Home network is becoming very common. This integration allows for timely alerts in case of intrusions. A Smart Home can also implement intelligent access systems based on passwords, voice, or smart card authentication. In other words, the Smart Home system serves as a vigilant observer, continuously monitoring all activities and events inside and outside the home [65].
- **Assisted Living**: new IoT technologies can drastically modify home environments to accommodate elderly or disabled individuals. There are applications and devices that simplify everyday activities, such as turning on lights and opening doors, through gestures like head movements or eye blinking. Additionally, these devices can quickly and easily alert someone in the event of a fall or other sudden emergencies.
- **Entertainment**: although this application may seem less important compared to others, IoT implementations in the entertainment field can significantly improve a person's mood after a stressful day.

2.2.3 Residents Adoption

Despite the numerous listed advantages of Smart Home systems, many people remain hesitant to install them in their homes. This study [66] assumes that a smart home system can save between 20% and 30% on household expenses. However, it found that a significant number of individuals doubt the utility, functionality, and tangible benefits that such a system promises to deliver. Furthermore, respondents expressed concerns about both the security and reliability of Smart Home systems. They worry that these systems might introduce vulnerabilities or fail to perform consistently. Cost is another major factor. Many potential customers question whether the advantages of installing a Smart Home system are worth the expense involved.

[67] [68] Other obstacles include the difficulty some people have in using these technologies and the lack of interoperability between devices from different vendors. Each vendor proposes its own implementation, complicating the process of integrating various devices into a cohesive system. This lack of standardization makes it challenging to create a single, interoperable Smart Home network. As a result, residents often struggle with compatibility issues and may find it difficult to make the most of their Smart Home devices.

According to another survey [69], people are concerned about how their personal data, including audio and video recordings of private conversations and moments, is handled. They worry about the possibility of this data being shared with third-party companies. However, it is important to acknowledge that sensitivity to privacy varies among individuals [70]. What may not violate one person's privacy could be a concern for another. Therefore, inhabitants should have the opportunity to configure and manage in detail how their personal data is processed. This ensures their complete control over privacy preferences and settings.

All the previous concerns highlight the need for better education in addition to reassurance about the practical benefits and security measures of Smart Home technologies. Moreover, to ensure that those assurances are based on concrete improvements, these systems require advancements in their security and privacy.

2.3 IoT Challenges

The concerns expressed by users regarding the adoption of smart home systems highlight other unresolved challenges in the context of the Internet of Things (IoT). The following list summarizes the most significant ones [71] [72] [73] [74] [75] [76]:

- **Scalability**: the rapid growth and proliferation of IoT technologies present a complex problem. As the number of smart devices and IoT networks continues to increase, managing these systems becomes progressively more challenging. Advanced technologies will lead to faster devices capable of collecting and exchanging ever-larger amounts of data. Consequently, managing Big Data, extracting useful information, and discarding noise become critical tasks. Developing smarter, faster, and more efficient algorithms for data processing is essential. Additionally, there is the challenge of storing this vast amount of data and ensuring that networks can manage the data exchange load. As each device needs a unique identity within the global IoT ecosystem, transitioning to IPv6 becomes necessary to accommodate this expanding network.

- **Interoperability**: another significant challenge is the need for interoperability between different devices and systems. Various devices using different communication protocols must coexist on the same network. Devices within the same subsystem may have different performance levels and processing speeds. This makes developing efficient interaction and cooperation essential. Despite progress, significant work remains in standardizing IoT architectures and data exchange formats and protocols. The lack of a unique adopted architecture hinders interoperability between IoT systems and creates barriers to seamless integration [77].

- **Reliability**: as IoT networks grow, the risk of system failures increases proportionally with the number of connected devices. Therefore, IoT networks must be designed to manage these failures effectively and continue functioning even under adverse conditions. It is essential to implement robust protocols that detect device problems and minimize the impact on other network devices. Additionally, providing adequate technical support to many customers poses a significant challenge. Efforts should focus on creating robust and reliable devices to reduce the frequency and severity of technical issues, thereby minimizing the need for support.

- **Energy Management**: efficient energy use is a priority as the number of IoT devices grows. We need to find more effective ways to power these devices, reducing their dependence on batteries and minimizing their environmental impact. Exploring sustainable energy alternatives, such as solar-powered smart devices, can significantly enhance energy management.

- **Social Implications**: the increasing development and adoption of IoT devices in various sectors will likely lead to significant social implications. The use of IoT technologies in enterprises and industries simplifies and automates many tasks, potentially reducing or eliminating the need for human labour in certain areas. This may result in job displacement and increased unemployment, causing social unrest. Furthermore, the pervasive integration of these technologies into daily life could foster a dependency on them, leading to unpredictable social consequences. Addressing these social implications requires careful consideration and proactive measures to ensure that the benefits of IoT technologies are equitably distributed and do not deepen existing inequalities.

These challenges illustrate the complexity and scope of issues that must be addressed to fully realize the potential of IoT technologies in Smart Homes and other applications. Overcoming these challenges will require coordinated efforts from researchers, developers, policymakers, and industry stakeholders to create robust, secure, and user-friendly IoT systems that can enhance the quality of life while mitigating potential risks.

2.4 IoT and Smart Home Security Overview

Another significant challenge is enhancing the security of Internet of Things systems. Equally important is ensuring the privacy of data collected by smart devices.

IoT systems are inherently vulnerable to cyberattacks due to their connection to the Internet [78]. This vulnerability is especially concerning Smart Homes, where security systems and appliances such as ovens and stoves are present. Security breaches in these areas can pose potential risks of physical harm, highlighting the critical need for implementing strong security measures.

In their haste to keep up with market growth, many vendors release IoT devices with security vulnerabilities that attackers can exploit. According to the 2023 SonicWall Cyber Threat Report [79], there was a 243% increase in IoT malware attacks between early 2018 and late 2022. Within the shorter span from early 2021 to late 2022, the volume of such attacks grew by 87%.

2.4.1 Security Threats

Threats to the security of Internet of Things systems are numerous and varied. Many of these threats are malware, each with specific characteristics that can damage and compromise an IoT network. Malware is malicious software designed to alter the normal functioning of targeted software [80]. It can cause harmful actions within the target system, compromising both the integrity and confidentiality of the data it manages [81].

Here is an overview of some possible IoT security threats [82] [83] [84] [85]:

- **Denial of Service (DoS)**: This attack makes the IoT system's service inaccessible by occupying necessary resources, preventing the system from responding to users' requests. Mitigating and preventing DoS attacks within an IoT network is highly challenging. Researchers dedicate significant efforts to develop increasingly effective methods to tackle this issue [86].

- **Botnet**: malware can take control of certain IoT devices and use them for illegal activities. Multiple compromised devices form a botnet. Attackers use these compromised devices to mask their real identity during attacks [87]. Botnets are commonly used to execute Distributed Denial of Service (DDoS) attacks, which are large-scale DoS attacks using multiple resources.

- **Spyware**: this malware monitors activities within an infected system, collecting users' data and personal information [88]. In IoT systems with multimedia devices like cameras, spyware can access users' audio and video recordings, severely violating their privacy [89].

- **Backdoors**: a backdoor is a piece of code inserted by developers or attackers to bypass security or authentication measures [90]. While developers may insert backdoors to facilitate certain actions, attackers can exploit them to take control of a smart device and gain network access, potentially attacking other network parts [91].

- **Sleep Deprivation**: this attack targets network endpoints to drain their power, reducing battery life and shortening device lifespan. Typically, smart devices enter sleep mode to minimize power consumption when not in use. Sleep deprivation attacks prevent devices from entering this state, increasing power consumption to the point of making the devices unavailable [92] [93].

A significant challenge in IoT security stems from the inherent limitations of IoT devices [94]. These devices cannot manage complex cryptographic operations due to limited computational power and storage capacity. Additionally, such operations can increase power consumption, which is often prohibitive since many IoT devices operate in low-power contexts without the aid of powerful batteries. Therefore, it is complex to implement the same robust security solutions adopted in other contexts.

The following list presents some guidelines and principles to take into consideration when defining security solutions for the Internet of Things systems [95] [96]:

- **User Awareness**: educate users about their pivotal role in ensuring IoT security. Developing sophisticated security solutions is futile if users employ weak passwords or fail to use them at all. It is well documented that users often leave default passwords or choose easily guessable ones [97].

- **Authentication**: strengthen authentication mechanisms for IoT devices and users. Only authorized devices and users should interact within the IoT network and access its resources. It is also crucial to ensure strong authentication between IoT network endpoints and supporting Cloud platforms [98], as sensitive user information is exchanged in these communications.

- **Confidentiality**: protect user data privacy through encryption and secure keys. Despite various proposed solutions, there is currently no standardized and properly structured solution to guarantee confidentiality in an IoT system [99].
- **Integrity**: maintain data integrity within the IoT network [100]. Unauthorized devices and users must not be able to alter system data and resources.
- **Availability**: ensure that an IoT system remains functional when users need to access it [101]. Implement measures to guarantee continuous operation even during failures or attacks, enabling quick recovery from disruptions.
- **Mutual Trust**: implement a framework that fosters mutual trust among users, particularly for devices with multiple owners. Secure management of shared devices and facilitation of ownership transfers within corporate environments are essential to maintain operational efficiency and security.

2.4.2 Security Solutions

Various solutions have been proposed, ranging from well-established concepts to innovative methodologies such as: *Machine Learning*, *Blockchain*, and *Firmware Analysis* [102]. We will briefly overview the first two approaches before focusing on the third one:

- **Machine Learning**: a Machine Learning (ML) system can learn and make decisions within a specific domain after being trained on extensive data related to that domain [103]. These models are effectively used in the context of Internet of Things security. In IoT security, ML models analyze network traffic to identify patterns indicative of cyberattacks and those of normal traffic. This enables the detection of abnormal intrusion attempts or potential DoS [104].
- **Blockchain**: initially developed for cryptocurrency, blockchain technology has rapidly found applications in IoT security. [105] A blockchain comprises a series of transactions, known as records, which correspond to the blocks of a chain. Each record is linked to the previous one, and a set of records is called a ledger. All participants who take part in the transactions must have a copy of the ledger. A miner, an autonomous entity, oversees the blockchain and ensures the validity of the transactions added to the ledger. This structure makes it extremely difficult for an attacker to manipulate or insert false records without being detected. In a Smart Home system, a miner can monitor communications, effectively managing security, authentication, and access control. Thus, blockchain technology can be used within IoT networks to guarantee confidentiality, integrity, and availability [106].

2.5 Firmware Analysis

Smart devices serve as the endpoints of an Internet of Things (IoT) system. They are responsible for collecting data and carrying out user-requested operations. The firmware is a binary provided by the device vendor and is stored in the non-volatile memory of a device. It regulates the behaviour of the device by interacting with actuators and sensors [107] and serves as a connection between software and hardware components. Vendors must ensure firmware security and issue patches as needed to protect user data privacy and physical safety in an IoT system [108].

There are various methods for analyzing firmware for vulnerabilities. Two of the most popular solutions are static and dynamic analysis [109]. Static analysis examines a firmware source code in search of vulnerabilities, but it is not applicable if the code is not available. On the other hand, dynamic analysis involves executing firmware.

While effective against known vulnerabilities, these methods struggle with zero-day vulnerabilities, which are vulnerabilities that hackers can exploit because they are unknown to developers at the time of their discovery [110]. The most effective approach is a hybrid approach that integrates static and dynamic analyses, possibly enhanced by machine learning. This method enhances vulnerability detection, covering both known and zero-day vulnerabilities [111].

2.5.1 Static Analysis

Static analysis involves examining a source code without executing the relative binary or needing any input data [112]. It starts from the source code and checks for issues by comparing it against known error patterns [113] [114].

Some issues that can be detected through static analysis include the following ones [115] [116] [117] [118]:

- **Incorrect Memory Management**: this includes attempts to access deallocated memory, inefficient allocations, and memory leaks. Allocated memory is a portion of memory reserved for a specific purpose. A memory leak occurs when that portion of memory is not deallocated after its use is no longer needed.
- **Incorrect Resource Management**: this involves mishandling the opening of resources such as files and sockets, or failing to close them properly.
- **Poor code architecture**: this refers to structuring code inefficiently without using all the characteristics of the language. Poor architecture makes it more difficult to read, understand, and debug, complicating future revisions or

error corrections. Lack of modularity and unclear code logic also reduce maintainability.

- **Bad Practices**: this involves coding practices that reduce code clarity and efficiency, such as overly nested conditions, functions with too many parameters, hard-coded values, circular dependencies, and poor class coupling.
- **Dead Code**: this refers to code that will never be executed because the condition that determines its execution will never be set to true.
- **Duplicated Code**: this refers to code segments repeated in various parts of the code. This duplication significantly affects maintainability and can be resolved by extracting the repeated code into a separate function.
- **Failure to comply with guidelines**: this includes organization-level guidelines for code writing, such as indentation or variable naming conventions.
- **Use of insecure constructs**: this involves constructs known to be vulnerable to security issues. It also includes outdated libraries with known vulnerabilities.
- **Concurrency errors**: this includes mishandling synchronization among threads accessing the same resources, detecting possible deadlocks, race conditions and starvation situations [119].
- **Logical errors**: these are mistakes in conditional constructs and loops. They could, for example, lead to an infinite loop [120].
- **Improper handling of errors and exceptions**: this refers to the failure in managing errors or exceptions, such as in the Java language [121].
- **Security vulnerabilities**: this identifies code portions which potentially expose a program to attacks like SQL Injection [122] or buffer overflow [123].

One of the earliest static analysis tools is Lint [124]. Developed in the 1970s at Bell Laboratories by Stephen Curtis Johnson, Lint was designed for the C language. It provides additional checks on source code correctness without impacting compiler performance. Integrating static analysis into the compilation process might seem advantageous, but it could slow down and undermine its efficiency. Therefore, Lint performs its checks independently from the compiler.

Lint's checks are stricter than those of the compiler. It imposes tighter constraints on typing and identifies code segments that might reduce portability across different operating systems and architectures [125].

There are various methods to check source code in search of issues [126]. Code review is among the most straightforward. Peer code review involves team members analyzing each other's code, but this process can be complex and resource-intensive in large codebases. Static analysis tools partially automate some aspects of this process, making the two methods complementary. Conducting static analysis before code review helps identify common errors. This reduces the effort required for a review, allowing reviewers to focus on more complex issues, thus improving the efficiency of the entire review.

Static analysis is vital for code maintainability because it allows the identification of issues early in development, avoiding high effort for complex refactors later.

The main techniques adopted for static analysis are [127]:

- **Lexical analysis**: this technique divides a source code into fragments and compares them with reference patterns to detect potential vulnerabilities. However, it has limitations, and it often results in a high rate of false positives.
- **Type inference**: it uses the compiler's type inference system to perform checks, applying additional rules to ensure the correctness of variables and functions.
- **Data flow analysis**: by examining the path of variables within a program through the use of a control flow graph, this method identifies situations where the assigned values could lead to vulnerabilities.
- **Rule checking**: this technique verifies program adherence to predefined security rules.
- **Constraint analysis**: it creates and verifies constraints regarding the relationships between variables.
- **Patch comparison**: this method compares patches against the current program state to identify security vulnerabilities.
- **Symbolic execution**: it replaces input values with symbolic ones and analyzes the resulting algebraic expressions to check whether they meet determined constraints. For example, an execution with symbolic values could reveal that passing a negative number to a specific function causes the program to crash.
- **Abstract interpretation**: this technique provides a formal description of a program, aiming to derive its semantic meaning.
- **Theorem proving**: it transforms a program into logical constructs and uses them in a mathematical proof-like demonstration to establish the program's correctness.
- **Model checking**: this technique constructs a formal model of a program, such as a state machine or a graph, that could be used to explore all the states, thereby allowing the detection of unexpected behaviours of a program.

To determine software quality, static analysis can use several metrics calculated from the source code under analysis [128]. These can be combined into more complex code quality assessment models. Setting thresholds for these metrics is complex and requires well-defined steps and processes. However, the same thresholds may not be suitable for different scenarios.

Among those metrics, we are presenting the ones related to source code size and complexity.

Lines of Code

Lines of Code (LOC) metrics measure the size of the source code [129] [130]:

- **Source Lines of Code** (**SLOC**): the total number of source code lines.
- **Comment Lines of Code** (**CLOC**): the number of lines containing comments.
- **Physical Lines of Code** (**PLOC**): the number of lines that are not blank or commented out. A line with both code and comments is counted in both PLOC and CLOC metrics.
- **Logical Lines of Code** (**LLOC**): the number of statements in a source code. The definition of a statement varies from language to language, and it generally refers to a specific action performed by a programming language [131]. A statement may span several code lines, or there may be multiple statements in a single line. Each statement, regardless of its length or position, counts as one for the final sum.
- **Blank Lines of Code** (**BLOC**): the number of blank lines in a code.

Complexity

The following metrics are designed to calculate the complexity of the source code:

- **Cyclomatic** [132] [133] [134]: this metric calculates the complexity of a program by considering all possible independent execution paths. Cyclomatic complexity is mainly used to establish software's testability, i.e., the ease with which a program can be tested. This metric starts from a control flow graph where each node represents either an uninterrupted sequence of operations, or a jump due to conditional constructs, such as if-then, if-then-else and switch statements, or iterative constructs, such as while and for. Each edge that Each edge that comes out of a jump node represents one of the possible paths of the program. Cyclomatic complexity is computed using a formula based on the number of nodes (N) and edges (E) in the graph, originally defined by Thomas J. McCabe in 1976 as

$$CC = E - N + 2 \tag{2.1}$$

- **Cognitive** [135] [136]: this metric measures the clarity of a program and the effort required to intuitively understand its functionality. It primarily assesses the maintainability of a source code rather than its testability, overcoming the limitations of cyclomatic complexity in evaluating code maintainability. The calculation method varies depending on the specific static analysis tool used. Generally, a weight is assigned to conditional and iterative constructs, and the complexity value is then calculated based on how these constructs are combined. If they are not deeply nested, the complexity value will be low, indicating an easy-to-understand program. Instead, if there are heavily nested consturcts, the complexity increases significantly. Logical conditions involving logical operators such as AND and OR are also considered. Long conditions with many operators contribute significantly to the final complexity value.

Metrics Example

```c
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 
 4 // C function
 5 // operation = 1 -> a + b
 6 // operation = 2 -> a - b or b - a
 7 int function(int a, int b, int operation) {
 8     int result;
 9 
10     if (operation == 1) {
11         result = a + b;
12     } else if (operation == 2) {
13         if (a >= b) {
14             result = a - b;
15         } else {
16             result = b - a;
17         }
18     } else {
19         printf("Not recognized 'operation'");
20     }
21 
22     return result;
23 }
```

Listing 2.1: C code snippet

For completeness, we provide an example of these metrics calculated on the code snippet written in C language as shown in Listing 2.1. To obtain the metric values, we use a tool called rust-code-analysis [2], which will be explained in more detail later. This tool allows the calculation of LOC, cyclomatic complexity and cognitive complexity.

The metric values for the example above are:

- **SLOC**: the value is equal to 23. Note that this value corresponds to the total number of source code lines since it is equal to the number shown in the left part of the figure.
- **CLOC**: the value is equal to 3 and corresponds to lines 4, 5 and 6.
- **PLOC**; the value is equal to 17 and corresponds to the lines 1, 2, 7, 8, 10-20, 22 and 23.

[2]rust-code-analysis: https://github.com/mozilla/rust-code-analysis

- **LLOC**: the value is equal to 9 and corresponds to lines 8, 10-14, 16, 19 and 22.
- **BLOC**: the value is equal to 3 and corresponds to lines 3, 9 and 21.
- **Cyclomatic Complexity**: the value is equal to 4.
- **Cognitive Complexity**: the value is equal to 6.

2.5.2 Dynamic Analysis

Different from static analysis, dynamic analysis involves examining software during its execution. This approach provides a closer view of the program's actual behaviour and helps to clarify uncertainties related to issues that static analysis may only partially address [137]. Advancements in software development have introduced paradigms and constructs that manifest only at runtime, posing challenges for static analysis due to its inherent imprecision in such scenarios. Concepts like dynamic binding, dependency injection, polymorphism, and concurrency are more suited for dynamic analysis.

A comparison study [138] points out that both types of analysis examine only a subset of possible program executions. Therefore, they are considered complementary, with static analysis results often serving as input for dynamic analysis. A combined approach of the two avoids redundant analyses and can concentrate the effort on unexplored execution paths. The following list details some dynamic analysis processes and the relating techniques [137]:

- **Instrumentation**: the dynamic analysis process typically involves an initial phase called profiling or tracing. During profiling, program execution is observed to capture events such as function calls or conditional branch points. This information is captured through instrumentation, which involves modifying a program's source code, object code, or binary by inserting specific instructions to extract information from the program's execution. The information gathered during the profiling phase will be used as input for the subsequent phases.
- **VM Profiling**: this method runs the program on specific virtual machines on which profiling is done through interfaces and plugins. It simplifies the profiling process by leveraging the virtual machine's interface, rather than directly managing trace extraction.
- **Aspect-Oriented Programming** (**AOP**): It is a programming paradigm that enables the implementation of cross-cutting behaviors, which are functionalities that affect multiple parts of a program without having to modify

each part individually. In dynamic analysis, AOP can be used to perform profiling.

2.5.3 Testing and Coverage

[139] Testing is another fundamental activity in ensuring software quality. It is used to verify that software operates as intended. Validation and Verification are two key aspects for testing. Validation aims to ensure that software meets user requirements by testing against specific inputs in order to produce an expected. Verification, on the other hand, checks for design errors and bugs in software with the aim of assessing whether it has been developed correctly.

Coverage is a metric used to evaluate the quality of testing by indicating how thoroughly tests cover a codebase. Different coverage metrics include the percentage of covered lines, statements, functions, or paths [140]. To obtain coverage values, the code under analysis must be instrumented [141], which means that the code will be extended with additional instructions to gather coverage information during the execution of the tests.

This study [142] suggests that higher coverage typically leads to uncovering more bugs, making it desirable to aim for a high coverage value. However, it is important not to rely solely on this metric to assess testing quality, as even 100% coverage does not guarantee the identification of all bugs. Therefore, evaluating test quality based exclusively on coverage can lead to misleading conclusions.

Regarding coverage thresholds, each development team typically establishes its own conventions. However, commercial tools often employ a traffic light model. For example, Codecov [143] uses a model in which red identifies coverage below 60%, yellow means coverage between 61% and 79%, and green denotes coverage above 80%.

2.5.4 Rust

Issues detected by static and dynamic analysis vary significantly depending on the programming language features. Certain issues are common in every language all languages, while others are only specific to certain languages. Some programming languages are developed to address common issues and mitigate or eliminate them, and Rust [144] is one such language. It stands out as a relatively new language compared to established ones such as C, C++, or Java, and originated as a personal project by Mozilla employee Graydon Hoare in 2006 and gained prominence starting from 2010.

Rust was deliberately designed to proactively eliminate issues present in other languages, and has established itself as one of the most secure and at the same time efficient programming language [145]. The following lists outlines some of its key features: [146] [147]:

- **Memory Management**: Rust implements a memory management mechanism based on the ownership concept. This mechanism manages memory through a set of rules that the compiler checks at compile time to enforce that each portion of memory has a single owner at any given time. When a variable is declared, Rust allocates memory to it and automatically deallocates this memory when the variable goes out of scope. Additionally, Rust supports shared references for read access and exclusive references for write access, ensuring safe management of these references to prevent common memory-issues, such as the reference of deallocated memory areas, attempts to deallocate an already deallocated memory zone or null pointer exceptions.

- **Bounds Checks**: Rust detects any attempts to access out-of-bounds array elements at runtime. If such an attempt occurs, execution is immediately stopped, and an error message is returned reporting the unauthorized access attempt.

- **Zero-cost Abstraction**: Rust does not require a garbage collector, in contrast to other languages such as Java. This results in a higher performance since it completely removes the garbage collector overhead. In fact, Rust provides high-level constructs at zero cost. These constructs are more compact and easier to use compared to the ones provided by low-level languages. Despite their simplicity, they maintain the same or even higher performance.

- **Safe Concurrency**: Rust allows the implementation of concurrent programs in a simple and safe manner. Through the use of special constructs and relying on the concepts of borrowing and ownership mentioned earlier, it makes it easy to create thread-safe programs free of issues such as race conditions, starvation, and deadlocks.

- **Crate**: represents the smallest piece of compiled Rust code that the compiler processes, and can be used to either define a binary or a library. In the first case it represents an executable program that can be directly runned through the command-line. A library, instead cannot be directly executed and is used to define reusable functionalities intended to be shared among multiple projects. The crate concept promotes modularity by offering developers the possibility to encapsulate in separated and reusable units. Moreover, Rust comes with a package manager called Cargo that allows to facilitate the integration of external libraries in a crate, ensuring that the versions of the dependencies are managed correctly.

Rust is ideal for contexts such as software and firmware development, as it is well-suited for resource-constrained environments [148]. Furthermore, it can be easily used in conjunction with other low-level programming languages such as C, making it very versatile.

This study [149] demonstrated that Rust delivers performance comparable, or even superior, to C/C++ when used for programming ESP32 microcontrollers. This outlines Rust's potential in IoT while providing the benefits of a higher-level programming language.

Rust has been chosen for developing the tools discussed in this thesis. All the features we have listed and the advantages in terms of performance, memory safety, security and minimal runtime overhead make this language an ideal choice for developing reliable and efficient software.

2.6 Certification

As already emphasized, the lack of well-defined standards and regulations poses substantial challenges for the Internet of Things ecosystem. According to [150], there is an urgent need for regulatory agencies capable of quickly adapting to the rapid growth of the IoT market. These agencies must establish dynamic regulations and certification methodologies to ensure that high-security standards are maintained as the number of smart devices continues to grow.

In addition, certification methodologies must account for the diversity among IoT systems and the fact that they often consist of a wide variety of smart devices using different technologies [151].

This study [152] points out the inadequacy of current cybersecurity certification systems when applied to the Internet of Things context. It highlights that the static nature of existing certification processes struggles to address the dynamic and evolving nature of IoT systems. Furthermore, it identifies a lack of meaningful metrics to evaluate the real effectiveness of certification processes in enhancing the security of IoT systems.

The OWASP IoT Top 10 [153] lists the lack of proper firmware validation as one of the major issues threatening the security of IoT devices. The absence of well-defined processes for certifying firmware represents a significant gap for the Internet of Things security landscape and must be addressed as a primary issue.

The firmware analysis approaches and techniques presented in the previous sections could serve as a starting point for defining certification methodologies which validate and certify smart devices' firmware.

This thesis aims to partially address these challenges by proposing a solution tailored for Smart Home firmware that could be integrated into a more vast certification process.

We propose a manifest that details all potential risks that may arise from device actions in a Smart Home system. This is similar to an Android application's manifest [154], which outlines permissions required for installation and execution of the applications, such as camera, location, or microphone usage.

The purpose of our manifest in the certification process is to ensure transparency regarding the risks associated with the smart devices' actions within the Smart Home. Additionally, we define some mandatory risks for specific device actions and highlight in the manifest if a developer has failed to declare them.

Chapter 3

Firmware Analysis Tools

Software developers tend to prioritize implementing new features over performing activities such as testing, writing documentation, setting up a continuous integration workflow, or reviewing the code to find which parts need to be refactored [155] [156]. These activities are crucial for maintaining and enhancing software quality. Therefore, it would be better to design tools that streamline and automate them. Such tools would also help reduce potential human errors caused by inadequate attention and prioritization of these tasks.

Implementing proper measures and tools to ensure high code quality is particularly important in the development of IoT device firmware. Since IoT devices interact directly with the physical environment and people, ensuring high code quality is crucial. In fact, poor coding practices can alter device behavior, posing risks to people's safety. Therefore, implementing proper measures and tools is essential to mitigate these potential risks.

In this chapter, we present `weighted-code-coverage` and `complex-code-spotter`, two static analysis tools that have been created in the context of Internet of Things development. These tools have been designed to analyse the source code of IoT applications and firmware, but can also be used to perform a static analysis of software pertaining other domains. Our work focused on refactoring these tools, streamlining their structure and implementation, introducing new features, and trying to enhance their overall performance.

3.1 weighted-code-coverage

`weighted-code-coverage`[1] is a tool written in Rust that proposes a solution to merge the concepts of code coverage and code complexity into single metrics. In particular, this tool implements several metrics, each aiming to combine coverage and complexity through specific algorithms and mathematical expressions.

Typically, these two concepts are separately analyzed using different tools to assess code test coverage and structural complexity. Instead, `weighted-code-coverage` implements an analysis where both these aspects have been taken into account, ensuring that neither high coverage nor low complexity alone will be enough to obtain a positive evaluation.

For example, code with maximum coverage but very high complexity will not be positively evaluated. Similarly, a well-structured and modular code formed by simple functions, but with a small coverage value, will also receive a negative evaluation. The best-case scenario is code that is both extensively tested and organized to avoid overly complex functions and scopes.

By combining these two aspects, `weighted-code-coverage` promotes a more comprehensive approach to software quality, encouraging developers to strive for both thorough testing and maintainable code structures.

The main goal of our work consisted of simplifying and refactoring `weighted-code-coverage`. However, we have also introduced new features related to output and metrics computation and improved the functionalities already present.

The initial version of the tool had an unorganized code and several design problems. It was characterized by extensive use of repeated code in various places throughout the program. This redundancy greatly increased the developer's maintenance effort, as any changes had to be applied to all instances of the repeated code. Failing to update all repetitions could lead to inconsistencies and unpredictable behaviour.

Moreover, the presence of repeated code and lack of modularity increased the source code length, making it more difficult to navigate and understand at first glance. Therefore, it was necessary to review the entire software design and perform a comprehensive refactor. This refactoring aimed to increase modularity and reorganize the repeated code into reusable functions or constructs, improving both maintainability and clarity.

[1]https://github.com/SoftengPoliTo/weighted-code-coverage

Another problem was that the tool's API was overly complex and featured a large number of functions. This complexity made the API unintuitive and challenging to use. Additionally, the API's intricate nature made it difficult to thoroughly test the general behaviour. It then seemed absolutely necessary to simplify the API and reduce the number of exposed functions to enhance usability, improve developer experience, and ensure more straightforward and effective testing.

3.1.1 Tool Structure and Workflow

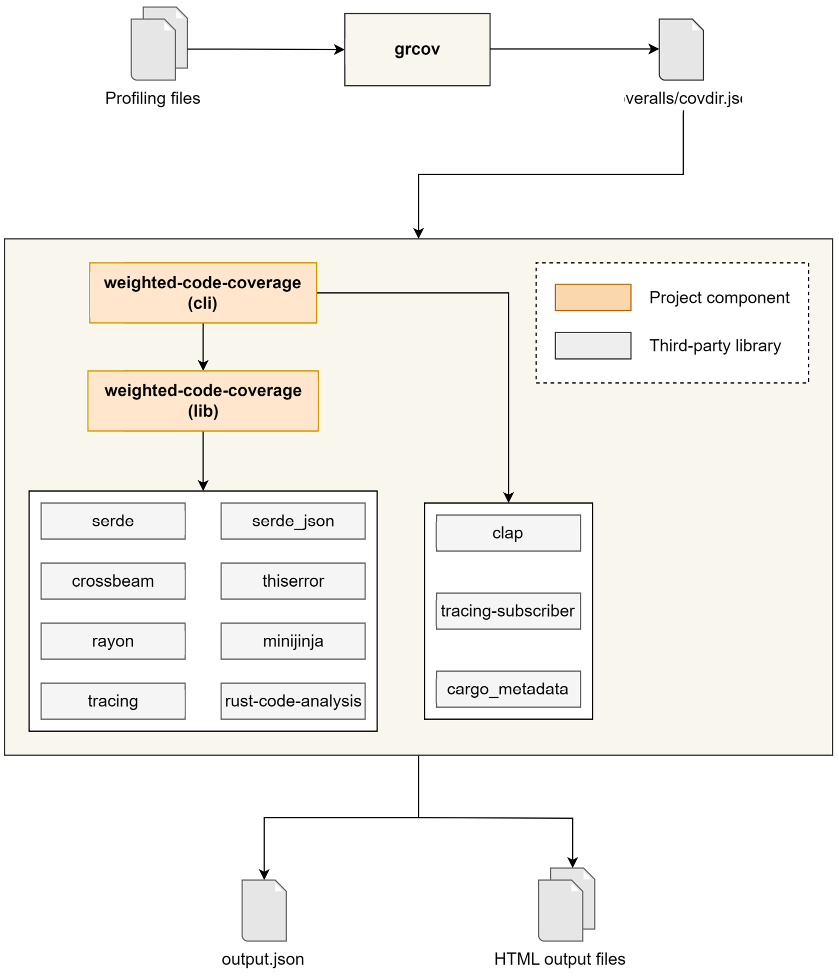


Figure 3.1: weighted-code-coverage structure and workflow

We start by providing a brief overview of the tool's structure, followed by an explanation of its workflow. This overview will clarify how the tool has been designed and how it operates. Understanding these aspects is fundamental to comprehending the rationale behind every modification we have made in the code and why they have been so necessary.

As shown in Figure 3.1, `weighted-code-coverage` provides a library crate and a command line interface (CLI). The crate can be used to obtain a Rust data structure containing a project's metrics values. The CLI, leverages on the API provided by the crate, to compute the metrics values for a project whose path is passed as input, and generate outputs in different file formats containing the results of analysis.

Both the CLI and the crate leverage on some third-party libraries. The most important ones are:

- **`serde`** and **`serde_json`**: are used within the tool to serialize Rust data structures into JSON files and to deserialize JSON files into the respective data structures.
- **`crossbeam`** and **`rayon`**: are used in conjunction with the Rust standard library to implement the concurrent execution mechanism of the tool. These crates provide specific and optimised constructs to implement various concurrent programming patterns in a simple and efficient manner.
- **`minijinja`**: is a lightweight crate that provides a template engine that allows the management and rendering of templates in Rust. Templates are files that contain placeholders and logic that can be replaced and evaluated with values dynamically assigned during program execution.
- **`rust-code-analysis`**: this crate enables the analysis of code written in various programming languages by constructing an Abstract Syntax Tree (AST)[2]. Each node of an AST corresponds to a specific programming language construct, keyword or token.
- **`clap`**: enables the creation of a CLI by providing constructs to easily manage command line arguments.

We will provide more details about these crates when we describe them during the description of the program workflow.

Examining Figure 3.1, we can see that the workflow starts with profiling files of the project under analysis. These files are obtained by instrumenting a program, which means that the compiler inserts additional instructions into the code to collect data during its execution [157] [158]. This profiling data may include execution time, memory allocation, and frequency calls of specific functions. Such information can eventually be used to analyze the execution of a software instance and its performance [159].

In our case, we use profiling to gather test coverage information for a project [160]. To obtain coverage profiles for a Rust project, we start by configuring the development environment. This involves installing tools that support profiling and setting compilation options that instruct the compiler to include coverage extraction instructions. These options can be set using environment variables.

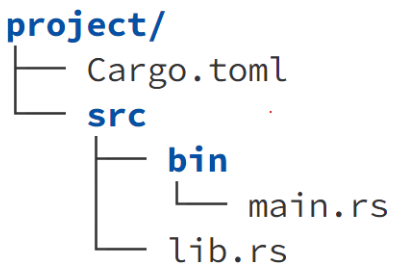
[2]An AST provides a hierarchical representation of the syntactic structure of a source code, depicting its organization and relationships among the elements

When compiling the Rust project with these options, the `rustc` compiler embeds extra code into the program. This extra code, or instrumentation, tracks which parts of the code are executed and records this information.

When the project's tests are run, the instrumented code is executed. Therefore, data retrieved from code pieces covered by tests is collected and recorded in the output profiling files. At the end of this process, we obtain various profiling files containing the coverage information of the entire project.

These files are then passed as input to a tool called `grcov`[3]. This tool analyses the profiling files and extracts from them information to produce a detailed report about the test coverage of a project. `grcov` allows producing reports in various formats, including `HTML`, `lcov`[4] and `JSON`.

weighted-code-coverage parses as input the `grcov JSON` format, which can be in two different variants: **coveralls** and **covdir**. Despite minor differences in how metadata have been represented, both variants contain all the coverage information necessary to compute the project metrics.



```
project/
├── Cargo.toml
└── src
    ├── bin
    │   └── main.rs
    └── lib.rs
```

Figure 3.2: Example project tree structure.

For example, if we consider a Rust project with the same structure as the one shown in Figure 3.2, first we execute `grcov`, passing as input the profiling files of a project and choose coveralls or covdir as output format.

```
"source_files": [
  {
    "coverage": [2, 2, 2, 2, 2, 0, 0, null, 0, null, 2],
        "name": "src/lib.rs"
  },
  ...
]
```

[3]https://github.com/mozilla/grcov

[4]https://github.com/linux-test-project/lcov

As we can see from the example above, the coveralls format contains a list of all project files, each accompanied by a `coverage` array whose number of elements corresponds to the total number of lines in a file. Each array value represents the coverage of a specific line, and can be of three different types:

- **0**: means that the line of code in a file, which represents the array index for obtaining its corresponding coverage value, is not covered by tests.
- **Positive integer**: represents the number of tests that cover that line.
- **null**: denotes a line that is ignored during the computation of coverage. For example, when a line contains a comment.

On the other hand, if we consider this snippet of code taken from the covdir output for the same project file:

```
"src": {
  "children": {
    "lib.rs": {
      "coverage": [2, 2, 2, 2, 2, 0, 0, -1, 0, -1, 2],
      "coveragePercent": 66.67,
      "linesCovered": 6,
      "name": "lib.rs"
    },
  ...
  }
}
```

we can see that this variant contains also other information in addition to the `coverage` array, such as file coverage percentage and the total number of covered lines. The main difference with respect to the coveralls format is that covdir uses the -1 value instead of `null` to denote an ignored line. Another major difference is that coveralls utilises a flat representation, listing each source file one by one as an array element, while covdir uses a nested, hierarchical structure.

`weighted-code-coverage` accepts as input both the covidr and coveralls variants produced by `grcov`. As we will see, part of our work focused on the parsing of these two variants. This effort led to create data structures that could uniformly represent the information of both variants despite their differences.

Once the coverage information has been obtained, the tool requires complexity metrics and certain lines of code (LOC) metrics in order to compute its own metrics. For this purpose, it uses `rust-code-analysis` to compute cyclomatic complexity, cognitive complexity and the LOC metrics presented in Section 2.5.1.

We will further explore the details of these metrics and their composition later on. For now, it is enough to know that once these metrics are computed, they are stored within specific data structures.

Furthermore, the tool can also generate an output, in two different formats, to present the information contained within these data structures. The `JSON` format is the first available output type, which consists of a mere deserialisation of the data structure using `serde_json`. The second output type instead is the `HTML` format, which provides a more user-friendly and easier-to-understand representation. Later in this chapter, we will dedicate an entire section to the output formats of this tool.

Now, we will have a more in-depth look at the main parts of `weighted-code-coverage`, listing every modification we have made and highlighting the rationales behind our choices.

3.1.2 Parsing of Grcov files

The first step consisted of reimplementing the deserializatio of the coverage files produced by `grcov`. In fact, by analysing `weighted-code-coverage` code, we have noticed that the parsing of these files was partially implemented within the same functions that were instead supposed to only use the coverage information the files contain. Moreover, the two formats were managed separately, leading to a repetition of code for parsing logic common among modules. These issues resulted in poor code modularity and significantly decreased maintainability.

To address these problems, we created a separate module called `grcov` exclusively dedicated to the parsing of the two `JSON` variants. This module provides data structures that uniformly represent the information contained within the coverage files. Consequently, the rest of the code can use these structures without having to differentiate between formats. Furthermore, this approach allowed us to separate the parsing logic specific for each format into distinct files and consolidate the logic which was common among them, thereby eliminating a significant amount of repeated code and enhancing reusability.

We also realised that the previous implementation did not completely deserialize the cover format. As we have seen before, this format represents project files hierarchically rather than linearly. The original implementation failed to correctly parse covdir files for projects with multiple directories and subdirectories, stopping at the first level of the hierarchy. To address this problem, we created a function that deeply parses the entire structure represented in the covdir format, retrieving coverage information for every file contained in the project. This ensures that projects with complex directory structures are fully and accurately analysed.

One modification that contributed to standardise the representation of the two `grcov` variants within `weighted-code-coverage`, was ensuring that coverage arrays for project lines were uniformly represented. Specifically, we ensured that `null` and -1 values, used respectively in coveralls and covdir to denote an ignored line during coverage computation, were represented consistently using the same Rust construct. This change, though seemingly trivial, was crucial as it gave us the possibility to use a single, interoperable format to represent the coverage information. Previously, this was not the case, leading to a worse tool maintainability.

In fact, the previous `grcov` file parsing implementation was one of the main reasons behind the code repetition problem within `weighted-code-coverage`. The other reason that led to an increase in duplicate code stems from the fact that the tool can operate in two modes:

- **files**: the tool computes the metrics only for the files contained in a project.
- **functions**: the tool also computes the metrics for each single function of a project.

The absence of data structures capable of uniformly and independently representing coverage information, coupled with the distinction between these two operational modes, resulted in dividing the tool's workflow in four cases:

1. **files-coveralls**: the tool operates in files mode using a coveralls file.
2. **files-covdir**: the tool operates in files mode using a covdir file.
3. **functions-coveralls**: the tool operates in functions mode using a coveralls file.
4. **functions-covdir**: the tool operates in functions mode using a covdir file.

These four cases, however, share a significant part of operational logic, which means that there are code segments where they execute the exact same instructions. Despite that, the old implementation opted for duplicating code instead of using an approach based on functions that can efficiently encapsulate and reuse the shared logic. This approach increased the overall code complexity, making it challenging to maintain and introducing the risk of errors when updating functionalities which are shared between the four different operational scenarios of the tool.

Many of the modifications applied to this tool have been made to address problems resulting from structuring its workflow according to this separation. The creation of the `grcov` module, as well as the changes we will eventually present, allowed us to eliminate this separation and compact the tool's workflow, leading to a more streamlined and maintainable code.

3.1.3 Reimplementation of the Concurrent Execution Mechanism

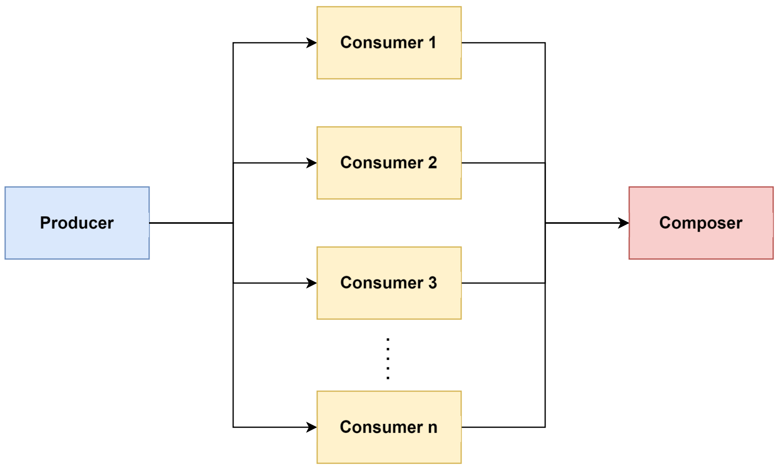


Figure 3.3: producer-consumers-composer pattern

After the implementation of the *grcov* module, we focused on refactoring and simplifying the concurrent execution mechanism. This can be considered a key aspect of the tool, because it led to the analysis of multiple files in parallel, significantly accelerating the computation of a project's metrics.

The concurrent implementation of this tool is based on the *producer-consumers-composer* pattern shown in Figure 3.3. This pattern consists of splitting and assigning independent execution units to multiple threads[5] according to the following structure:

- **Producer**: this thread is responsible for partitioning the workload into various jobs, which are tasks that can be performed independently and concurrently by different threads. The producer is also responsible for sending the jobs to consumer threads by placing them into a shared data structure. In the context of `weighted-code-coverage`, this shared data structure is represented by a channel, a construct provided by the `crossbeam` library. A channel is a communication mechanism that enables threads to communicate asynchronously. This means that a thread can send a message through the channel without having to wait for the receiver thread to receive it. Consequently, the receiver can manage the message whenever it is ready. This mechanism allows

[5]are the smallest unit of execution within a process. Threads allow the parallel execution of several operations by sharing the same process resources.

threads to work without blocking each other, thereby improving efficiency for multi-threaded programs.

- **Consumers**: these threads asynchronously receive jobs from the producer through the channel. Upon receiving a job, each consumer executes a specific algorithm which resolves its assigned task, producing a result in output. Subsequently, the consumer sends this result to the composer thread using another shared channel.
- **Composer**: this is the thread at the end of the entire structure. Also referred to as the *sink*, the composer gathers the data generated by consumers and, in case, processes it, producing the final result.

In `weighted-code-coverage`, the `producer-consumers-composer` pattern has been employed to create a job for each file of the project under analysis. This allows consumer threads to calculate metrics for every file in parallel. In addition to computing the metrics of a file, consumers also create data structures filled with information about the coverage of a file, its complexity, and the respective LOC values. These data structures are then passed to the composer, who exploits them with the objective of computing the `weighted-code-coverage` metric values. We will provide a more detailed description of these output metrics in a following section 3.1.7. Furthermore, the composer also computes minimum, maximum and average values of the files' metrics.

The tool's implementation of the concurrency before our modifications was based on the pattern we have just presented. However, it had similar problems to those outlined in the previous `grcov` section3.1.2. In particular, the implementation lacked modularity, as it did not separate the logic into simple and reusable functions that could be called independently across different parts of the code. Instead, it relied on duplicated code and used huge functions into which the entire logic was inserted. This approach significantly increased the effort required by developers to identify and understand code sections that implement the various components that are part of the `composer-consumers-composer` pattern. Therefore, it seemed necessary to completely refactor this section of the tool, by adopting crates and good programming practices aimed at simplifying and, at the same time, improving the concurrent implementation.

We started the redesign of the multi-threading execution code by creating a new module called `concurrent`, and clearly defining the structure of the concurrent execution mechanism pattern within it. We achieved this by creating specific functions for the producer, consumer, and composer components and defining their interaction and data exchange logic. However, we did not specify a priori how the consumer processes the files or how the composer produces the final result. This approach enabled us to create a more general and reusable structure, eliminating redundant code and condensing the pattern logic into a few lines of code.

The specific instructions performed by a producer, a consumer, and a composer are defined separately, thus allowing the concurrent pattern implementation to be detached from the tool's actual algorithm. This approach proved very useful at the beginning of the refactor, as we initially maintained separate workflows for the files and functions modes. Therefore, we were able to define the behaviour of pattern components for each mode without having to repeat the entire concurrent logic.

Subsequently, we further simplified the tool's workflow by eliminating the separation between the two modes and putting in different files the mode-specific operations. These operations are then appropriately called within the single obtained workflow, resulting in a more maintainable structure, and drastically reducing the complexity of the concurrent code.

For the implementation of this channel-based thread communication, we have used the `crossbeam` library. Instead, we relied on `rayon` to manage the aspects related to the creation of a thread. This crate provides specific constructs API that can be used to compactly create multiple threads and manage their results.

Once we had completed the `concurrent` module, we realized that, together with the `grcov` module, we had all the necessary elements to entirely remove the workflow separation pointed out by the four cases presented before: *files-coveralls*, *files-covdir*, *functions-coveralls*, and *functions-covdir*. Therefore, these two modules simplified the workflow and removed redundant code, resulting in a well-structured and more maintainable codebase.

3.1.4 API Redesign

Another fundamental part of `weighted-code-coverage` that underwent significant refactoring was API. An API serves as the interface between a developer and a library, so its design should prioritize ease of use and clarity [161]. It is essential that the exposed constructs have concise names that immediately convey their functionalities. Another important criterion for an API is to minimize a developer's effort, ensuring that operations that can be internally managed by a library do not unnecessarily burden a developer [162]. This principle highlights the necessity for API to automate complex tasks and streamline workflows. Therefore, a developer should concentrate only on the application logic of a library instead of on the specific details of its implementation.

The initial version of this API was particularly problematic. It consisted of an excessive number of functions and required developers to know a priori the exact sequence in which these functions should be called. As a result, developers had to invoke several intermediate functions to obtain the final metrics, and explicitly manage the passage among the intermediate results from one function to another. A library should perform these operations internally instead of discharging this responsibility to a developer. Indeed, a developer should only have to input the necessary parameters and obtain the desired results without performing avoidable operations.

Another problem with API was that it exposed a different function for each of the four cases in which the workflow had been initially split. This design choice resulted in an API that lacked cohesion and forced developers to write a large amount of code to manage all the different scenarios. As a result, the initial version of the API suffered from poor maintainability and usability due to its complexity, which increased the likelihood of errors for developers.

For the API redesign, we decided to base our implementation on the builder pattern [163]. The main principle of this pattern is to separate the building process of an object from its internal representation. In our case, the object corresponds to a data structure that contains all the parameters of the analysis performed by `weighted-code-coverage`.

The builder pattern allows us to build this data structure step by step since, for each considered parameter, there is a function that can be called to set its value. Each parameter also has a default value, which will be used if no value has been provided. Another advantage of the builder pattern is that a function used to set the parameters can also validate its input values.

After defining every desired parameter, a developer must call a function named `run`, to start the tool's workflow. This function takes as arguments only two mandatory parameters: the path to the project to be analysed and the path to the `grcov` file.

The functions adopted for setting the various parameters, the `run` function, and some custom data types constitute the new API for the `weighted-code-coverage` library. This API is straightforward to understand because every construct is well-documented, and the functions that set the analysis parameter have names that clearly indicate their purpose. Furthermore, a developer only needs to set the values of the parameters they what to change, keeping the default values for the others, thereby reducing the effort.

As a result, this API implementation is very maintainable and flexible, as it can be easily extended with additional parameters by just adding additional functions to the interface. This redesign has significantly simplified the tool's API, offering to developers a minimal and self-documented interface.

3.1.5 CLI Refactoring

After implementing this new API, we focused on the refactoring of the `weighted-code-coverage` CLI. This refactor came as a natural consequence of the API redesign, as the CLI calls this API and was thus affected by the API's usability problems. The previous API implementation resulted in a binary with significant code repetition, mainly due to the widely discussed four-case distinction. Instead, by adopting the new API, we created a binary which is more compact, eliminating all duplicated code.

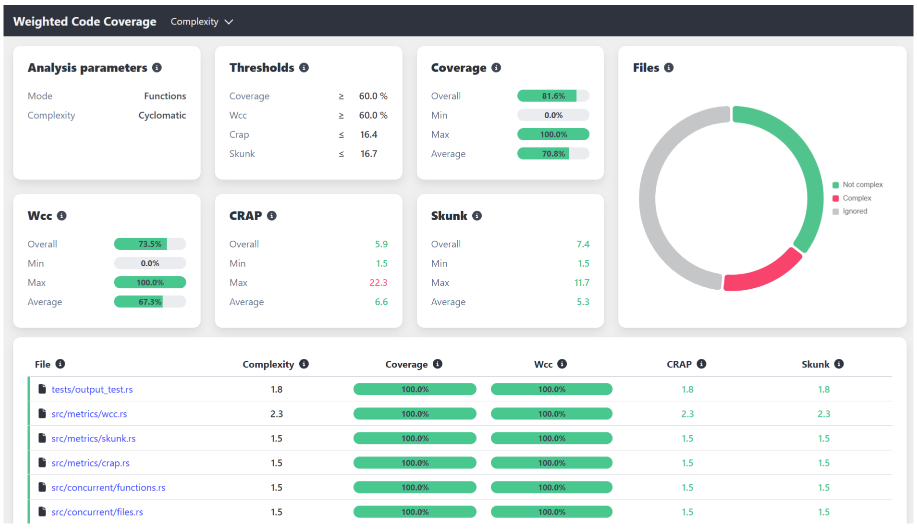
CLI commands allow users to set a series of arguments with the main goal of customizing the analysis. These arguments map one-to-one to the parameters settable through API. Compared to the previous version of the CLI, we removed some arguments that were no longer needed and modified other ones in order to make them more straightforward to use. As a result, users can now set the following arguments:

- **Project Path**: the path of the project to be analysed by `weighted-code-coverage`. This is a mandatory argument.
- **Grcov File Format**: a mandatory argument specifying the `grcov JSON` variant passed as input to the tool. The possible values are: *coveralls* and *covdir*.
- **Grcov File Path**: the path of the `grcov` file containing the coverage information for a project. This is a mandatory argument.
- **Thresholds**: this argument can be used to set the thresholds for the computed metrics instead of the default ones. We will further detail the thresholds in the metrics section 3.1.7.
- **Threads**: this can be used to set the number of threads that the tool will use for the parallel computation of metrics. By default, the tool will use $threads_{max} - 1$.
- **Mode**: specifies which mode to use for the analysis. The possible values are *files* or *functions*, with *files* as the default.
- **Sort**: specifies according to which metric to sort the final result. The possible values are *wcc*, *crap*, and *skunk*. If not specified, the default value is *wcc*.
- **Json Path**: is a mandatory argument that specifies the output path of the `JSON` file containing the analysis process result.
- **Html Path**: an optional argument that can be used if a user also wants to obtain the `HTML` version of the output. If not specified, the tool will only

produce the `JSON` format as output.

To implement the CLI logic, we leveraged the `clap` crate and defined a module called `cli`, within which we inserted all the CLI code. This approach allowed us to obtain a reusable module that can be easily employed for implementing different variations of the CLI binary.

3.1.6 Output



Analysis parameters

Mode	Functions
Complexity	Cyclomatic

Thresholds

Coverage	≥	60.0 %
Wcc	≥	60.0 %
Crap	≤	16.4
Skunk	≤	16.7

Coverage

Overall	81.6%
Min	0.0%
Max	100.0%
Average	70.8%

Files

Wcc

Overall	73.5%
Min	0.0%
Max	100.0%
Average	67.3%

CRAP

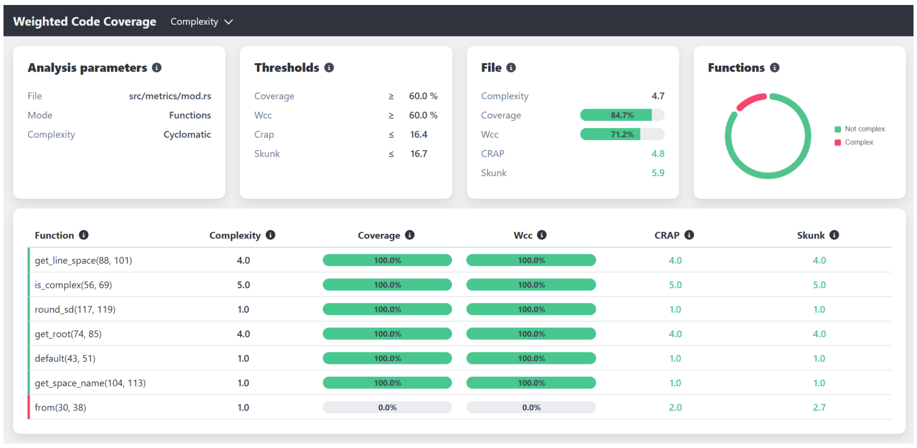
Overall	5.9
Min	1.5
Max	22.3
Average	6.6

Skunk

Overall	7.4
Min	1.5
Max	11.7
Average	5.3

File	Complexity	Coverage	Wcc	CRAP	Skunk
tests/output_test.rs	1.8	100.0%	100.0%	1.8	1.8
src/metrics/wcc.rs	2.3	100.0%	100.0%	2.3	2.3
src/metrics/skunk.rs	1.5	100.0%	100.0%	1.5	1.5
src/metrics/crap.rs	1.5	100.0%	100.0%	1.5	1.5
src/concurrent/functions.rs	1.5	100.0%	100.0%	1.5	1.5
src/concurrent/files.rs	1.5	100.0%	100.0%	1.5	1.5

Figure 3.4: Files mode HTML output



Analysis parameters

File	src/metrics/mod.rs
Mode	Functions
Complexity	Cyclomatic

Thresholds

Coverage	≥	60.0 %
Wcc	≥	60.0 %
Crap	≤	16.4
Skunk	≤	16.7

File

Complexity	4.7
Coverage	84.7%
Wcc	71.2%
CRAP	4.8
Skunk	5.9

Functions

Function	Complexity	Coverage	Wcc	CRAP	Skunk
get_line_space(88, 101)	4.0	100.0%	100.0%	4.0	4.0
is_complex(56, 69)	5.0	100.0%	100.0%	5.0	5.0
round_sd(117, 119)	1.0	100.0%	100.0%	1.0	1.0
get_root(74, 85)	4.0	100.0%	100.0%	4.0	4.0
default(43, 51)	1.0	100.0%	100.0%	1.0	1.0
get_space_name(104, 113)	1.0	100.0%	100.0%	1.0	1.0
from(30, 38)	1.0	0.0%	0.0%	2.0	2.7

Figure 3.5: Functions mode HTML output

The refactoring of the output produced by `weighted-code-coverage` is one of the aspects we have worked on the most. Indeed, this is a crucial aspect of this tool since it should provide users with a comprehensive view of the entire analysis. The output should present the computed metrics in a clear and concise manner, allowing developers to easily assess the quality of their codebase. It should highlight potential problems, and offer insights into the code maintainability. Additionally, the output should be well-organized and formatted, making it easy for developers to quickly locate specific information and take informed decisions about necessary code improvements.

The initial version of the CLI version allowed the production of `JSON`, `CSV`, or `HTML` as outputs, all of which were optional. We initially maintained this approach, only to later realize that it would be more effective to always produce at least one output format. We believed that CLI users would have been more interested in viewing the analysis results instead of getting the outputs from the crate. We chose the `JSON` format for this purpose because it strikes a good balance between usability and the number of generated files, encapsulating all project metrics in a single file. Instead, the specific API contained in the crate allowed users to obtain a data structure containing all metrics results without forcing the creation of any output file. We also decided to drop the `CSV` output format due to its lack of clarity and entirely focus on the `JSON` and `HTML` variants.

The changes we implemented for the `JSON` output directly reflect the changes to the data structure containing all the final metrics. The `JSON` output is simply a serialization of this data structure, containing the same information as the `HTML` output. The advantage of `JSON` is that information is represented in an easily serializable format, which is widely used among developers, and it can be seen as the machine-readable version of the output, suitable for being parsed by other tools and systems. Conversely, the HTML output has been choses as a more user-friendly alternative due to its possibility for better displaying metrics in a clear way.

The initial version of the `HTML` output, despite containing all the analysis information, was very minimal and not simple at all to understand. Our contribution involved a complete output redesign with the objective of making it as intuitive as possible, allowing developers to comprehend the information about the project quickly.

The first thing we have done in our refactoring is to replace the previous template engine crate with `minijinja`. This crate provides a lightweight, minimal, yet powerful solution for creating even complex software based on templates.

The outputs vary slightly depending on whether the files or functions mode has been selected. Every mode starts from a common base template that displays every file in a project, but the functions mode adds the possibility of displaying all the functions contained in each file, along with its respective metrics.

As shown in Figure 3.4, we have chosen a data dashboard as a visualization style.

This allows us to present the information, on a single page broken down into different cards. Initially, we structured the page using only a series of tables. However, we then realized that these tables did not provide a clear and immediately understandable view of the metrics.

The first card shows the common parameters of our analysis, such as granularity mode and complexity type. In fact, as we will further describe, the tool can compute metrics using both cyclomatic and cognitive complexity. In the initial tool version, a developer had to specify which type of complexity to consider. We have noted that it would be inconvenient for a user running the tool to have to run the tool twice in order to obtain the metrics computed with both cyclomatic and cognitive complexity. As a result, the `HTML` offers the possibility to switch between the metrics computed with complexity metric through a dropdown menu in the navigation bar called *Complexity*.

Next to the card that shows the analysis parameters we have placed the *Thresholds* one, which displays the metrics thresholds.

The following cards are for *Coverage*, *Wcc*, *CRAP*, and *Skunk*, the `weighted-code-coverage` metrics. These cards show the total, minimum, and average values for each metric

The *Files* card shows a doughnut chart that provides a general overview of every file in a project. The *Not complex* category includes files whose metrics do not exceed the thresholds. Instead, the *Complex* category includes files whose at least one of the thresholds is exceeded. Finally, the *Ignored* category includes files for which we cannot retrieve profiling and coverage data, thus those files are excluded from the analysis. Clicking on the section of the chart corresponding to the *Ignored* icon shows a complete list of these files

In the bottom section of the page, there is a table where every row represents a file of a project, along with its path and the corresponding metric values. A green vertical line next to the file name indicates that that file is *Not complex*, while a red vertical line indicates it is a *Complex* one.

As mentioned before, whenever the functions mode is chosen, we can reach the corresponding metrics values, in addition to other information, by clicking on the function name in the table. This event takes a developer to a new page completely dedicated to the chosen function This page, as seen in Figure 3.5, shows the same fie metrics present in the previous page, along with an additional table containing the list of functions metrics.

3.1.7 Metrics Analysis

Now, we are going to describe the core aspect of `weighted-code-coverage`, which is the implemented metrics and how they have been computed. We start by describing the state of the metrics before our contribution and highlighting the problems we noticed during the refactoring. After that, we are going to list all the modifications we have made to address these problems and present the new metrics' state. To better explain the changes applied to metrics, we will build some concrete scenarios by referring to the snippet of code shown in Listing 3.1.

```
function f1() {
    instruction a
    instruction b
    instruction c
}

function f2() {
    instruction d
    instruction e
    instruction f
}
```

Listing 3.1: Example code snippet

This code has a SLOC2.5.1 value of 11 and a PLOC value of 10.

The initial version of the tool implemented four metrics: *Wcc plain*, *Wcc quantized*, *CRAP* and *Skunk*. However, before starting our description, we first need to introduce the **code space** concept. It refers to any programming language construct which contains a function, such as functions themselves, classes, methods, structs or closures. In the example code, we can identify two code spaces: functions `f1` and `f2`. Note that we are assuming that the instructions inside the two functions do not define other internal code spaces.

To compute the *Wcc quantized* from a source file, we need to analyse the file line by line and apply on each line the following algorithm:

- The algorithm starts by initializing to 0 a variable that accumulates the weights associated with the considered lines.
- When a file does not have PLOC lines, the *Wcc quantized* value for that file is equal to 0. Otherwise the algorithm proceeds to analyse every single PLOC line.
- For each PLOC line it checks whether the line is covered.
- If a line is not covered the tool proceeds to analyse the following PLOC line.

- Otherwise, if the line is covered the algorithm retrieves the complexity from the code space associated with that line. If the complexity value is greater than 15, the weights variable is incremented by 2. Instead, when the complexity value is less than or equal to 15, the weights variable is incremented by 1. Note that the same algorithm is adopted for both cyclomatic and cognitive complexity.
- Finally, after parsing all lines, the algorithm computes the *Wcc quantized* value for the file under analysis by dividing the weight value by the total number of PLOC lines in that file.

If the tool is run in functions mode, it uses the exact same algorithm to compute the metrics for each function. The only difference is that it will only analyse the lines of code of the function, so the PLOC value will correspond to the PLOC of the function. Regarding complexity, the algorithm determines the complexity value based on the innermost code space that encapsulates the line in question. This means that if a line belongs to an inner closure within the function, the complexity value used corresponds to the closure's complexity rather than that of the function itself.

To highlight this metric's issues, we present some specific scenarios, using the code in Listing 3.1 as a reference:

- **Worst case**: we start by considering the worst-case scenario, in which the code under analysis is poorly covered and very complex. For example, we suppose that both functions *f1* and *f2* have only 2 lines out of 5 covered by the tests, and that both have a complexity greater than 15. In this case, we have that the *Wcc quantized* of the file is equal to:

$$wcc\_quantized(f) = \frac{weights}{PLOC} = \frac{2+2+2+2}{10} = \frac{8}{10} = 0.8 \tag{3.1}$$

  where we have that since there are a total of 4 lines covered and they have a complexity greater than 15, they will all have a weight of 2.
- **Balanced case**: we now consider the case where the first function `f1` has a coverage of 3 out of 5 lines and complexity equal to 10, thus not very low but still less than 15. Instead, `f2` has coverage equal to 2 and complexity much lower than 15. So we have that:

$$wcc\_quantized(f) = \frac{(3 \times 1) + (2 \times 1)}{10} = \frac{5}{10} = 0.5 \tag{3.2}$$

- **Balanced case**: if we consider a `f1` as the one of the previous case, but a `f2` that has coverage equal to 4 lines and complexity greater than 15, we have that the metric is equal to:

$$wcc\_quantized(f) = \frac{(3 \times 1) + (4 \times 2)}{10} = \frac{11}{10} = 1.1 \tag{3.3}$$

- **Best case**: finally, we take into account the best case scenario, which is a highly covered file with low complexity. In particular we consider maximum coverage and a complexity lower than 15 for both functions. The *Wcc quantized* value is equal to:

$$wcc\_quantized(f) = \frac{10 \times 1}{10} = \frac{10}{10} = 1 \tag{3.4}$$

Upon examining these results, it is evident that there are some inconsistencies between the obtained values and the scenarios considered. First of all, when transitioning from the worst case to the first balanced case, we observe a decrease in the metric value despite an increase in the number of covered lines and a reduction in complexity. This contradicts the expected behaviour of the *Wcc quantized* metric, which should ideally increase with higher coverage and reduced complexity. However, in our case, the metric value decreases from 0.8 to 0.5, which is not consistent with the intended design.

When comparing the second balanced case with the best case, we observe another anomaly. In this scenario, a balanced case exhibiting high coverage and high complexity, results in a *Wcc quantized* value greater than that of the best case. Moreover, the value of the balanced case is equal to 1.1, which is odd because the *Wcc quantized* metric, like coverage, was designed to be expressed as a percentage. In this case, the percentage would be 110%, which is incorrect as it exceeds the 100% value of the best case.

We now analyse the second originally implemented metric, known as *Wcc plain*. Its algorithm closely resembles that of *Wcc quantized*.

The first difference is that the algorithm computes a file's metric using the total complexity of the entire file, rather than the complexity value of each line's code space. Instead, for computing the metric of a single function, the algorithm uses the complexity value of the entire function rather than that of the inner most scope that encapsulates the line.

The second difference is in how the algorithm analyses each covered line. Instead of incrementing the weights partial sum by 1 or 2 based on a complexity threshold, for each line, it adds to weights the total complexity value. The *Wcc plain* value is subsequently derived by dividing the total weight sum by the PLOC.

In the following scenarios, we focus solely on computing the metric for a file. However, the issues highlighted also manifest when computing metrics for single functions:

- **Worst case**: we assume a file complexity value of 20 and a coverage of 2 covered lines per function. Therefore, the metric value is:

$$wcc\_plain(f) = \frac{weights}{PLOC} = \frac{4 \times 20}{10} = \frac{80}{10} = 8 \tag{3.5}$$

- **Balanced case**: keeping the same coverage value of the worst-case and decreasing the complexity to 2, we obtain:

$$wcc\_plain(f) = \frac{4 \times 2}{10} = \frac{8}{10} = 0.8 \tag{3.6}$$

- **Balanced case**: we assume a coverage of 4 out of 5 covered lines for each function, and a complexity equal to 20. The metric value is:

$$wcc\_plain(f) = \frac{8 \times 20}{10} = \frac{160}{10} = 16 \tag{3.7}$$

- **Best case**: for the best case we consider maximum coverage and a low complexity equal to 1.0. The *Wcc plain* value is:

$$wcc\_plain(f) = \frac{10 \times 1}{10} = \frac{10}{10} = 1 \tag{3.8}$$

We notice that the increasing and decreasing trend of the metric values is equal to the one observed for *Wcc quantized*. However, *Wcc plain* values exhibit even more pronounced anomalies than before. Notably, some values are much higher than 1, which should ideally be the maximum value. Moreover, the worst-case scenario exhibits a value eight times higher than the best-case. This clearly illustrates the critical nature of this metric.

Given the issues with the previous two metrics, we attempted to find a solution to overcome them. First of all, we decided to eliminate the distinction between *Wcc quantized* and *Wcc plain*, and merge them in a single metric called *Wcc*. This decision was made because our analysis revealed that *Wcc plain* is essentially a less restricted version of *Wcc quantized*, with the same issues and even less accuracy.

As a result, *Wcc* builds upon the concepts and ideas of the previous two metrics and aims to resolve their issues. *Wcc*'s algorithm is almost identical to the *Wcc* *quantized* one. The only difference is that in *Wcc quantized*, when it encounters a line from a code space of complexity greater than 15, a weight of 2 is added to the weights, whereas in *Wcc*, such lines are directly ignored, leaving the weights unchanged.

In other words, *Wcc* excludes from the metric calculation all lines that, despite being covered, belong to a code space that is too complex. This slight modification significantly alters the metric values, making them consistent with the cases under analysis. If we revisit the exact same scenarios considered for *Wcc quantized*, using the same coverage and complexity values, we obtain the following results:

- **Worst case**: 2 covered lines and complexity greater than 15 for both functions:

$$wcc(f) = \frac{weights}{PLOC} = \frac{0}{10} = \frac{0}{10} = 0 \tag{3.9}$$

- **Balanced case**: 3 covered lines for the first function and 2 for the second, with complexity lower than 15 for both:

$$wcc(f) = \frac{3+2}{10} = \frac{5}{10} = 0.4 \tag{3.10}$$

- **Balanced case**: 3 covered lines and complexity lower than 15 for `f1`, 4 covered lines and complexity greater than 15 for `f2`:

$$wcc(f) = \frac{3}{10} = 0.3 \tag{3.11}$$

- **Best case**: 10 total covered lines and complexity lower than 15 for both functions:

$$wcc(f) = \frac{10 \times 1}{10} = \frac{10}{10} = 1 \tag{3.12}$$

To provide a clearer evaluation of these results, we can express them as percentages:

- **Worst case**:

$$wcc_{\%}(f) = wcc(f) \times 100 = 0 \times 100 = 0\% \tag{3.13}$$

- **Balanced case**:

$$wcc_{\%}(f) = 0.4 \times 100 = 40\% \tag{3.14}$$

- **Balanced case**:

$$wcc_{\%}(f) = 0.3 \times 100 = 30\% \tag{3.15}$$

- **Best case**:

$$wcc_{\%}(f) = 1 \times 100 = 100\% \tag{3.16}$$

Now, the best-case scenario achieves a 100% percentage, whereas the other three cases exhibit lower percentages. In addition, the balanced cases have higher percentages compared to the worst-case scenario. This aligns with the intended behaviour of the metric that we wanted to obtain. Upon further analysis of the algorithm, it becomes evident that *Wcc* values will never exceed 100%. This is due to the exclusion of lines within code spaces with complexity exceeding 15, even if adequately covered by tests.

This is the behaviour we wanted to get from the metric Moreover, analyzing the algorithm further, we realize how it will never be possible to have a value of *Wcc* greater than 1, thus greater than 100%. In fact, all code spaces with complexity greater than 15 are excluded, despite perhaps being reasonably covered by tests. While this approach may seem too strict, it is justified for code segments with a complexity greater than 15 to have a *Wcc* of 0. This is because the maximum complexity value should typically fall within the range of 10 to 15, although it's generally recommended to keep it below 10 [164]. In fact, a complexity greater than 15 suggests that the function is extremely complex and should be refactored. This implies that having a fully covered function is not sufficient to obtain a positive evaluation of the analysis, and the metric calculated in this way helps to identify these scenarios.

Wcc can be seen as a sort of coverage weighted by complexity. Therefore, as the default threshold, we decided to adopt the same 60% value, typically used for coverage. However, it is possible to change this value using the function provided by the API.

The *Wcc* value of the entire project is computed using the same algorithm previously described. In fact, as each file undergoes analysis, the tool records the PLOC and weight values for each of them. In the end, it sums up the weights of all files and divides this total by the aggregate PLOC, thereby deriving the project's *Wcc*.

CRAP

The next metric we analysed is *CRAP* [165], which can be represented by a mathematical expression. Let c be a code space with coverage equal to $cov(c)$, where $cov(c) \in [0, 1]$, and complexity equal to $comp(c)$, then $CRAP$ of c is defined as:

$$CRAP(c) = comp(c)^2 \times (1 - cov(c))^3 + comp(c) \tag{3.17}$$

This metric, unlike previous ones, is a score, which means that it should be kept as low as possible. Indeed, a high value of *CRAP* suggests that the code is either poorly covered, too complex, or both. Instead, a low value means that the code is well-covered and not too complex.

The presented equation allows to compute *CRAP* score of a code space, but it can also be used to compute the metric for an entire file. In that case, *comp* corresponds to the average complexity of code spaces within the file, and *cov* represents the file's coverage. Moreover, to compute the *CRAP* score for the entire project, *comp* reflects the average complexity of code spaces within the project, and *cov* represents the project's overall coverage.

Initially, we explored the possibility of converting all metrics implemented by `weighted-code-coverage` into percentages. This approach could have provided more representative values and simplified threshold definitions. However, this was only feasible for *Wcc*. Score metrics, in contrast, cannot be easily expressed as percentages because a score can grow indefinitely, making it impossible to define a maximum value and, consequently, a range for percentage conversion.

The *CRAP* threshold is defined by computing the metric value using a coverage of 60% and a complexity of 10. We consider this an average case and use its value as the default threshold for the metric. In fact, 60% represents the typical coverage threshold, while a complexity value of 10 aligns with the complexity range considerations discussed previously. This results in a threshold of:

$$CRAP_{thr} = 10^2 \times (1 - 0.6)^3 + 10 = 16.4 \tag{3.18}$$

CRAP does not have particular problems that require us to modify the way values are computed. However, it does have some accuracy issues which we will discuss later when comparing the various metrics.

Skunk

Skunk is another score metric, which can also be represented through a mathematical expression. Let c be a code space with coverage equal to $cov(c)$, where $cov(c) \in [0, 1]$, and complexity equal to $comp(c)$, then *Skunk* of **c** defined as:

$$Skunk(c) = \frac{comp(c)}{COMPLEXITYFACTOR} \times (100 - cov(c)) + comp(c) \tag{3.19}$$

where $COMPLEXITYFACTOR$ is an empirically obtained value, equal to 60.

To compute a file's *Skunk*, we have to substitute *comp* with the average complexity of code spaces within the file, and *cov* with the file's coverage. Instead, to compute the score for the entire project, we have to substitute *comp* with the average complexity of code spaces within the project, and *cov* with the project's overall coverage.

However, the presented formula does not correspond to the initial expression of the metric. In fact, the previous version of the tool implemented an algorithm following this representation:

$$Skunk(c) = \begin{cases} \frac{comp(c)}{COMPLEXITYFACTOR}, & if cov(c) = 100 \\ \frac{comp(c)}{COMPLEXITYFACTOR} \times (100 - cov(c)), & \text{otherwise} \end{cases} \tag{3.20}$$

where $COMPLEXITYFACTOR$ is equal to 25.

This expression corresponds to the one presented in the original documentation of the *Skunk* metric [166]. However, while examining this expression we noticed that it presents several issues. First of all, if we consider a code space with a complexity equal to 50 and a maximum coverage of 100%, the Skunk value is equal to 2:

$$Skunk(c) = \frac{comp(c)}{COMPLEXITYFACTOR} = \frac{50}{25} = 2 \tag{3.21}$$

This value is anomalous because a highly complex function should not have such a low *Skunk* score. To address this issue, we modified the original expression as follows:

$$Skunk(c) = \frac{comp(c)}{COMPLEXITYFACTOR} \times (100 - cov(c)) + comp(c) \tag{3.22}$$

where $COMPLEXITYFACTOR$ remains equal to 25.

This modification resolves the previous issue. However, we observed that when computing the threshold value, assuming, as for *CRAP*, a coverage of 60% and a complexity of 10, the threshold becomes 26:

$$Skunk(c) = \frac{10}{25} \times (100 - 60) + 10 = 26 \tag{3.23}$$

This allows code spaces with complexities up to 26 to be accepted by the metric. For instance, a code space with maximum coverage of 100% and a complexity of 26 would have a *Skunk* score of 26:

$$Skunk(c) = \frac{26}{25} \times (100 - 100) + 26 = 26 \tag{3.24}$$

Instead, defining a $COMPLEXITYFACTOR$ equal to 60 yields a threshold value of 16.7:

$$Skunk_{thr} = \frac{10}{0.6} \times (1 - 0.6) + 10 = 16.7 \tag{3.25}$$

Therefore, based on these considerations, we decided to further modify the formula and adopt a $COMPLEXITYFACTOR$ equal to 60. However, despite these modifications, **Skunk** is the most problematic and least accurate metric. In fact, the authors do not adequately document how the initial $COMPLEXITYFACTOR$ of 25 was obtained. Furthermore, as stated in the conference video in which the metric was presented, it is defined as a sort of magic number [167].

Metrics Comparison

When comparing the three metrics, *Wcc* stands out as the most accurate. It achieves this by effectively balancing coverage and complexity. As mentioned earlier, *Wcc* can be briefly described as coverage augmented by complexity information. It first checks whether a line is covered and only then evaluates complexity to decide whether or not to consider the line's weight. This ensures that in the best case scenario, when all lines in the file are covered and none of them belongs to a code space of complexity greater than 15, the value of *Wcc* will be exactly equal to the coverage, without ever exceeding it.

In contrast, *CRAP* and *Skunk* prioritize complexity over coverage. For example, if we consider a file f with 0% coverage and an average code space complexity of 3, the *CRAP* score would be equal to:

$$CRAP(f) = 3^2 \times (1 - 0)^3 + 3 = 12 \tag{3.26}$$

and the *Skunk* would be:

$$Skunk(f) = \frac{3}{60} \times (100) + 3 = 8 \tag{3.27}$$

Despite the file being entirely uncovered, these scores are considered acceptable since they fall below the threshold. Therefore, *CRAP* and *Skunk* metrics are perceived as less accurate because they require complexity to exceed a certain threshold to achieve a high score, even when coverage is minimal. Moreover, *Skunk* is considered less accurate than *CRAP* due to its slower growth rate and the ambiguities surrounding the definition of the $COMPLEXITYFACTOR$.

3.1.8 Testing

For the tool's testing, we replaced the previous unit and integration tests, which were based on standard Rust testing methodologies, with tests implemented using snapshots. Snapshots provide a mechanism to capture the output of a test and compare it against a reference value stored in a file. This method is particularly useful in situations where a developer needs to test the values of large data structures. To implement these tests, we used the `insta` crate, which offers a lightweight and easy-to-use solution for this type of testing.

The old tests were cumbersome and verbose because developers had to manually create and populate data structures with expected metrics values. This approach greatly reduced testing maintainability, because if a metric parameter changed, causing the metric values to change, the developer had to manually update all the expected values in every test. Snapshots, on the other hand, allow us to quickly update the reference values against which we want to compare the tool's result.

In particular, we implemented integration tests to thoroughly assess all functionalities and potential execution scenarios of `weighted-code-coverage`, covering the diverse parameters settable via the API. In addition to this, we have also defined unit tests to verify the behaviour of the new `grcov` module we created, and ensure that the files are parsed correctly.

In conclusion, this new implementation allowed us to provide a comprehensive evaluation of `weighted-code-coverage`. Furthermore, the adoption of a more modular structure has streamlined the testing process and enhanced its maintainability, facilitating an eventual expansion of the test suite in the future.

3.1.9 Final Remarks

The refactoring of `weighted-code-coverage` illustrates how poor initial API design and implementation choices can have a detrimental impact on software, compromising its usability and maintainability.

For instance, the decision to split the tool's execution into four cases based on mode and `grcov` file variant led to extensive code repetition and significantly decreased code readability. This design choice not only complicated the codebase but also introduced challenges in maintaining and extending the software's functionality.

These challenges emphasize the importance of thoughtful design and implementation strategies from the project's beginning. Furthermore, they highlight the need for scalable and modular programming practices, to ensure the software's long-term viability and ease of development.

3.2 generate-ci

`generate-ci`[6] is a tool designed to facilitate the creation of build systems and Continuous Integration (CI) configuration files using predefined templates. It supports various programming languages, allowing developers to generate project-specific configurations for tasks such as building, testing and static or dynamic analysis. Among the tasks related to static analysis, there is also one dedicated to `weighted-code-coverage`.

In fact, our contribution to `generate-ci` focused on enhancing the integration of `weighted-code-coverage` analysis within the CI workflow automatically generated by this tool. Initially, the workflow included basic integration of `weighted-code-coverage`, which only exposed the `JSON` output file containing all computed project metrics. To enhance the usability and visibility of the results produced during the analysis, we have modified the CI workflow in order to have in output the `HTML` format of `weighted-code-coverage`. This update also defined a command to set up a dedicated URL for accessing the produced HTML pages.

Furthermore, we enriched the `weighted-code-coverage` task by adding a badge creation. This badge displays the overall *Wcc* metric value for the project under consideration and offers a direct link to the hosted `HTML` output once clicked. At last, we have updated the documentation templates generated by `generate-ci` to prominently integrate this badge.

Our contribution significantly improved the visualisation and accessibility of the `weighted-code-coverage` analysis performed within the CI workflow generated by `generate-ci`. Developers can now quickly access and navigate the analysis reports, allowing a complete evaluation of a project.

[6]https://github.com/SoftengPoliTo/generate-ci

3.3 complex-code-spotter

The last tool we worked on is `complex-code-spotter`[7], which is a static analysis tool that extracts pieces of code deemed complex from the source code of a project. The main objective of this tool is to help developers identify code snippets that are too complex, and can thus be difficult to understand or may increase the likelihood of introducing errors.

The current version of `complex-code-spotter` evaluates the cyclomatic and cognitive complexity of a code fragment. When the snippet exceeds the threshold for either of the two metrics, it is extracted and saved. This tool is very flexible and allows developers to customize the analysis by specifying a different threshold instead of the default value of 15. Additionally, developers can specify the file extensions to include in the analysis for a more focused complexity evaluation.

The tool's workflow heavily leverages multithreading to parallelize and optimize the analysis. The initial concurrent execution mechanism was based on the `producer-consumers` pattern, a variation of the `producer-consumers-composer` pattern that uses a shared data structure among consumers instead of a final composer. In `complex-code-spotter`, this shared data structure is used by consumers to store the code snippets that exceed the complexity thresholds.

However, the implementation of this pattern was verbose and thus difficult to maintain. Therefore, we decided to replace it with the *producer-consumers-composer* pattern created in the context of `weighted-code-coverage`. This allowed us to significantly streamline the workflow of `complex-code-spotter` and remove a great amount of code, enhancing the tool's maintainability and readability. Since the implementation of the `producer-consumers-composer` pattern in `weighted-code-coverage` was designed with reusability in mind, it was straightforward to adapt it for `complex-code-spotter` with minimal effort.

[7]https://github.com/SoftengPoliTo/complex-code-spotter

Chapter 4

Hazard Generator

Internet of Things devices gather data from the physical environment in order to perform specific actions. These actions can be a direct response to a customer request, such as switching on a light, or a consequence of a previously set configuration, such as automated shutters that open and close at specific times each day. Furthermore, devices can also act autonomously according to the data collected on a daily basis, such as irrigation systems, which activate themselves depending on weather conditions and soil humidity.

The firmware drives all these actions and the overall behaviour of smart devices, ensuring that they promptly respond to customer commands and operate as intended. It serves as the interface to the hardware components, reading data collected by sensors and sending commands to actuators. Therefore, firmware is a fundamental but also critical part of IoT systems, as errors in firmware can lead to extremely dangerous situations in terms of safety and security. For example, faulty firmware for light devices could cause overheating and trigger a fire. Similarly, issues in appliances such as dishwashers and washing machines could lead to flooding and potential shock threats if water comes into contact with any electrical components.

As mentioned in section 2.6, the lack of well-structured certifications in the IoT landscape, particularly for firmware which controls smart devices, represents a significant problem. This gap increases people's scepticism and fears regarding the security of these systems, as highlighted in the user adoption section 2.2.3. Therefore, a certification process must also consider the description of a device's behaviour and the potential dangers that may arise from its actions.

All these concerns broadly apply to the entire Internet of Things ecosystem. However, in this chapter, we focus on the Smart Home domain. In fact, the dangers and firmware certification problems discussed above are highly relevant for Smart Home devices, which often play an essential role in everyday living environments.

In particular, we concentrate on the behaviour of Smart Home devices and on the risks that may arise from their actions. We model these risks using an ontology, which is a structured way of representing the knowledge of a specific domain of interest. Within the ontology, the risks are referred to as hazards. This part of the thesis aims to implement a tool that parses the hazard ontology and generates APIs for the Rust language. These APIs can then be integrated into a crate for developing Smart Home firmware in order to represent the hazards.

4.1 Hazards

Hazards represent dangerous situations that can arise as a consequence of actions issued by Smart Home devices. These hazards could occur due to firmware errors or device defects that alter its normal operation, or because of an incorrect device use. Inexperienced Smart Home residents, unfamiliar with such technologies, could unintentionally endanger themselves and other inhabitants by misusing some of the functions provided by a device. For example, it is possible to mistakenly switch off the monitoring system of a house, potentially exposing it to intrusions that would go undetected. In that case, turning off the monitoring system could result in the hazard of strangers intruding into the home.

To describe and classify hazards, we started from the hazard concept defined within SIFIS-Home[1], which is a research project that implemented a secure-by-design framework for developing Smart Home systems, ensuring safety, security, and privacy for the inhabitants of a house. This framework defines APIs that represent, at a higher level, the actions performed by a Smart Home device. For example, for a light device, one of its actions is switching it on, and the way this is done changes depending on how the manufacturer has implemented the device. As a result, the framework provides APIs that represent these device-specific actions, offering a standardized approach to managing and controlling the actions of Smart Home devices.

This approach has the advantage that, given every action is represented by a corresponding API, the framework can ensure a more reliable and consistent management of each action. In particular, the framework assigns each API a label containing a list of all possible hazards that may arise as a consequence of executing the action. Each hazard can belong to only one of the following three

[1]https://www.sifis-home.eu/

categories:

- **Safety**: hazards that may endanger the physical safety of the inhabitants. Examples include `AirPoisoning`, which indicates that an action may release toxic gases, and `FireHazard`, which indicates that an action may lead to a fire.
- **Privacy**: hazards that may compromise personal and sensitive data, affecting the privacy of the residents. For example, `TakePicture` warns that an action could lead to image acquisition, while `AudioVideoStream` indicates a potential audio or video recording.
- **Financial**: hazards that may lead to unwanted expenses, for example, as a consequence of excessive electricity or gas consumption by a device.

For defining a hazard, it is also possible to use a slightly different variant that allows the assigning of a risk score to it. This risk score is a numerical value between 1 and 10 that quantifies the severity of a hazard. For example, turning on the oven might have an `ElectricEnergyConsumption` hazard with a risk score of 8, while lowering the refrigerator's temperature might have the same hazard with a risk score of 5. This indicates that the oven's energy consumption, and thus the associated costs, is higher than those of the refrigerator.

Firmware producers assign these risk scores during the firmware development phase, allowing for extensive diversification of the associated hazards, even among devices of the same type. In fact, even if two devices of the same type share a hazard for a given action, their risk scores might be very different.

However, this hazard variant, and particularly the process to follow in order to assign the risk score, is still in its early stages and faces some issues. For example, if we consider two devices of type light, such as an LED light and a halogen bulb, they both have an action to turn the light on, which should be associated with a `FireHazard`. Since the overheating risk for an LED light is smaller than for a halogen bulb, the risk score for the `FireHazard` of turning on the LED light is expected to be lower than that for the halogen bulb. The problem is that there is no well-established assessment process that follows standardized methodologies for rigorously assigning a risk score value. Therefore, even if it is reasonable to assume that an LED light's fire risk is lower than that of a halogen bulb, a procedure has not yet been established to assess this according to precise rules.

Despite its current incompleteness and instability, we still decided to include the option of using the hazard variant with the risk score in our implementation. In fact, with further refinement, this variant offers great potential since it could enable a more complete and precise description of the hazards associated with an action.

Starting from the hazard concept defined in SIFIS-Home, helped us reduce our efforts in delivering a comprehensive evaluation of device behaviours and associated risks. In fact, it provided us with a strong foundation that thoroughly describes and categorizes the hazards of Smart Home devices, allowing us to leverage it to further expand our knowledge and analysis of the dangers that may arise from their actions.

4.2 Ontology

As we have already mentioned, the hazard concept is modelled using an ontology, which provides a detailed and organized representation of all its characteristics. In computer science, an ontology is a formal representation of a given domain of interest [168]. This representation provides a clear and unambiguous description that can be shared among all entities interested in the domain. A shared understanding facilitates communication among involved entities, ensuring they all refer to the same concepts. Entities are people or computational systems that can communicate with each other by referring to a shared model.

The fundamental components of an ontology are [169]:

- **Classes**: an ontology categorizes the elements of a domain of interest and establishes clear relationships between these categories. Therefore, classes represent the categories into which the domain of interest concepts can be divided. Furthermore, it is possible to define a hierarchy between classes: the vehicle class could have two subclasses, car and truck.
- **Properties**: a class may be characterized by one or more attributes representing the properties an element must have to be considered part of that class.
- **Individuals**: are concrete instances of elements that belong to a given class.
- **Relationships**: represent the relationships that may exist between the various classes of an ontology. Relationships are generally implemented using properties. For example, a `Person` class may have a property called `owns` that links it to a `Car` class, representing the fact that a person can own a car.

With regard to the hazard ontology[2], it defines the following classes:

[2]https://www.sifis-home.eu/ontology/index-en.html

- **Hazard**: represents an hazard. Each hazard is characterized by an `id` property, which allows it to be uniquely identified within the ontology. The class also includes properties such as `name` and `description`, which provide a brief definition of the hazard. Additionally, when using the risk score variant, it is also possible to use the `riskScore` property. Finally, there is a property named `hasCategory` that allows the implementation of the one-to-many relationship between hazard and category. In fact, the value assigned to this property is an `id` that identifies the corresponding hazard category.
- **Category**: represents all the possible hazard categories. Its properties are `id`, `name` and `description`.

In addition to these classes, the original representation of the SIFIS-Home hazard ontology (SHO), from which we started, also includes another class called InteractionAffordance. This class is not defined by the SHO but corresponds to a class defined within the Web of Things (WoT) Thing Description (TD) Ontology[3], which is an ontology that aims to standardize and provide a formal description of IoT ecosystem resources, establishing a common vocabulary to enhance their usability and interoperability [170]. The InteractionAffordance class represents the various types of interactions that may occur between IoT system entities, such as an action issued by a device in response to a user command. In fact, the SIFIS-Home ontology was born as an extension of the TD ontology to associate device actions with their potential hazards, thereby including aspects of behaviour in a device's description. However, in our hazard implementation, we do not explore this part of the SHO and how it relates to the concepts expressed by WoT and within the TD ontology. In fact, we use a subset of the SHO, specifically the part related to the definition and categorization of the hazards.

4.3 JSON-LD

When working with an ontology, it is essential to have a serialized version of it, since this helps automating its processing and ensures it can be standardized for interoperability across different systems and applications. Over time, various formats have been developed, each with its own advantages and drawbacks. The available formats of the hazard ontology are: RDF/XML[4], N-Triples[5], TTL[6] and JSON-LD. Among these, we have opted to use the JSON-LD variant due

[3]https://www.w3.org/2019/wot/td

[4]https://www.w3.org/TR/rdf-syntax-grammar/

[5]https://www.w3.org/TR/n-triples/

[6]https://www.w3.org/TR/turtle/

to its intrinsic characteristics, which allows us to greatly simplify the parsing process.

JSON-LD (JavaScript Object Notation for Linked Data) is an extension of the JSON format originally born to represent Linked Data. Linked Data is a well-established standard that involves structuring data published on the Internet to facilitate linking between different datasets and interconnect the information [171]. In addition, this structuring also makes the data more comprehensible for both humans and machines, enabling easier analysis and automated processing to implement complex algorithms and applications. Therefore, Linked Data was created to address the problem of unstructured and disjointed data, which hinders the full exploitation of the vast amount of data available on the web.

Linked Data fits into the broader context of the Semantic Web, proposed by World Wide Web founder Tim Berners-Lee [172]. The Semantic Web aims to create a Web of interconnected data that are easily understandable and interpretable by a machine. This interconnectedness improves interoperability between data of different domains, facilitating navigation and knowledge discovery. Furthermore, it enhances search engines' efficiency and the quality of their answers to complex queries involving data from different datasets.

Linked Data principles mandate the use of unique identifiers, known as URIs (Uniform Resource Identifiers), and standard data representation formats [173]. URIs facilitate direct referencing and linking to specific data resources, fostering seamless integration and cross-referencing of information.

Various formats exist for representing Linked Data, one of which is JSON-LD, which provides a syntax for expressing Linked Data in the form of a JSON file. One of the main advantages of using JSON-LD is that it is completely compatible with the JSON format, which is already widely known and adopted among developers [174]. This compatibility ensures that developers can utilize existing JSON tools to parse JSON-LD files, thereby minimizing the effort required for adoption and integration.

JSON-LD has a great expressive power that makes it well-suited for defining and representing ontologies [175]. It allows a clear representation of ontology components such as individuals, properties, classes, and relationships. Furthermore, since it is a format for representing Linked Data, it also allows referencing external classes, enabling the creation of an ontology that relies on concepts defined in other ontologies. The strong interoperability of the JSON-LD format and the ability to easily create an ontology leveraging Linked Data concepts allows it to provide a comprehensive representation of the SIFIS-Home [7] ontology. Furthermore, the JSON-LD format also provides a set of algorithms that can be applied in order to facilitate the parsing [176]:

- **Expansion**: the JSON-LD format, such as JSON, uses a key-value pair syntax to represent data objects within a file. Each key corresponds to a class property name and is identified by a URI. In the first part of a JSON-LD file, there is a section called context, defined using the `@context` tag, that provides all the necessary information to interpret the content of a file. The purpose of the context is to define the mapping between the property name used within the file and its corresponding URI. For example, if we define the class `Person` in our ontology with a property `name` representing a person's name, we can specify in the context that `"name": "https://schema.org/name"`. This means that the URI for the `"name"` property is `https://schema.org/name`, and if we navigate to that link, we will be redirected to a web page that describes the meaning of the property. After defining this mapping, we can, for example, define a person named John Smith using the syntax `"name": "John Smith"` instead of `"https://schema.org/name": "John Smith"`. This enhances readability and, at the same time, allows the linking and referencing of different resources. The expansion algorithm resolves property names into their URIs, simplifying file parsing by eliminating the need to use the context to obtain the URI of a property name. As a result, all instances of `"name"` within the file will be expanded to `"https://schema.org/name"`.

- **Compaction**: this algorithm performs the opposite operation to expansion. It exploits the mapping defined in the `context` tag to compact an expanded document, making it more readable by replacing all URIs with their property name.

- **Flattening**: class relationships of the JSON-LD format can be represented by defining a property that contains a value uniquely identifying another object or by directly nesting objects, forming a tree structure. In the second case, for example, if the `Person` class property `"marriedTo"` indicates a person's spouse, this relationship is defined as `"name": "John Smith", "marriedTo": { "name": "Mary Smith", "age": 30 }`. However, this nested representation can overcomplicate the structure and readability of the format, especially if there are huge objects with further nested objects inside. The flattening algorithm removes the nesting by assigning a unique identifier to each tree node, simplifying the structure into an array of nodes. For example, in the case of the `"marriedTo"` relationship, the object `"name":`

[7] https://www.sifis-home.eu/ontology/ontology.jsonld

`"Mary Smith", "age": 30`, which represents the value of the relationship, will be assigned the ID `"_:ms0"`, and then the relationship will be represented as `"name": "John Smith", "marriedTo": {"@id":"_:ms0"}`.

The use of the JSON-LD format has enabled the creation of an ontology that is both highly readable and easy to parse. This format's underlying principles provide a high degree of flexibility, allowing for effortless and seamless potential extensions of the ontology.

4.4 Tool Structure and Workflow

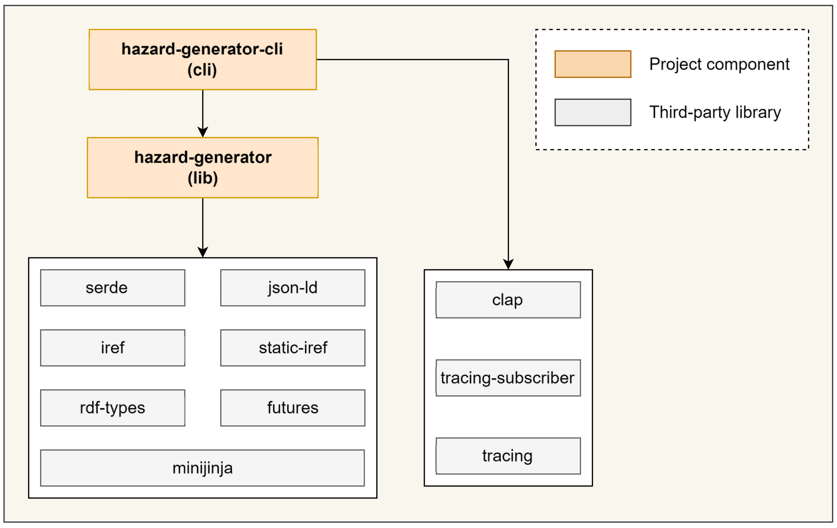


Figure 4.1: hazard-generator structure

As part of the work done in this thesis, we developed a Rust tool called **`hazard-generator`**. This tool takes the JSON-LD serialized version of the hazard ontology as input and parses it to create an API that enables straightforward integration of the hazard concept into firmware development.

The API generated by `hazard-generator` is seamlessly integrated into a Rust crate called `ascot-library`[8], which is used for implementing firmware of Smart Home devices. This new API enhances the crate, providing developers with the necessary constructs to efficiently define and manage the hazards of a device action.

The main objective of `hazard-generator` is to help fill one of the several gaps of Smart Home devices certification. In fact, a robust certification process should

[8]https://github.com/SoftengPoliTo/ascot-firmware/tree/master/src

encompass not only the functional aspects of Smart Home devices but also their behavioural characteristics, emphasizing potential risks and hazards. Using the API generated by `hazard-generator`, developers can effectively outline and integrate hazards during the firmware implementation phase and, at the same time, reduce the overall effort in doing so. As a result, integrating hazards directly into device firmware could streamline the certification process by enabling a more straightforward evaluation of a device's behaviour.

As with every other tool in this thesis, this one is also written in Rust, and its structure is better illustrated in Figure 4.1.

This project comprises both a library and a command-line interface (CLI). The library implements the program's operational logic, while the CLI offers a set of arguments that exploit the library APIs to enable customization of the execution. Using the CLI is straightforward due to its compact design and comprehensive documentation. It provides the following arguments, divided between mandatory and optional:

- **Ontology Path**: this mandatory argument specifies the path to the JSON-LD serialized version of the hazard ontology. Although there is currently only one version of the ontology, this parameter contributes to enhancing the tool's flexibility. In fact, in case of modifications to the ontology, users can easily specify the path to the new JSON-LD file and easily generate an updated version of the API.
- **Template**: `hazard-generator` produces the hazards API only for the Rust language. However, it was designed with the intention to be extendable for also generating APIs for other programming languages, such as C or C++. In fact, its modular structure allows straightforward extension by leveraging the work done for the Rust API. As we will discuss in more detail, we use a template to define the structure of the produced Rust API. Starting from the logic defined in this template, we can adapt and reuse it to implement different templates, thereby generating APIs for various programming languages. Therefore, the CLI's template argument specifies the name of the programming language for which we want to obtain the API, and at the moment, the only possible value is Rust.
- **With Risk**: this optional argument allows the generation of an API that is based on the risk score variant of hazards. In fact, by default, the `hazard-generator` generates an API without the hazard's risk score information. However, we can set this argument to obtain an API that includes the risk score property.
- **Output Path**: this mandatory argument specifies the path to the output

directory where the generated API will be placed.

The workflow implemented within the library can be subdivided into two main operational phases:

- **Ontology Parsing**: we first deserialize and parse the JSON-LD input file in order to extract the necessary information for building the APIs. We use the `json-ld` crate to apply the expansion and flattening algorithms. The expansion resolves the names of the properties in their corresponding URIs. Flattening removes nested structures, converting the parsing tree into a flat one. In this way, each hazard and category is represented by a single node. This facilitates the subsequent parsing process, which involves exploring the tree to determine whether a node is a hazard or a category. We create the corresponding data structure for each node using the extracted information, such as id, name, and description. Eventually, if a user requires the generation of the risk score variant of the API, we also extract that property. In the code, the Rust data structure used to define a hazard includes a field that stores the name of the hazard's category. Meanwhile, the data structure used to represent a category has a field that contains the list of all hazards belonging to that category.

- **API Generation**: after parsing the JSON-LD file, we can proceed to create the actual API. This is done by defining a template that contains the Rust constructs skeleton, which represents hazards and categories within the API. We then use the `minijinja` crate to dynamically fill at runtime the template with the data collected during the previous parsing step. Finally, we use the filled template to create the API file in the specified output path. When generating the API for the risk score variant, we use an extra template in addition to the base one. This other template includes the required Rust constructs that enable us to define a hazard with a risk score value.

To ensure that the tool operates correctly and that it produces the expected APIs, we implemented some integration tests using the `insta` crate. This crate allowed us to create snapshots, which helped us easily check if the data filled into the templates matched the information about hazards and categories of the ontology. Specifically, we created two tests: one to verify the API generated for the variant without the risk score and another to verify the API with the risk score. After confirming through these tests that the produced APIs meet our expectations, we integrated the API variant without risk score into the `ascot-library` crate, thereby successfully achieving the initial objective we set at the beginning of this chapter.

4.5 Final Remarks

`hazard-generator` strives to facilitate and streamline firmware development by providing a tested, flexible, and easy-to-use tool for creating APIs that can be integrated into a Smart Home device firmware library.

These APIs offer concrete support for developing secure and safe firmware, explicitly informing a resident about the potential hazards arising from running device actions in a Smart Home. Furthermore, making these hazards explicit and integrating them within the firmware code opens up the possibility of creating certification processes that, starting from an analysis of the source code, can easily individuate the hazards associated with a device.

Chapter 5

Code Certifier

Through the development of `hazard-generator`, we have successfully implemented a tool that, starting from a hazards ontology, generates a Rust API. This API can be used in firmware development to associate hazards to the actions performed by a Smart Home device. We integrate this API in a crate named `ascot-library`, which defines a set of functions and constructs that allow retrieving the information of a Smart Home device running a server. In fact, as we will explore in detail later, we provide the ability to implement a server that allows the use of specific APIs to interact with the commands defined on a Smart Home device.

Specifically, the `ascot-library` can be used to create the routes through which the device's actions can be controlled. This crate allows us to define the inputs of a route and to obtain information about previously defined routes. The integration of the API created by the `hazard-generator` made it possible to further extend the functionalities provided by `ascot-library`, giving the possibility of defining and retrieving the hazards of a device.

To implement a server running on a device, we use the `ascot-axum`[1] crate, which allows the creation of a local REST server. A REST server is a server which adheres to the principles of REST (Representational State Transfer) architecture [177]. According to these principles, the resources of a server are uniquely identified by URIs, also known as routes. Interaction with these resources is done through HTTP verbs, which define the type of operation to be performed on a resource, ensuring a standardized way of managing them:

- **GET**: this verb allows retrieving a specific resource from the server.

[1]https://github.com/SoftengPoliTo/ascot-firmware/tree/master/ascot-axum/src

- **POST**: creates a new resource on the server.
- **PUT**: updates an existing resource.
- **DELETE**: deletes an existing resource.

In the server implemented using `ascot-axum`, resources correspond to the various actions that can be performed by the device on which the server is running. An action can be controlled through a REST API, which is a combination of a verb and a unique route established by the firmware producer. The choice of verb depends on how the action changes the state of the device. For example, to switch off a light device, the server could provide the following REST API: `PUT 127.0.0.1:3000/light/off`. In this case, the verb is PUT because it updates the state of the light from on to off.

Therefore, device actions are executed using the corresponding REST APIs. Furthermore, when a firmware developer defines the REST API for an action, they also have the option to associate one or more hazards to it using the API added inside `ascot-library`.

`ascot-library` and `ascot-axum` are two crates that belong to a Rust project named `ascot-firmware`[2]. These two libraries enable the creation of firmware while also allowing the description of its behaviour. However, the resulting firmware needs to be certified, and for this purpose, we developed a tool called `code-certifier` that performs two tasks: hazard analysis using the `hazard-analzer` crate and the retrieval of all public APIs from `ascot-library` and `ascot-axum` using `pub-api`. Therefore, `hazard-analyzer` allows us to analyze the source code of a firmware developed using the `ascot-firmware` crates and generate a description of the hazards associated with the device actions. On the other hand, `pub-api` helps in identifying the public APIs defined within `ascot-library` and `ascot-axum` that have been used within a firmware.

The `code-certifier`, whose structure is depicted in Figure 5.1, also includes another crate named `ccertifier`, which implements a CLI that uses the APIs provided by `hazard-analyzer` and `pub-api` to create a subcommand for each of them. In this chapter, we provide a comprehensive description of `hazard-analyzer` and `pub-api`, delineating their structure and usage.

[2]https://github.com/SoftengPoliTo/ascot-firmware

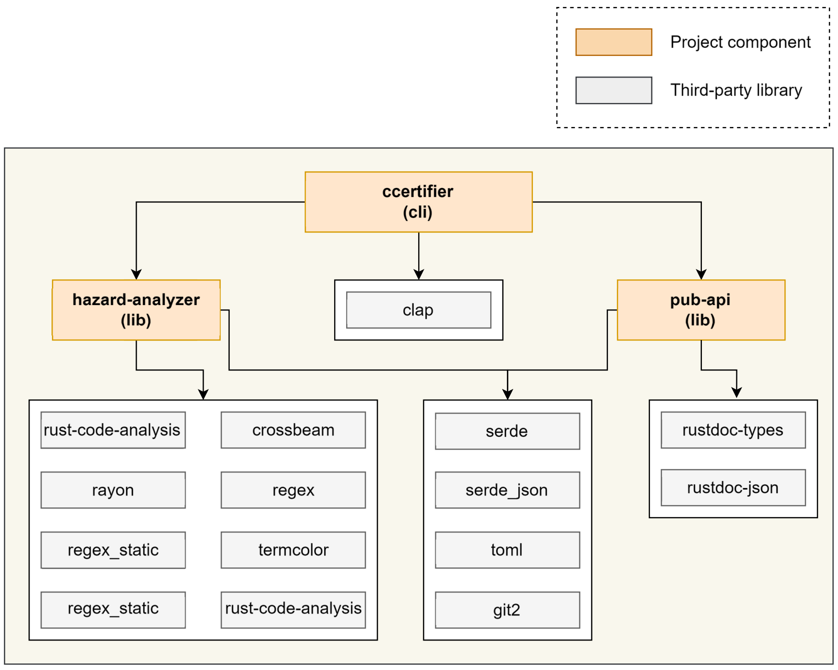


Figure 5.1: code-certifier structure.

5.1 Hazard Analyzer

To model a device within the server, one should create in the `ascot-axum` crate a Rust file for each type of device that can be found in a Smart Home. Currently, we have created only two files: one for light devices and one for refrigerators. However, additional devices can be added as needed, using these two files as starting points and adapting them accordingly.

These files provide the necessary constructs to map, for each action, the low-level interface of a device to the high-level interface provided by `ascot-axum`. Therefore, to interact with a device from the outside, a developer must define a route for each action. In this way, when that route is invoked by an external process, the corresponding action is executed. In the end, each device is characterized by a state and a set of routes that represent its REST interface. When a request is made via an API, the firmware executes the code that the developer associated with it during the firmware implementation, thereby changing the state of a device.

What really distinguishes a device from another is the type of actions it can perform. For example, a light device provides actions such as turning itself on and off, while a refrigerator provides actions such as increasing or decreasing its internal temperature. Each device, depending on its type, can have a certain number of

mandatory actions, which are those actions a device must be able to perform in order to fulfil its main tasks. Indeed, it would not make sense to think of a light that cannot be switched on or off, or of a refrigerator that does not allow its temperature to be adjusted.

Devices currently implemented in the `ascot-axum` crate, as well as those that will be added later, must specify as mandatory only those actions that are absolutely necessary for that device type. This approach enables a `ascot-axum` device to maintain broad applicability, thus allowing its use in defining firmware for various producer-specific variants of the device type. For example, the `ascot-axum` light can be utilized to develop firmware for both an LED light and a bulb light, as long as both provide all the mandatory actions of the `ascot-axum` device.

Each `ascot-axum` device also allows the addition of **optional actions**, which depend on the device's producer, and ensures the possibility of defining routes to control them. For example, a producer might provide options to modify the light's colour or regulate its intensity as features of its light device, whereas lights of other producers may not have these functionalities. As a result, mandatory actions are those that a device must have in order to define its type, while optional actions are those that a producer can add.

When implementing the firmware of a device using `ascot-axum`, a developer must define routes of both mandatory and optional actions. Furthermore, those actions can be associated with zero or more hazards. For example, a developer writing a firmware for a smart washing machine could associate its turn-on action with a `WaterFlooding` hazard. However, developers do not have complete freedom because we set some specific constraints on the hazards that can be associated with a device:

- **Mandatory Hazards**: as we said, each device is characterized by a set of mandatory actions that define it. For each of these actions, we define certain hazards that must be associated with them. For example, a developer defining the `turn light on action` for a light device must necessarily include the `FireHazard` in the list of hazards associated with that action. If a developer fails to add a mandatory hazard and the firmware is executed the same, the execution will abruptly stop, and an error will highlight that the `turn light on` action must have a `FireHazard`. Therefore, this runtime check allows us to assess whether the developer has assigned all the requested mandatory hazards to a device's mandatory actions.

- **Allowed Hazards**: each device is also characterized by a set of allowed hazards that indicates which kind of hazards can be assigned to it. Indeed, it is reasonable to assume that a light device will never have an action associated with a `WaterConsumption` hazard. Therefore, the allowed hazards list restricts the hazards that can be assigned to a device to a set of appropriate ones. During firmware execution, we use this list to check whether the hazards assigned to a device's action are all allowed, and if they are not, we halt the execution and report the error. As a result, hazards assigned to mandatory and optional actions must always be a subset of the device's allowed hazards.

The `hazard-analyzer` crate of `code-certifier` has been developed based on the considerations discussed so far. It takes as input the source code of a firmware which has been implemented using the `ascot-firmware` crates and analyzes its structure to ensure it adheres to the previously described constraints. The analysis consists of several steps and has been divided within the crate through various modules to ensure a high maintainability, reusability, and readability of the code.

5.1.1 Analysis of `ascot-axum` Devices

Before starting the analysis of firmware, we have to first examine the files of the `ascot-axum` devices in order to retrieve for each of them the information about their actions and hazard constraints.

In order to accomplish this, we use the crate `rust-code-analysis` that we introduced in one of the previous chapters 3. This crate generates an Abstract Syntax Tree (AST) from a source code and offers various APIs for traversing its nodes in search of specific constructs. To maximize its use within our project, we made some minor modifications, such as making certain internal functions and modules public for external use and generalizing some of the node search functions.

Leveraging on the functionalities provided by `rust-code-analysis`, we managed to create, for each `ascot-axum` device, a data structure that contains the following information:

- Device name, which represents the type of the device.
- The list of its allowed hazards.
- The list of its mandatory actions.
- For each mandatory action, the list of its mandatory hazards.

At the beginning of the analysis, we download the `ascot-axum` crate into a temporary directory. This ensures that we analyze the most recent version of the crate and can obtain information about any new devices that may have been added in the meantime.

Once we have gathered information about all the devices defined within `ascot-axum`, we can proceed to the next step.

5.1.2 Device Type Retrieval

The `ascot-firmware` crates have been written in Rust. Consequently, the firmware of a device implemented with these crates is also written in Rust language. Therefore, after we ensure that the input firmware path is a directory, we search for all files with the `.rs`[3] extension within that directory. Among these files, since a firmware source code can be split into multiple files, we have to find the one that contains the code snippet used to create the `ascot-axum` device through the crate API. In fact, a developer needs to instantiate an `ascot-axum` device corresponding to the device type for which the firmware is written.

To speed up the analysis of the various firmware files in search of the code snippet, we relied again on the concurrent execution mechanism based on the `producer-consumers-composer` pattern. Therefore, we split the workload among the various consumer threads, which work in parallel to find the code fragment related to the definition of the `ascot-axum` device.

To search for this definition inside a file, we use the `rust-code-analysis` crate. In particular, we exploit the AST generated by the crate to search through its nodes, looking for the root node of the device's definition code snippet. This is achieved by establishing specific conditions on the node to be searched, guaranteeing that the search will yield only one result if the node exists. One of these conditions requires that the node contains the name of one of the possible devices defined within `ascot-axum`. Despite helping us to find the root note, this also allows us to understand the device type for which the firmware has been written.

5.1.3 Device Analysis

Knowing the device type, we can now use the information about `ascot-axum` devices, gathered during the first phase of the workflow, to analyze the defined device starting from its root node.

To do this, we use `rust-code-analysis` to explore all the descendants of the root node and analyse the entire code snippet. However, we must first specify that there are two modes to create an instance of a `ascot-axum` device within a firmware:

1. The first one allows us to define all the mandatory actions of a device by passing them as arguments of a constructor function. This mode is adopted when a device is characterised by a few mandatory actions, thus allowing us to obtain a constructor function with few parameters.

[3]Rust files extension

2. The second mode is used when a device is characterized by a large number of mandatory actions, which would result in a constructor function with too many parameters. This approach consists of defining a method for each mandatory action, which must then be explicitly called one by one in order to define them.

In the first mode, a developer is obliged to define all mandatory actions, since if he does not pass all the necessary arguments to the constructor function, the firmware cannot be compiled. Therefore, in this case, it is possible to check directly at compile time if all mandatory actions are defined.

However, it is not possible to define a check during the compilation phase that verifies whether a developer has called a set of methods. Therefore, in the second case, it is not possible to check at compile time whether a developer has defined all the mandatory actions of a device. Therefore, in this case, a developer could implement firmware without mandatory actions that still compile correctly. However, there is instead a runtime check that detects and signals, by blocking the execution, an attempt to create a `ascot-axum` device without one or more mandatory actions.

Within the firmware analysis process, we consider these two different possibilities of creating a `ascot-axum` device. Specifically, knowing the type of device allows us to determine if it is defined using the first or second mode. Therefore, if it is a device belonging to the second category, we analyse the code snippet, starting from the root node, to ensure that all mandatory actions' methods have been called.

After this initial check, the analysis follows the same process for both device categories, verifying the hazards assigned to their mandatory actions. First, for each action, we ensure that all mandatory hazards have been defined. Next, since it is also possible to assign other hazards in addition to the mandatory ones, we check that the other hazards belong to the list of those allowed for the device.

After completing the mandatory actions analysis, we proceed to analyze the optional actions. For each action, we first extract the name that the developer assigned to it. After that, we check whether the list of hazards associated with the action is a subset of the devices's allowed hazards.

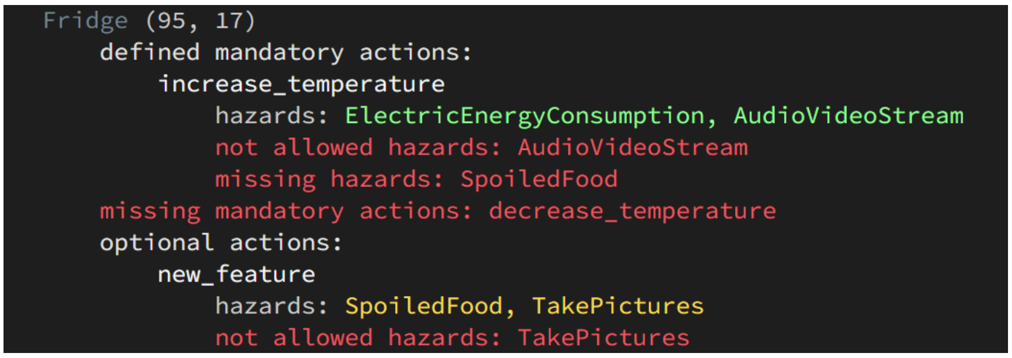
This analysis allows us to obtain a complete overview of all actions and hazards associated with a device. Furthermore, it helps us assess whether the initially defined constraints were adhered to during the definition of hazards.

5.1.4 JSON Manifest Creation

All the information we gather during the analysis is stored within a Rust data structure. Once the analysis is completed, we serialize this data structure, using the crates **serde** and **serde-json**, into a **JSON** file. This file contains a structured representation of the analysis result and is named **firmware manifest**. It includes:

- The device type for which the firmware has been implemented. It is equal to the name of the corresponding `ascot-axum` device.
- The filename that contains the code snippet for creating the instance of the `ascot-axum` device, along with the row and column numbers where the code snippet begins.
- The list of defined mandatory and optional actions. We show each action's name and the list of hazards that a developer has assigned to it. Of these hazards, if any, we show those that are not allowed. Additionally, for the mandatory actions, we list any missing mandatory hazards.
- Any mandatory actions that have not been defined.

5.1.5 Manifest Print



```
Fridge (95, 17)
    defined mandatory actions:
        increase_temperature
            hazards: ElectricEnergyConsumption, AudioVideoStream
            not allowed hazards: AudioVideoStream
            missing hazards: SpoiledFood
    missing mandatory actions: decrease_temperature
    optional actions:
        new_feature
            hazards: SpoiledFood, TakePictures
            not allowed hazards: TakePictures
```

Figure 5.2: hazard-analyzer terminal output format

In addition to the manifest in JSON format, we also give the possibility to print the analysis result on the terminal. For this purpose, as shown in Figure 5.2, we have defined a specific format which, through the use of specific colours, makes it possible to clearly display the various characteristics of a device and to highlight errors relating to actions and hazards:

- **Green**: represents the hazards assigned to the mandatory actions of a device.
- **Yellow**: represents the hazards assigned to the optional actions.

- **Red**: highlights erroneous conditions that cause the device instance to be incorrect. In particular, this colour is used to show the mandatory actions that have not been defined (`missing mandatory actions`), missing mandatory hazards (`missing hazards`), and hazards that are not in the list of allowed ones (`not allowed hazards`).

This format allows for a quick check and identification of potential issues within a firmware. Furthermore, it provides a more user-friendly representation than the `JSON` one, thus making it appropriate even for less experienced users.

5.1.6 Final Remarks

`hazard-analyzer` allows the analysis of firmware source code to extract a comprehensive description of device behaviour, focusing on the actions a firmware can execute. It identifies potential hazards stemming from these actions, forcing a developer to define them explicitly during firmware development. The tool also ensures that device actions adhere to the hazard constraints defined within the `ascot-axum` crate.

The produced outputs allow us to obtain a complete overview of the firmware analysis, and the `JSON` manifest can also be used in a firmware certification process to certify the aspects related to a device's behaviour.

The analysis conducted by `hazard-analyzer` requires access to a firmware source code. While this might be seen as a limitation, it is reasonable to expect that developers who make use of the `ascot-firmware` crates for developing their own firmware would permit to analyze its source code by the `hazard-analyzer`, with the objective of certifying its behaviour.

`hazard-analyzer` has been extensively tested by defining various device types and firmware examples to check whether the tool can detect every erroneous situation previously described. These tests have been implemented using the `insta` crate, which allowed us to exploit the advantages of snapshots in order to streamline the testing process.

We have created firmware which represents different device types as tests. Among them, we have intentionally omitted mandatory actions and firmware where actions lack mandatory hazards or include hazards that are not allowed. Finally, we also tested two firmware examples [4] that we implemented with `ascot-firmware`, which define a concrete application of the concepts we presented before. These tests

[4] https://github.com/SoftengPoliTo/ascot-firmware/tree/master/ascot-axum/examples

allowed us to comprehensively verify the `hazard-analyzer` operations and ensure that the firmware analysis is conducted as intended.

5.2 Public API

The second crate we implemented is `pub-api`. This crate is designed with the objective of extracting all public APIs from the `ascot-library` and `ascot-axum` crates. The information retrieved by `pub-api` can, for example, be used to identify all `ascot-firmware` APIs utilized within a firmware, providing additional information to better analyze and certify a device's behaviour.

To extract the public APIs, we leveraged on `rustdoc` [5], a documentation generation tool integrated into the Rust toolchain. This tool allows to generate a complete and detailed documentation for an entire Rust project.

In particular, we have used an experimental feature of this tool that permits the representation of project documentation as a `JSON` [6] file, instead of the `HTML` that `rustdoc` produces by default.

The `JSON` file represents the AST of a Rust project, and starting from the root node, it is possible to explore all the public constructs contained in the tree. To exploit `rustdoc` and its `JSON` represenation within `pub-api`, we have adopted two third-party crates:

- **`rustdoc-json`**: launches the `rustdoc` command on a project in order to produce a `JSON` documentation.
- **`rustdoc-types`**: defines some data structures and types that can be used to facilitate the parsing of the `JSON` file created with the first crate.

With these two crates, we are able to extract all the public APIs contained in `ascot-library` and `ascot-axum`. In particular, we currently extract the following Rust constructs:

- Structs [7]
- Enum [8]

[5]https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html

[6]https://rust-lang.github.io/rfcs/2963-rustdoc-json.html

[7]https://doc.rust-lang.org/book/ch05-00-structs.html

[8]https://doc.rust-lang.org/book/ch06-00-enums.html

- Traits [9]
- Functions [10]
- Macros [11]

As output, the crate produces a `JSON` manifest that contains a list of all these constructs. For each construct, it specifies its name, the file within which it is contained in addition to the eventual functions associated with it.

This tool has been extensively tested to verify that it is indeed capable of extracting all the constructs previously listed. As for `hazard-analyzer`, we used the `insta` crate to define snapshots, which allows us to define different scenarios and observe how the crate analyzes each of them. In particular, we create snapshots for each specific extracted construct and also test more complex use cases, including placing constructs within internal sub-modules or using aliases [12].

Using `rustdoc-json` and `rustdoc-types` allows for a straightforward expansion of the capabilities of this crate since the analysis relies only on the information contained in the `JSON` documentation. This ensures that the analysis remains independent of the specific project under analysis, allowing the extraction of public APIs from any Rust project.

5.3 `ccertifier`

As we mentioned at the beginning of the chapter, within `code-certifier` we have also defined a crate named **`ccertifier`** that provides an interface for using `hazard-analyzer` and `pub-api`. In particular, this crate implements a CLI by defining two subcommands: `hazard-analyzer` and `pub-api` for the respective crates.

The arguments provided by the `hazard-analyzer` subcommand are:

- **Firmware Path**: is a required argument that represents the path to the firmware that has to be analyzed.
- **Devices Path**: is an optional argument that can be used to specify the path to a local directory containing the files of the `ascot-axum` devices, in

[9]https://doc.rust-lang.org/book/ch10-02-traits.html

[10]https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

[11]https://doc.rust-lang.org/book/ch19-06-macros.html

[12]An alias is a way to create an alternative name for existing types or modules

order to use them instead of the devices defined within the `ascot-axum` crate downloaded by the `hazard-analyzer`.

- **Manifest Path**: is a mandatory argument that specifies the path in which to place the `JSON` firmware manifest.
- **Quiet**: If set, this argument causes the result of the analysis to no longer be printed on the terminal.

On the other hand, `pub-api` provides the following arguments:

- **Library Path**: is an optional argument that can be used to specify the path to a local version of the `ascot-library` crate, in order to use it instead of the version download by `pub-api`.
- **Axum Path**: is an optional argument that can be used to perform the same operation of the previous one but for the `ascot-axum` crate.
- **Manifest Path**: is a mandatory argument that can be used to specify the path in which to place the output `JSON` file that contains the extracted APIs.

Chapter 6

Performance Analysis

In this chapter, we conduct a performance analysis of some of the tools introduced in the previous chapters. The purpose of this analysis consists of evaluating their behaviour in increasingly computationally demanding scenarios and eventually identify potential improvements. In particular, we perform two types of performance analysis: the first one evaluates the tools' execution times, while the second one examines their memory usage.

We have decided to analyse: `weighted-code-coverage`, `complex-code-spotter`, as well as the `hazard-analyzer`, a crate integrated into `code-certifier` as one of its crates.

For the first two static analysis tools, in addition to adding new features, we also made significant changes to the implementation of their concurrent execution mechanism. Therefore, it is important to evaluate how these changes have impacted the performance of the tools, and determine whether they have improved or possibly worsened it.

As we previously introduced, `weighted-code-coverage` computes some code metrics from a Rust project source code, while `complex-code-spotter` evaluates its complexity. Therefore, both tools need to analyse every Rust file in a project, and the effort required to perform this analysis depends on the size of the project, specifically on the number of source files that compose it. In fact, it is reasonable to assume that as the number of files increases, the tool will be under greater stress since it will have to analyse and process a larger quantity of data. Therefore, this analysis aims to assess the impact of project size on the tools' execution time and memory usage, and verify how they vary according to this parameter. For this purpose, we choose three different Rust projects with increasing sizes, which will be used as input projects for analysing `weighted-code-coverage` and `complex-code-spotter`:

- **`seahorse`**[1]: is a minimal framework for creating CLIs in Rust. It is easy to use, has no external dependencies, and relies entirely on the constructs of the Rust standard library. `seahorse` comprises only 12 Rust files, so we used it as the small project example for our tools' analysis.
- **serde**[2]: is a library used for serializing Rust data structures known such as `JSON`, `XML`, `BSON`, or deserializing these formats into Rust data structures. It is one of the most widely used libraries within the Rust ecosystem and consists of 164 Rust files, making it a good choice as an example of a medium-sized project.
- **rust-analyzer**[3]: represents the choice for the large project example, consisting of 1270 Rust files. This tool provides a front-end for the Rust compiler that can be integrated into various text editors used by developers for writing code. It improves developers' productivity by providing complete support for code writing, code navigation, debugging, and error checking.

As the two previous static analysis tools, `hazard-analyzer` also analyses all the Rust files of a project. However, we cannot consider any Rust project, but only a firmware implemented with the `ascot-firmware` crates. For this reason, to analyse `hazard-analyzer`, we use one of the `ascot-firmware` examples, specifically the one that implements a firmware for a light device [4].

6.0.1 Analysis Tools

To conduct this execution time and memory usage analysis, have chosen the following tools:

- Hyperfine[5]: is a command line tool that allows precise measurement of a program execution time through benchmarking, making it ideal for a performance evaluation of our tools. Hyperfine provides a statistical average of execution time by repeatedly running the program under analysis, enabling the identification of time variations and fluctuations. Its ability to perform warm-up runs and prepare the execution environment for optimal conditions

[1]https://github.com/ksk001100/seahorse

[2]https://github.com/serde-rs/serde

[3]https://github.com/rust-lang/rust-analyzer

[4]https://github.com/SoftengPoliTo/ascot-firmware/tree/master/ascot-axum/examples/light

[5]https://github.com/sharkdp/hyperfine

ensures stable and accurate results, reducing the impact of random variations and temporary system conditions.

- **Heaptrack**[6]: is a memory profiling tool that tracks memory allocation during execution. This tool is essential for analyzing how our software manages memory and for identifying usage peaks, temporary allocations, and potential memory leaks. By using Heaptrack, we obtain a detailed view of the tools' memory usage, which we can use as information for optimizing their memory management, and thus improving their overall efficiency.

Heaptrack computes various metrics from the analysis it performs. Among all of them, we have decided to consider the following ones:

- **Allocations**: an allocation occurs when an operating system dynamically assigns a certain amount of memory to a program based on a certain request. This memory is used by the program to store data needed at runtime for its execution. Heaptrack provides the total number of allocation requests made by a program, thus allowing the detection of any memory usage anomaly.
- **Temporary allocations**: refer to memory allocations that persist for a short period, such as temporary data structures quickly released as soon as they are no longer needed. Knowing the total number of temporary allocations provides insights into memory management efficiency, revealing opportunities for further optimization.
- **Permanent allocations**: in contrast to temporary allocations, permanent allocations refer to allocations that persist for a longer time during the program's execution and thus have a greater impact on its memory usage. Monitoring this parameter is essential, as a high number of permanent allocations may overload the memory and decrease overall program performance.
- **Peak heap memory consumption**: the heap is the program's memory area used to dynamically allocate the memory requested by the program during its execution. This value helps identify the program's maximum memory requirements and detect potential memory leaks, which refers to allocations that are not correctly released.
- **Peak RSS (Resident Set Size)**: RSS represents the amount of physical memory (RAM) that an operating system reserves for a process at a given time. Peak RSS indicates the maximum RAM required by a program during its entire execution. This information contributes to defining a program's memory requirements and can help identify inefficient memory usage.

[6]https://github.com/KDE/heaptrack

6.1 weighted-code-coverage

We start by analysing `weighted-code-coverage` It is the first tool on which we have applied the `producer-consumer-composer` pattern, which was subsequently adopted for the other tools we implemented. We therefore use the analysis of `weighted-code-coverage` to understand whether this pattern actually brings advantages. At the same time, we want to evaluate the impact of new features and further modifications on the tool's performance.

In particular, we will examine how the tool performed in terms of execution time and memory usage before the modifications and then compare the obtained results to those achieved after the changes.

6.1.1 Execution Time

Before

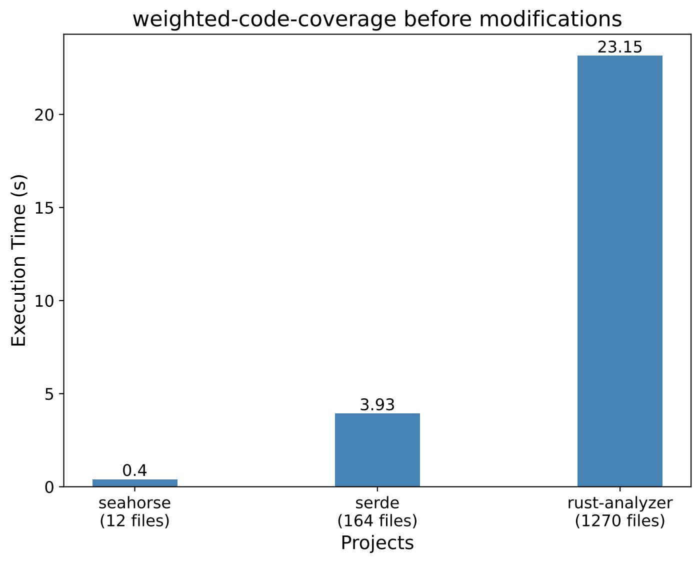


Figure 6.1: weighted-code-coverage execution times before modifications

As depicted in Figure 6.1, the execution time of the original version of `weighted-code-coverage` tends to increase as the number of project files grows. Specifically, we observe that the execution time is nearly ten times when transitioning from `seahorse`, which has 12 files, to `serde`, which has 164 files. Moreover, in the case of `rust-analyzer`, with the number of files increasing by approximately 105 times that of `seahorse` and around eight times that of `serde`, the execution time rises to 23.15 seconds, which is nearly fifty-six times that of `seahorse` and about six times that of `serde`.

After

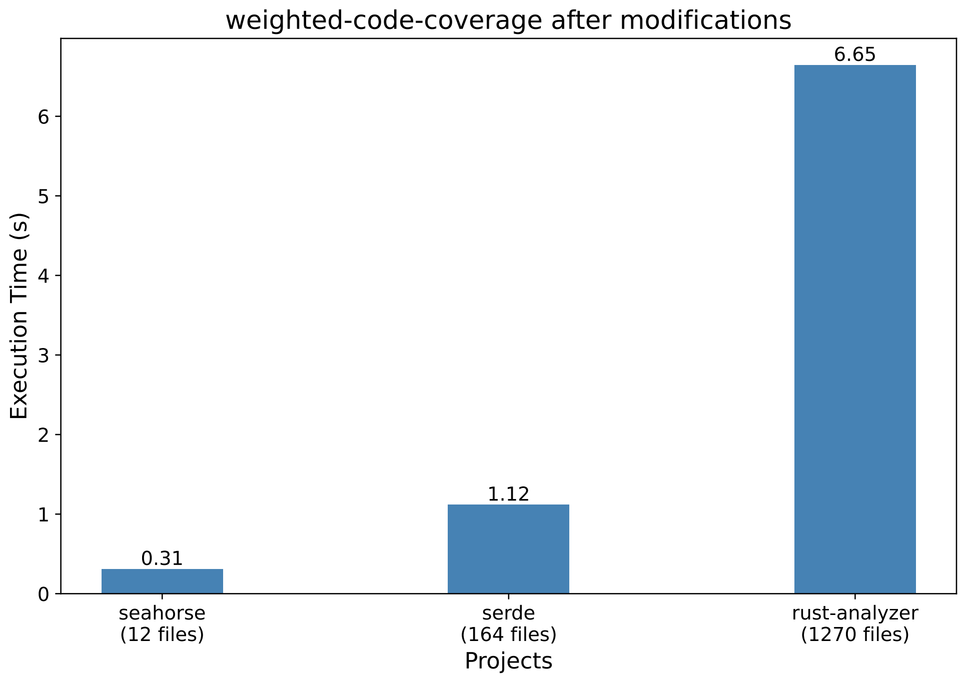


Figure 6.2: weighted-code-coverage execution times after modifications

In the chart shown in Figure 6.2, we observe that even after our modifications, the execution times of `weighted-code-coverage` increase as the number of files grows. Starting with an execution time of 0.31 seconds for `seahorse`, the time rises to 1.12 seconds for `serde`, nearly a four times increase. For `rust-analyzer`, the execution time is 6.65s, which is about twenty-two times longer than `seahorse` and nearly six times longer than `serde`.

Comparison

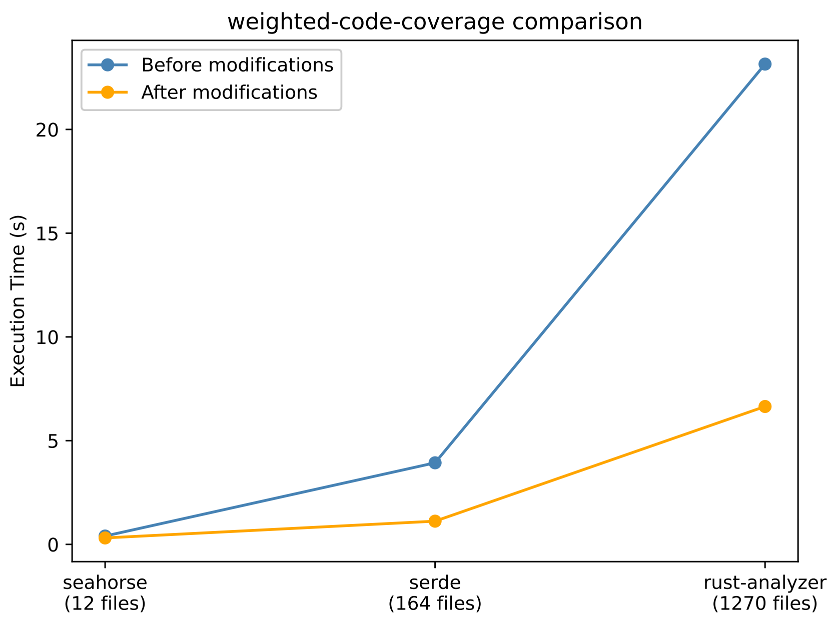


Figure 6.3: weighted-code-coverage execution times comparison

These results confirm our initial assumptions and demonstrate that as the size of a project increases, the tool must process more files, leading to longer execution times. This trend is true for both versions of the tool, but there are some differences.

As shown in Figure 6.3, the execution times of the modified version of `weighted-code-coverage` are consistently shorter than those of the unmodified version. This difference becomes more and more accentuated as the projects become larger. In fact, for `seahorse`, the execution time of the new version is 0.31s compared to 0.4s for the old version. However, if we consider `rust-analyzer`, we observe that the execution time of the new version is approximately four times shorter and has a slower growth rate as the number of files increases.

These observations allow us to conclude that adding features to `weighted-code-coverage` has not negatively impacted its execution time. Furthermore, replacing the old concurrent execution implementation with the *producer-consumer-composer* pattern has made the tool faster, decreasing the overall execution times and growth rate.

6.1.2 Memory Usage

Before

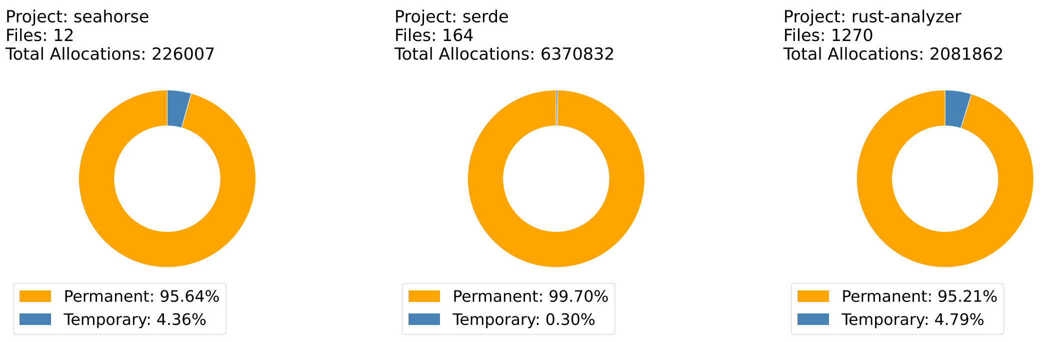


Figure 6.4: weighted-code-coverage allocations before modifications

Analysing the charts in Figure 6.4, we can observe that the initial version of `weighted-code-coverage` records a total of 226,007 allocations for `seahorse`, 6,370,832 for `serde` and 2,081,862 for `rust-analyzer`.

The first thing we notice is that the number of allocations does not seem to depend on the size of the project being analysed. In fact, `serde` has significantly fewer files than `rust-analyzer`, and despite that, it has approximately three times more allocations.

The proportion of temporary allocations also seems independent of the number of files. `rust-analyzer` has the highest percentage of temporary allocations at 4.79%, closely followed by `seahorse` at 4.36%, while `serde` has a much lower value of just 0.3%.

This data suggests very poor memory management for what concerns the `serde` analysis, given the very high number of allocations and the very low percentage of temporary allocations. Instead, the more effective utilization of temporary allocations is during the analysis of `rust-analyzer`.

After

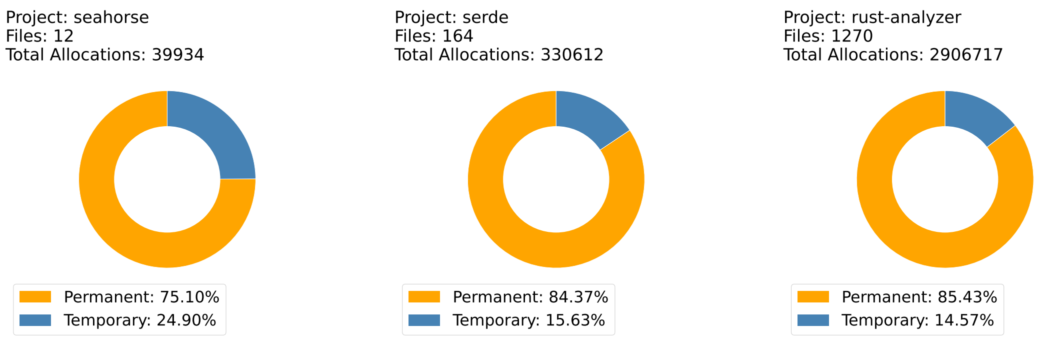


Figure 6.5: weighted-code-coverage allocations after modifications

Figure 6.5 shows the memory allocation values for the new version of the tool. The charts reveal that the number of allocations increases with the number of files in the project. Specifically, the number of allocations starts at 39,934 for `seahorse`, rises to 330,612 for `serde`, and reaches 2,906,717 for `rust-analyzer`.

The percentages of temporary allocations follow a similar trend: for `seahorse`, the value is 24.9%, for `serde` it is 15.63%, and for `rust-analyzer` it is 14.57%. This data indicates that memory utilization intensifies as the number of files to be analysed increases, leading to more allocations. Additionally, they suggest that as projects grow in size, managing the analysis with temporary allocations becomes increasingly difficult. This results in a greater reliance on permanent allocations, which have a larger impact on overall memory usage.

Comparison

Project	Files	Memory Peak	Peak RSS	Allocations	Temporary
seahorse	12	2.3 MB	14.5 MB	226007	4.36 %
serde	164	55.7 MB	69.4 MB	6370832	0.3 %
rust-analyzer	1270	183.5 MB	200.8 MB	2081862	4.79 %

Table 6.1: weighted-code-coverage memory usage before modifications.

Project	Files	Memory Peak	Peak RSS	Allocations	Temporary
seahorse	12	2.5 MB	1.7 MB	39934	24.9 %
serde	164	10.5 MB	29.1 MB	330612	15.63 %
rust-analyzer	1270	19.9 MB	46.9 MB	2906717	14.57 %

Table 6.2: weighted-code-coverage memory usage after modifications

Tables 6.1 6.2 provide a comprehensive overview of the memory usage of the two versions of the tool. Using this data, we can show how these changes have impacted the tool's memory management.

Starting with the memory peak parameter, which corresponds to the peak heap memory consumption, we see a clear decrease for `serde` and `rust-analyzer` projects, from 55.7MB to 10.5MB and from 183.5MB to 19.9MB. For `seahorse`, on the other hand, the value has not decreased but actually increased from 2.3MB to 2.6MB. The improvements we made to boost the tool's performance are being balanced out in small projects by the extra memory used for the new features.

Regarding peak RSS values, there is a notable decrease across all projects. For `seahorse`, it dropped from 14.5MB to 1.7MB, for `serde` from 69.4MB to 29.1MB, and for `rust-analyzer` from 200.8MB to 46.9MB.

Looking at the number of allocations, `seahorse` and `serde` saw significant decreases, from 266,007 to 39,934 and from 6,370,832 to 330,612. However, `rust-analyzer` saw an increase from 2,081,862 to 2,906,717.

The percentage of temporary allocations is much higher in the update version of the tool. It increased from 4.36% to 24.9% for `seahorse`, from 0.3% to 15.63% for `serde`, and from 4.79% to 14.57% for `rust-analyzer`. Notably, there is a large difference between the two percentages of `serde`, and in the updated version, the values for `serde` and `rust-analyzer` are very similar despite the latter being a much larger project.

From this comparison, we can conclude that our changes have led to a general improvement in memory management, with memory usage following a more predictable trend as project size increases. The tool's memory peaks have decreased considerably, and the use of different programming practices has optimized the program's allocations by increasing the usage of temporary allocations instead of permanent ones.

6.2 complex-code-spotter

For `complex-code-spotter`, the changes did not involve adding new features. Instead, we have modified the concurrent execution mechanism by replacing the old implementation with the `producer-consumer-composer` pattern and simplified most parts of the code. Since we have not added new features which could increase execution times or memory usage, we expect our modifications to improve the tool's performance or, at the very least, not degrade it.

6.2.1 Execution Time

Before

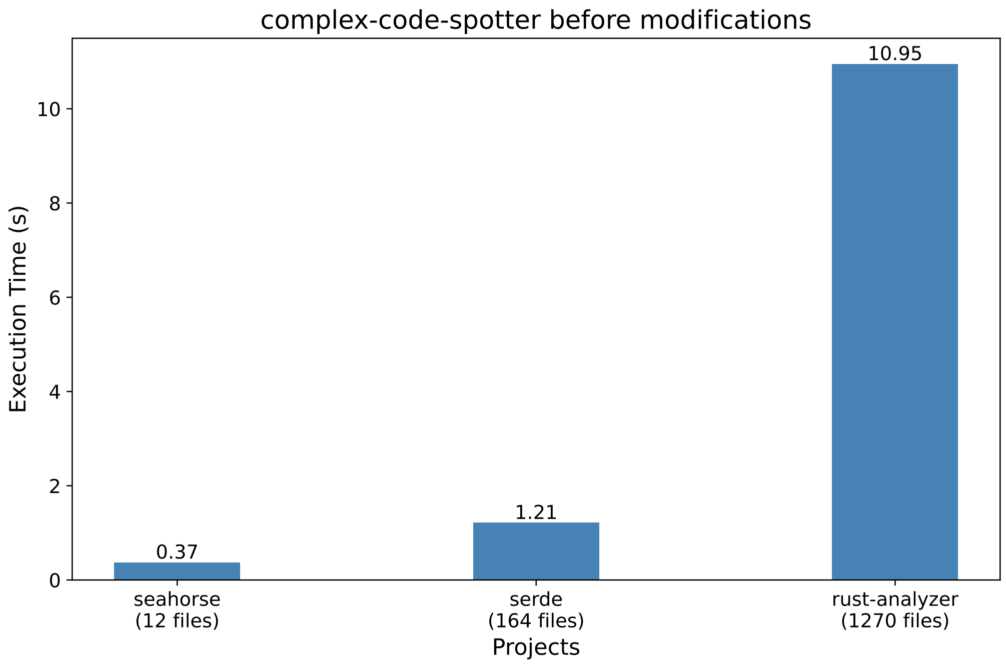


Figure 6.6: complex-code-spotter execution times before modifications

In the version prior to our modifications, as shown in Figure 6.6, the execution time increases significantly with the size of the project. When moving from `seahorse` to `serde`, the execution time rises from 0.37s to 1.21s, roughly tripling. For `rust-analyzer`, the execution time is 10.95s, which is approximately thirty times longer than `seahorse` and about nine times longer than `serde`.

After

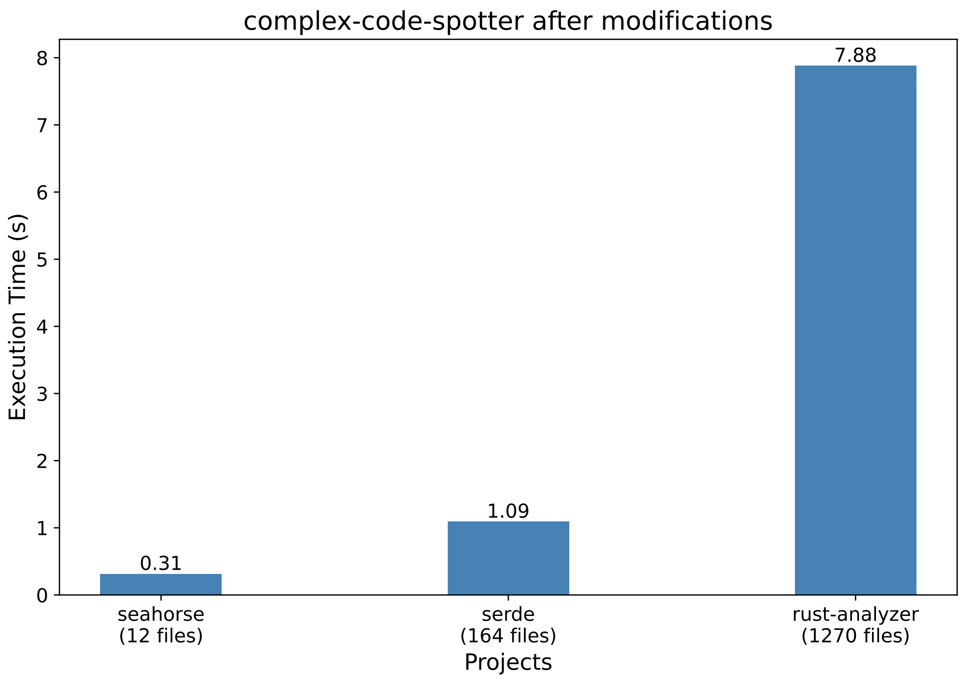


Figure 6.7: complex-code-spotter execution times after modifications

As shown in Figure 6.7, in the modified version of `complex-code-spotter`, the execution time for `seahorse` is 0.31s. For `serde`, the value increases to 1.09s, roughly four times longer. For `rust-analyzer`, the tool takes 7.88s, which is about twenty-five times longer than `seahorse` and seven times longer than `serde`.

Comparison

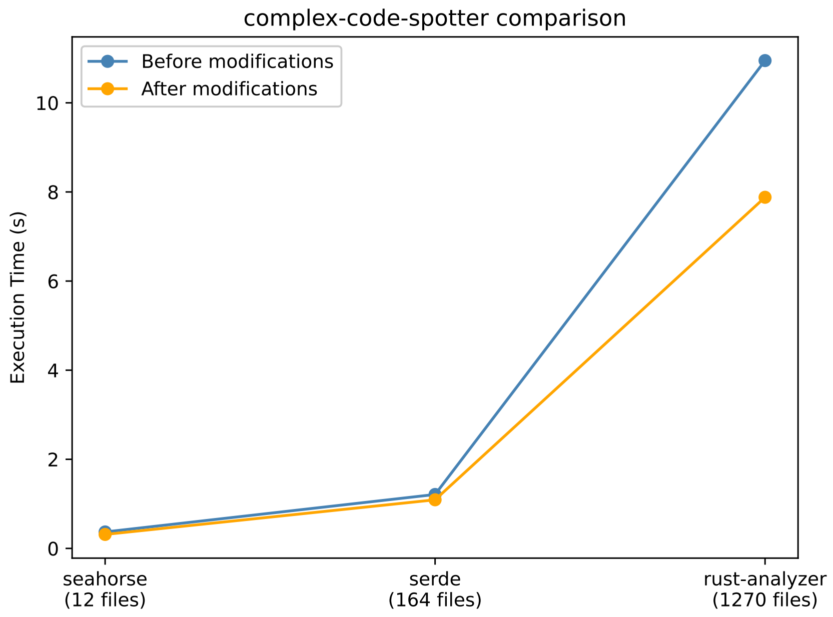


Figure 6.8: complex-code-spotter execution times comparison

We note that, as expected, execution times increase as the number of project files increases, regardless of whether the tool has been used with or without modifications.

However, as seen in Figure 6.8, the difference between the execution times of the two versions of the tool is not very pronounced for small to medium-sized projects, with similar execution times for `seahorse` and `serde`. For `seahorse`, the time decreases from 0.37s to 0.31s, and for `serde`, it decreases from 1.21s to 1.09s.

These observations lead us to conclude that the impact of our changes on `complex-code-spotter` becomes more significant as the project size increases. The old version has comparable execution times to the updated one for projects with few files. For `rust-analyzer`, the changes reduce the execution time from 10.95s to 7.88s, a decrease of over 3s.

These considerations allowed us to assess that the impact of the changes made to `complex-code-spotter` tends to become more and more significant as the project size increases. For `rust-analyzer`, our changes reduce the execution time from 10.95s to 7.88s, a decrease of over 3s. Instead, the old version of the tool has execution times comparable to the updated version for projects with fewer files.

6.2.2 Memory Usage

Before

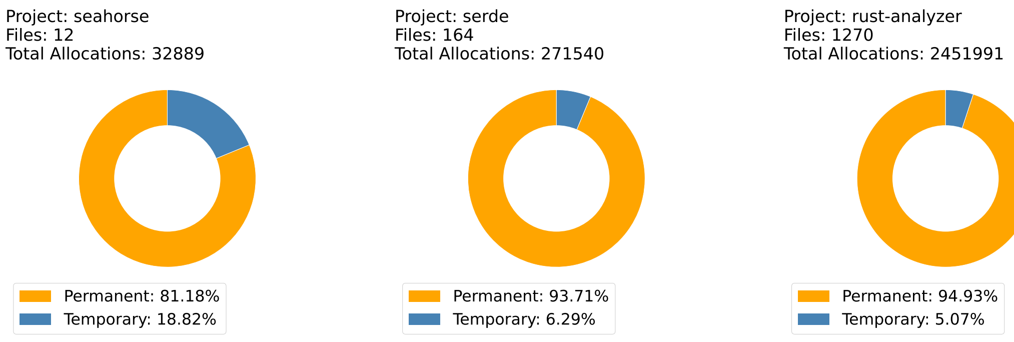


Figure 6.9: complex-code-spotter allocations before modifications

Looking at the charts depicted in Figures 6.9, we can observe that as the number of files in the project increases, the total number of allocations also rises. Moving from `seahorse` to `serde`, the number of allocations increases from 32,889 to 271,540, and for `rust-analyzer`, it reaches 2,451,991.

However, the trend for the percentage values of temporary allocations is less consistent. The highest value is for `seahorse` at 18.82%, while it decreases to 6.29% for `serde`, and then increases again to 14.57% for `rust-analyzer`.

After

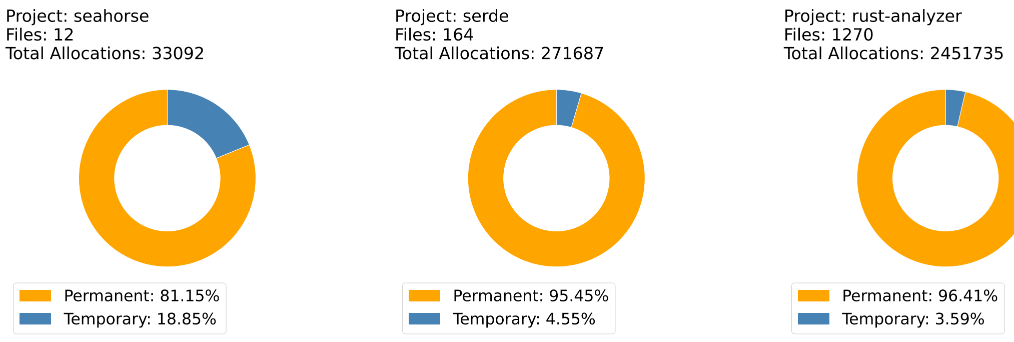


Figure 6.10: complex-code-spotter allocations after modifications

The changes made to the tool are reflected in the allocation charts depicted in Figure 6.10. Here, we can observe that the total number of allocations for the three projects are 33,092 for `seahorse`, 271,687 for `serde`, and 2,451,735 for `rust-analyzer`.

The percentage of temporary allocations appears to correlate with the number of files in the project under analysis. This value shows a gradual decrease, starting at 18.85% for `seahorse`, decreasing to 4.55% for `serde`, and further dropping to 3.59% for `rust-analyzer`.

Comparison

Project	Files	Memory Peak	Peak RSS	Allocations	Temporary
seahorse	12	7.6 MB	16.5 MB	32889	18.82 %
serde	164	87.4 MB	83.4 MB	271540	6.29 %
rust-analyzer	1270	220.6 MB	260.2 MB	2451991	5.07 %

Table 6.3: complex-code-spotter memory usage before modifications.

Project	Files	Memory Peak	Peak RSS	Allocations	Temporary
seahorse	12	7.6 MB	15.3 MB	33092	18.85 %
serde	164	94.4 MB	74.2 MB	271687	4.55 %
rust-analyzer	1270	233.7 MB	242.8 MB	2451735	3.59 %

Table 6.4: complex-code-spotter memory usage after modifications.

Comparing the memory management of the two versions of the tool by means of Tables 6.3, 6.4, we can observe that the implemented changes resulted in increased peak memory values for `serde` and `rust-analyzer`, from 87.4MB to 94.4MB and from 220.6MB to 233.7MB, respectively. However, the peak memory value for `seahorse` remained unchanged at 7.6MB. This suggests a worse allocation management within the tool, indicating an increased requirement for memory allocations compared to the previous version.

This consideration also seems to be confirmed by comparing the total number of allocations. In fact, while the value decreased for `rust-analyzer`, from 2451991 to 2451735, there was an increase for the other two projects. For `seahorse` and `serde`, it increased from 32,889 allocations to 33,092, and from 271,540 to 271,687.

The percentage of temporary allocations also indicates a less efficient memory usage. Except for `seahorse`, where the value remained almost unchanged from 18.82% to 18.85%, there was a decrease for this metric as project size increased. Specifically, for `serde`, it decreased from 6.29% to 4.55%, and for `rust-analyzer`, it dropped from 5.07% to 3.59%.

This decline in allocation management efficiency could be attributed to the implementation of concurrent execution code. It improved execution times but, at the same time, increased the total number of allocations and reduced the percentage of temporary ones.

Conversely, there was a minor improvement in the peak RSS parameter. In each of the projects, this value decreased in the new version of the tool: from 16.5MB to 15.3MB for `seahorse`, from 83.4MB to 74.2MB for `serde`, and from 260.2MB to 242.8MB for `rust-analyzer`. This suggests that despite an increase in allocation requests, the tool now requires less overall RAM memory.

6.3 hazard-analyzer

As we have already mentioned in Chapter 55, to evaluate the performance of `hazard-analyzer`, we use a small light firmware implemented with the `ascot-firmware` crate. This is because, differently from the analysis of the two static analysis tools, for `hazard-analyzer`, we cannot have a large variety of firmware projects of different sizes. Therefore, we will provide an overview of performance based on analysing a single firmware, while still attempting to comprehensively evaluate the tool's execution time and memory management aspects.

6.3.1 Execution Time

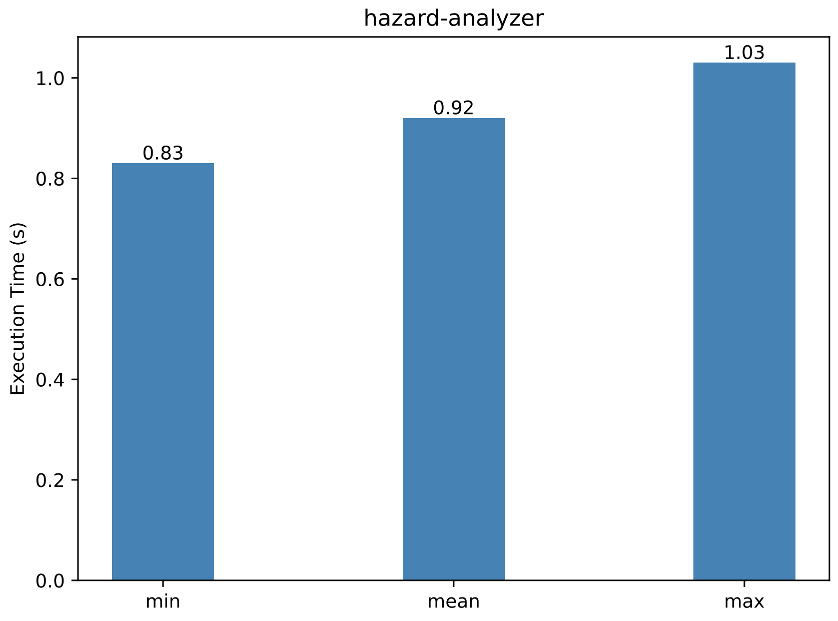


Figure 6.11: hazard-analyzer execution times

Figure 6.11 shows the minimum, mean, and maximum execution times for `hazard-analyzer`. These values are derived from the analysis performed with hyperfine, which runs the program multiple times, extracting these specific execution times. The average value is 0.92 seconds, while the minimum and maximum times are 0.82 seconds and 1.03 seconds, respectively. In particular, we can observe that the average execution time is under 1s, with a value of 0.92, while the minimum and maximum values are 0.82s and 1.03s.

These data show that the execution time of the tool is fairly consistent between the various executions and is comparable with the execution times of `weighted-code-coverage` and `complex-code-spotter` in the case of a medium-sized project.

6.3.2 Memory Usage

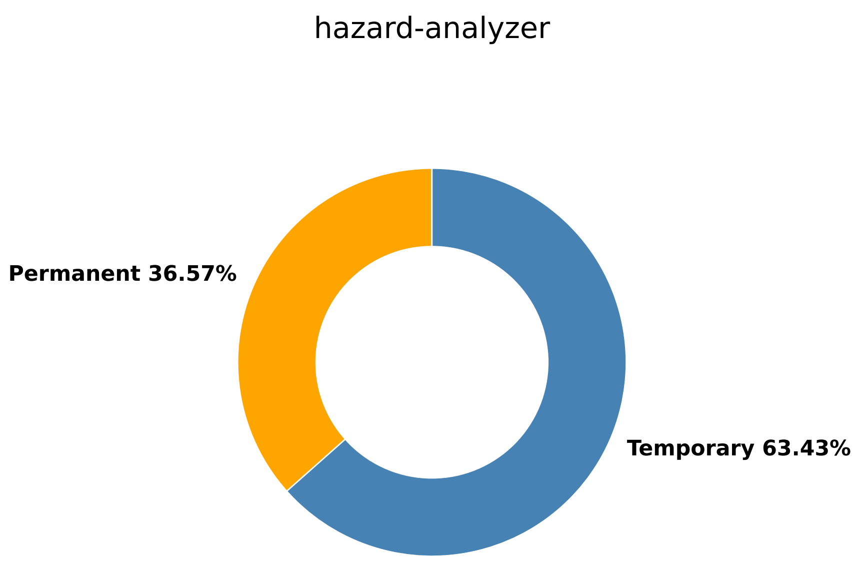


Figure 6.12: hazard-analyzer memory allocations

Memory Peak	Peak RSS	Allocations	Temporary
3.1 MB	25.8 MB	161772	63.43 %

Table 6.5: hazard-analyzer memory usage

Figure 6.12 and Table 6.5 illustrate that the tool accumulates 161,772 allocations, with 63.43% of them being temporary. This indicates that while the number of allocations is relatively high, a significant majority of them are temporary, facilitating a quick memory release and resulting in a small memory peak of only 3.1MB.

In contrast, the RSS peak reaches 25.8MB. This relatively high value, considering that we are using a small firmware for the analysis, is probably caused by the intensive exploration and traversal of AST nodes using `rust-code-analysis`. This process often involves deep tree traversal to locate nodes with specific properties, which can intricately increase both the number of allocations and memory usage.

Therefore, it is important to further optimize the tool to reduce the RSS peak and potentially decrease the total number of allocations, while maintaining a high percentage of temporary allocations for a more efficient memory usage.

6.4 Final Remarks

This performance analysis allowed us to assess that for `weighted-code-coverage` and complex-code-spotter, there is a close dependence between execution time and the size of the project under analysis. In particular, execution times increase as the number of files of a project grows. In addition, the data obtained allowed us to evaluate the impact of our changes, noting that they improved execution times for both tools regardless of the size of the input project.

For what concerns memory management of these two tools, we observed, with some exceptions, that it is generally influenced by the size of the project. In fact, memory peak, peak RSS, and total allocation values tend to increase as the project size increases, while the percentage of temporary allocations tends to decrease. The changes made to `weighted-code-coverage` resulted in an overall improvement in memory utilization. For `complex-code-spotter`, however, there was a general worsening in all parameters except peak RSS, suggesting that the tool's memory management could be further improved.

The performance analysis of `hazard-analyzer` enabled us to evaluate its execution time and memory usage by passing a small firmware as input. This analysis was less exhaustive compared to that of `weighted-code-coverage` and `complex-code-spotter`, due to the lack of projects of different sizes. This indicates a need for future re-evaluations with a broader range of firmware with the objective of better assessing the tool's behaviour under more computationally demanding conditions. However, the data obtained was generally encouraging, both in terms of execution time and memory management. The latter, in particular, could be further improved by reducing total RAM usage and optimizing the tool's allocations.

Chapter 7

Conclusions

In this thesis, we have observed how the rapid evolution of the Internet of Things (IoT) ecosystem has led to various unresolved challenges related to security, reliability, and standardization.

One particular problem that stands out is the absence of a well-defined certification process for analyzing the firmware of IoT devices. As a result, many devices, released by different manufacturers, may contain major vulnerabilities that significantly compromise their security.

These problems become even more pronounced if we think of a Smart Home system, where, for example, some devices may manage a home's surveillance or fire prevention system.

Thus, for the kind of devices that can collect sensitive data and manage critical functions within a home, vulnerabilities can result in situations that seriously threaten the privacy and physical safety of its inhabitants.

It is therefore essential to define certification processes that, starting from the firmware of a Smart Home device, can draw a comprehensive assessment of its security and associated risks.

In this context, we have initially introduced two static analysis tools, `weighted-code-coverage` and `complex-code-spotter`, which can be used to analyze and evaluate the source code of a firmware in terms of software quality.

Next, we have introduced `hazard-generator` and `code-certifier`. The former takes as input a hazard ontology and generates an API that can be used to define the hazards that may arise as a consequence of the actions performed by a device. The generated API is then integrated inside a crate used to develop firmware for Smart Home devices. This library aims to enhance the understanding and classification of devices' behaviour while reducing the effort required by a firmware developer. `code-certifier`, instead, analyses a firmware source code with the objective of obtaining a description of all its actions and associated hazards. Furthermore, it generates a firmware manifest which can be integrated into a certification process.

7.1 Future Developments

Both the tools developed for static firmware analysis and those for certifying device behaviour can be improved and extended with new features. The continuously evolving Smart Home ecosystem presents numerous opportunities to enhance these tools, adapting them to ongoing and emerging challenges while increasing their effectiveness. The following sections outline some specific examples of where further advancements can be made:

- **`weighted-code-coverage`**: in terms of possible future improvements, we might think of adding a new feature that could be used to exclude specific project functions from the analysis performed by this tool. This would be beneficial when complex or poorly covered functions could affect the metrics values of an entire project. Another idea could be that of exploring the integration of the necessary processes focused on producing a `grcov` file directly within `weighted-code-coverage`, in order to further streamline the tool's workflow. Furthermore, ongoing efforts could focus on refining the metrics implemented by the tool, particularly *CRAP* and *Skunk*, with the objective of increasing their precision and relevance during code quality evaluation.
- **`hazard-generator`**: we might focus on refining the risk score definition and providing a rigorous assessment methodology in order to calibrate its values. This would allow for a more precise description of the hazards assigned to an action, leading to a more comprehensive understanding of a device's behaviour.
- **`code-certifier`**:
  - **`pub-api`**: the public API extraction performed by this create could be extended by adding new features, such as extracting additional types of constructs or gathering more information about the ones which have already been retrieved. For example, one could implement the possibility to extract the parameters of a function.
  - **`hazard-analyzer`**: we foresee the possibility of defining firmware for more complex devices in the future, which might be formed by a composition of several devices. For example, we can think of a smart mirror that, in

addition to humidity sensors and a touch user interface, also provides some LED lights. In that case, the device could be implemented as a combination of simple devices, each one with its corresponding `ascot-axum`5 device implementation and actions. For such scenario, we could extend the firmware analysis performed by our crate in order to provide a manifest that includes all necessary information about every device that is part of the composition. Additionally, supplementing the source code analysis with a firmware binary analysis would make the certification process even more flexible and comprehensive. This would also allow a firmware behaviour analysis even when a source code is unavailable. Finally, further improvements can be made to streamline firmware analysis by eventually adding new features. For example, we could try to modify some constraint checks in order to perform them at compile time, such as those related to the definition of all mandatory hazards of an action.

Bibliography

[1] Detlef Schoder. «Introduction to the Internet of Things». In: *Internet of things A to Z: technologies and applications* (2018), pp. 1–50 (cit. on p. 4).

[2] Tajkia Nuri Ananna and Munshi Saifuzzaman. *Introduction to IoT*. 2024. arXiv: `2312.06689 [cs.CR]` (cit. on p. 5).

[3] Manuel Silverio-Fernández, Suresh Renukappa, and Subashini Suresh. «What is a smart device? - a conceptualisation within the paradigm of the internet of things». In: *Visualization in Engineering* 6.1 (May 2018), p. 3. ISSN: 2213-7459. DOI: `10.1186/s40327-018-0063-8`. URL: `https://doi.org/10.1186/s40327-018-0063-8` (cit. on p. 5).

[4] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. «A brief history of the internet». In: *SIGCOMM Comput. Commun. Rev.* 39.5 (Oct. 2009), pp. 22–31. ISSN: 0146-4833. DOI: `10.1145/1629607.1629613`. URL: `https://doi.org/10.1145/1629607.1629613` (cit. on p. 6).

[5] Kevin Ashton et al. «That 'internet of things' thing». In: *RFID journal* 22.7 (2009), pp. 97–114 (cit. on p. 6).

[6] Arindam Giri, Subrata Dutta, Sarmistha Neogy, Keshav Dahal, and Zeeshan Pervez. «Internet of things (IoT): a survey on architecture, enabling technologies, applications and challenges». In: *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*. IML '17. Liverpool, United Kingdom: Association for Computing Machinery, 2017. ISBN: 9781450352437. DOI: `10.1145/3109761.3109768`. URL: `https://doi.org/10.1145/3109761.3109768` (cit. on p. 6).

[7] Pawan Kumar Verma et al. «Machine-to-Machine (M2M) communications: A survey». In: *Journal of Network and Computer Applications* 66 (2016), pp. 83–105. ISSN: 1084-8045. DOI: `https://doi.org/10.1016/j.jnca.2016.02.016`. URL: `https://www.sciencedirect.com/science/article/pii/S1084804516000990` (cit. on p. 6).

[8] Somayya Madakam, R Ramaswamy, Siddharth Tripathi, and Balqes Mujahed. «Internet of Things (IoT): A Literature Review». In: (Dec. 2022) (cit. on p. 6).

[9] Krishan Goyal, Amit Garg, Ankur Rastogi, and Saurabh Singhal. «A Literature Survey on Internet of Things (IoT)». In: *International Journal of Advanced Manufacturing Technology* 9 (Jan. 2018), pp. 3663–3668 (cit. on p. 7).

[10] P. Suresh, J. Vijay Daniel, V. Parthasarathy, and R. H. Aswathy. «A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment». In: *2014 International Conference on Science Engineering and Management Research (ICSEMR)*. 2014, pp. 1–8. DOI: `10.1109/ICSEMR.2014.7043637` (cit. on p. 7).

[11] *The 'Only' Coke Machine on the Internet*. URL: `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf` (cit. on p. 7).

[12] Stephen Ornes. «The Internet of Things and the explosion of interconnectivity». In: *Proceedings of the National Academy of Sciences* 113.40 (2016), pp. 11059–11060. DOI: `10.1073/pnas.1613921113`. eprint: `https://www.pnas.org/doi/pdf/10.1073/pnas.1613921113`. URL: `https://www.pnas.org/doi/abs/10.1073/pnas.1613921113` (cit. on p. 7).

[13] Chhaya A Khanzode and Ravindra D Sarode. «Evolution of the world wide web: from web 1.0 to 6.0». In: *International journal of Digital Library services* 6.2 (2016), pp. 1–11 (cit. on p. 7).

[14] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. «The World-Wide Web». In: *Commun. ACM* 37.8 (Aug. 1994), pp. 76–82. ISSN: 0001-0782. DOI: `10.1145/179606.179671`. URL: `https://doi.org/10.1145/179606.179671` (cit. on p. 7).

[15] *Web of Things*. URL: `https://www.w3.org/WoT/` (cit. on p. 7).

[16] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. «From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices». In: *Architecting the Internet of Things*. Ed. by Dieter Uckelmann, Mark Harrison, and Florian Michahelles. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 97–129. ISBN: 978-3-642-19157-2. DOI: `10.1007/978-3-642-19157-2_5`. URL: `https://doi.org/10.1007/978-3-642-19157-2_5` (cit. on p. 7).

[17] Nour Oweis, Claudio Aracenay, Waseem George, Mona Oweis, Hussein Soori, and Vaclav Snasel. «Internet of Things: Overview, Sources, Applications and Challenges». In: vol. 427. Jan. 2016, pp. 57–67. ISBN: 978-3-319-29503-9. DOI: `10.1007/978-3-319-29504-6_7` (cit. on p. 7).

[18] Douglas Mauro and Kevin Schmidt. *Essential SNMP: Help for System and Network Administrators.* " O'Reilly Media, Inc.", 2005 (cit. on p. 7).

[19] John Romkey. «Toast of the IoT: The 1990 Interop Internet Toaster». In: *IEEE Consumer Electronics Magazine* 6.1 (2017), pp. 116–119. DOI: 10.1109/MCE.2016.2614740 (cit. on p. 7).

[20] *The Trojan Room Coffee Pot.* URL: https://www.cl.cam.ac.uk/coffee/qsf/coffee.html (cit. on p. 7).

[21] Quentin Stafford-Fraser. «On site: The life and times of the first Web Cam». In: *Commun. ACM* 44.7 (July 2001), pp. 25–26. ISSN: 0001-0782. DOI: 10.1145/379300.379327. URL: https://doi.org/10.1145/379300.379327 (cit. on p. 7).

[22] Paul Saffo. «Sensors: the next wave of innovation». In: *Commun. ACM* 40.2 (Feb. 1997), pp. 92–97. ISSN: 0001-0782. DOI: 10.1145/253671.253734. URL: https://doi.org/10.1145/253671.253734 (cit. on p. 7).

[23] SB Prapulla, G Shobha, and T Thanuja. «Smart refrigerator using internet of things». In: *Journal of Multidisciplinary Engineering Science and Technology* 2.1 (2015), pp. 1795–1801 (cit. on p. 8).

[24] Abrar Mohammed. «Implementation of Smart Refrigerator based on Internet of Things». In: *IJITEE (International Journal of Information Technology and Electrical Engineering)* 9 (Dec. 2019). DOI: 10.35940/ijitee.B6343.129219 (cit. on p. 8).

[25] *The History of RFID Technology.* URL: https://www.rfidjournal.com/the-history-of-rfid-technology (cit. on p. 8).

[26] Aman Ullah. «IoT: Applications of RFID and issues». In: *International journal of internet of things and web services* 3 (2018) (cit. on p. 8).

[27] Stevan Preradovic, Nemai C. Karmakar, and Isaac Balbin. «RFID Transponders». In: *IEEE Microwave Magazine* 9.5 (2008), pp. 90–103. DOI: 10.1109/MMM.2008.927637 (cit. on p. 8).

[28] Xiaolin Jia, Quanyuan Feng, Taihua Fan, and Quanshui Lei. «RFID technology and its applications in Internet of Things (IoT)». In: *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet).* 2012, pp. 1282–1285. DOI: 10.1109/CECNet.2012.6201508 (cit. on p. 8).

[29] *The Internet of Things.* URL: https://www.itu.int/net/wsis/tunis/newsroom/stats/The-Internet-of-Things-2005.pdf (cit. on p. 8).

[30] Xian-Yi Chen and Zhi-Gang Jin. «Research on Key Technology and Applications for Internet of Things». In: *Physics Procedia* 33 (2012). 2012 International Conference on Medical Physics and Biomedical Engineering (ICMPBE2012), pp. 561–566. ISSN: 1875-3892. DOI: `https://doi.org/10.1016/j.phpro.2012.05.104`. URL: `https://www.sciencedirect.com/science/article/pii/S1875389212014174` (cit. on p. 8).

[31] Gustavo Ramirez Gonzalez, Mario Muñoz Organero, and Carlos Delgado Kloos. «Early infrastructure of an internet of things in spaces for learning». In: *2008 eighth IEEE international conference on advanced learning technologies*. IEEE. 2008, pp. 381–383 (cit. on p. 8).

[32] *International Conference for Industry and Academia*. URL: `https://iot-conference.org/iot2008/` (cit. on p. 8).

[33] Christian Floerkemeier. *The Internet of Things: First International Conference, IOT 2008, Zurich, Switzerland, March 26-28, 2008, Proceedings*. Vol. 4952. Springer Science & Business Media, 2008 (cit. on p. 8).

[34] *IPSO Alliance*. URL: `https://omaspecworks.org/ipso-alliance/` (cit. on p. 8).

[35] De-Li Yang, Feng Liu, and Yi-Duo Liang. «A Survey of the Internet of Things». In: *Proceedings of the 1st International Conference on E-Business Intelligence (ICEBI 2010)*. Atlantis Press, 2010/12, pp. 524–532. ISBN: 978-90-78677-40-6. DOI: `10.2991/icebi.2010.72`. URL: `https://doi.org/10.2991/icebi.2010.72` (cit. on p. 8).

[36] Luigi Atzori, Antonio Iera, and Giacomo Morabito. «The Internet of Things: A survey». In: *Computer Networks* 54.15 (2010), pp. 2787–2805. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2010.05.010`. URL: `https://www.sciencedirect.com/science/article/pii/S1389128610001568` (cit. on p. 8).

[37] Dave Evans. «The internet of things». In: *How the Next Evolution of the Internet is Changing Everything, Whitepaper, Cisco Internet Business Solutions Group (IBSG)* 1 (2011), pp. 1–12 (cit. on p. 8).

[38] *Internet of Things — An action plan for Europe*. URL: `https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2009:0278:FIN:EN:PDF` (cit. on p. 8).

[39] Tianyu Cui, Chang Liu, Gaopeng Gou, Junzheng Shi, and Gang Xiong. «A Comprehensive Study of Accelerating IPv6 Deployment». In: *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. 2019, pp. 1–8. DOI: `10.1109/IPCCC47392.2019.8958771` (cit. on p. 8).

[40] Geoff Huston. *Testing IPv6 for World IPv6 Day.* 2011 (cit. on p. 8).

[41] Tianyu Cui, Chang Liu, Gaopeng Gou, Junzheng Shi, and Gang Xiong. «A Comprehensive Study of Accelerating IPv6 Deployment». In: *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC).* 2019, pp. 1–8. DOI: `10.1109/IPCCC47392.2019.8958771` (cit. on p. 8).

[42] Nuno Miguel Carvalho Galego, Rui Miguel Pascoal, and Pedro Ramos Brandão. «IPv6 in IoT». In: *Management, Tourism and Smart Technologies.* Ed. by Carlos Montenegro, Álvaro Rocha, and Juan Manuel Cueva Lovelle. Cham: Springer Nature Switzerland, 2024, pp. 89–94. ISBN: 978-3-031-44131-8 (cit. on p. 9).

[43] Antonio J Jara, Socrates Varakliotis, Antonio F Skarmeta, and Peter Kirstein. «Extending the Internet of Things to the Future Internet through IPv6 support». In: *Mobile Information Systems* 10.1 (2014), pp. 3–17 (cit. on p. 9).

[44] *AWS IoT.* URL: `https://aws.amazon.com/iot/` (cit. on p. 9).

[45] *Azure IoT.* URL: `https://azure.microsoft.com/en-us/solutions/iot` (cit. on p. 9).

[46] Daniel Bastos. «Cloud for IoT—A survey of technologies and security features of public cloud IoT solutions». In: *Living in the Internet of Things (IoT 2019).* IET. 2019, pp. 1–6 (cit. on p. 9).

[47] *State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time.* URL: `https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/` (cit. on p. 9).

[48] *Current IoT Forecast Highlights.* URL: `https://transformainsights.com/research/forecast/highlights` (cit. on p. 9).

[49] *IoT Connections Forecast to 2030.* URL: `https://data.gsmaintelligence.com/research/research/research-2023/iot-connections-forecast-to-2030` (cit. on p. 9).

[50] Shadi Al-Sarawi, Mohammed Anbar, Rosni Abdullah, and Ahmad B. Al Hawari. «Internet of Things Market Analysis Forecasts, 2020–2030». In: *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4).* 2020, pp. 449–453. DOI: `10.1109/WorldS450073.2020.9210375` (cit. on p. 9).

[51] Yusuf Perwej, Kashiful Haq, Firoj Parwej, M Mumdouh, and Mohamed Hassan. «The internet of things (IoT) and its application domains». In: *International Journal of Computer Applications* 975.8887 (2019), p. 182 (cit. on p. 9).

[52] Sapandeep Kaur and Ikvinderpal Singh. «A survey report on Internet of Things applications». In: *International Journal of Computer Science Trends and Technology* 4.2 (2016), pp. 330–335 (cit. on p. 9).

[53] Hugh Boyes, Bil Hallaq, Joe Cunningham, and Tim Watson. «The industrial internet of things (IIoT): An analysis framework». In: *Computers in Industry* 101 (2018), pp. 1–12. ISSN: 0166-3615. DOI: `https://doi.org/10.1016/j.compind.2018.04.015`. URL: `https://www.sciencedirect.com/science/article/pii/S0166361517307285` (cit. on p. 11).

[54] *Cisco Annual Internet Report (2018–2023).* URL: `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf` (cit. on p. 11).

[55] Ivan Cvitić, Dragan Peraković, Marko Periša, Marko Krstić, and Brij Gupta. «Analysis of IoT concept applications: Smart home perspective». In: *International Conference on Future Access Enablers of Ubiquitous and Intelligent Infrastructures.* Springer. 2021, pp. 167–180 (cit. on p. 11).

[56] David Vasicek, Jakub Jalowiczor, Lukas Sevcik, and Miroslav Voznak. «IoT smart home concept». In: *2018 26th Telecommunications Forum (TELFOR).* IEEE. 2018, pp. 1–4 (cit. on p. 12).

[57] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. «A Review of Smart Homes—Past, Present, and Future». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1190–1203. DOI: `10.1109/TSMCC.2012.2189204` (cit. on p. 12).

[58] Vasanth Williams, Jude Immaculate, et al. «Survey on Internet of Things based smart home». In: *2019 International Conference on Intelligent Sustainable Systems (ICISS).* IEEE. 2019, pp. 460–464 (cit. on p. 12).

[59] Sam Solaimani, Wally Keijzer-Broers, and Harry Bouwman. «What we do–and don't–know about the Smart Home: An analysis of the Smart Home literature». In: *Indoor and Built Environment* 24.3 (2015), pp. 370–383 (cit. on p. 12).

[60] Michael Schiefer. «Smart home definition and security threats». In: *2015 ninth international conference on IT security incident management & IT forensics.* IEEE. 2015, pp. 114–118 (cit. on p. 12).

[61] Heetae Yang, Wonji Lee, and Hwansoo Lee. «IoT smart home adoption: the importance of proper level automation». In: *Journal of Sensors* 2018.1 (2018), p. 6464036 (cit. on p. 12).

[62] Changmin Lee, Luca Zappaterra, Kwanghee Choi, and Hyeong-Ah Choi. «Securing smart home: Technologies, security challenges, and security requirements». In: *2014 IEEE Conference on Communications and Network Security*. IEEE. 2014, pp. 67–72 (cit. on p. 13).

[63] Jawaher Abdulwahab Fadhil, Omar Ammar Omar, and Qusay Idrees Sarhan. «A survey on the applications of smart home systems». In: *2020 International Conference on Computer Science and Software Engineering (CSASE)*. IEEE. 2020, pp. 168–173 (cit. on p. 13).

[64] Sumathi Balakrishnan, Hemalata Vasudavan, and Raja Kumar Murugesan. «Smart home technologies: A preliminary review». In: *Proceedings of the 6th International Conference on Information Technology: IoT and Smart City*. 2018, pp. 120–127 (cit. on p. 13).

[65] Eileen Köhler and Daniel Spiekermann. «Smart Home as a Silent Witness - A Survey». In: *Proceedings of the 2022 European Interdisciplinary Cybersecurity Conference*. EICC '22. Barcelona, Spain: Association for Computing Machinery, 2022, pp. 12–16. ISBN: 9781450396035. DOI: `10.1145/3528580.3528583`. URL: `https://doi.org/10.1145/3528580.3528583` (cit. on p. 13).

[66] Anna Förster and Julian Block. «User adoption of smart home systems». In: *Proceedings of the 2022 ACM conference on information technology for social good*. 2022, pp. 360–365 (cit. on p. 13).

[67] Mariacristian Roscia, Vasile Dancu, and George Cristian Lazaroiu. «Smart Home Survey Analysis». In: *2023 IEEE International Smart Cities Conference (ISC2)*. IEEE. 2023, pp. 1–5 (cit. on p. 14).

[68] Tom Hargreaves, Charlie Wilson, and Richard Hauxwell-Baldwin. «Learning to live in a smart home». In: *Building Research & Information* 46.1 (2018), pp. 127–139 (cit. on p. 14).

[69] Samantha Reig, Elizabeth Jeanne Carter, Lynn Kirabo, Terrence Fong, Aaron Steinfeld, and Jodi Forlizzi. «Smart home agents and devices of today and tomorrow: Surveying use and desires». In: *Proceedings of the 9th International Conference on Human-Agent Interaction*. 2021, pp. 300–304 (cit. on p. 14).

[70] Mahsa Keshavarz and Mohd Anwar. «The Automatic Detection of Sensitive Data in Smart Homes». In: June 2019, pp. 404–416. ISBN: 978-3-030-22350-2. DOI: `10.1007/978-3-030-22351-9_27` (cit. on p. 14).

[71] Georgios Lampropoulos, Kerstin Siakas, and Theofylaktos Anastasiadis. «Internet of things (IoT) in industry: Contemporary application domains, innovative technologies and intelligent manufacturing». In: *people* 6.7 (2018) (cit. on p. 14).

[72] Noman Shahid and Sandhya Aneja. «Internet of Things: Vision, application areas and research challenges». In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. 2017, pp. 583–587. DOI: `10.1109/I-SMAC.2017.8058246` (cit. on p. 14).

[73] Laith Farhan, Sinan T Shukur, Ali E Alissa, Mohmad Alrweg, Umar Raza, and Rupak Kharel. «A survey on the challenges and opportunities of the Internet of Things (IoT)». In: *2017 Eleventh International Conference on Sensing Technology (ICST)*. IEEE. 2017, pp. 1–5 (cit. on p. 14).

[74] Arindam Giri, Subrata Dutta, Sarmistha Neogy, Keshav Dahal, and Zeeshan Pervez. «Internet of things (IoT): a survey on architecture, enabling technologies, applications and challenges». In: *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*. IML '17. Liverpool, United Kingdom: Association for Computing Machinery, 2017. ISBN: 9781450352437. DOI: `10.1145/3109761.3109768`. URL: `https://doi.org/10.1145/3109761.3109768` (cit. on p. 14).

[75] Shruti G Hegde Soumyalatha. «Study of IoT: understanding IoT architecture, applications, issues and challenges». In: *1st International Conference on Innovations in Computing & Net-working (ICICN16), CSE, RRCE. International Journal of Advanced Networking & Applications*. Vol. 478. 2016 (cit. on p. 14).

[76] Rob Van Kranenburg and Alex Bassi. «IoT challenges». In: *Communications in Mobile Computing* 1.1 (2012), p. 9 (cit. on p. 14).

[77] Sarah A. Al-Qaseemi, Hajer A. Almulhim, Maria F. Almulhim, and Saqib Rasool Chaudhry. *IoT architecture challenges and issues: Lack of standardization*. 2016. DOI: `10.1109/FTC.2016.7821686` (cit. on p. 15).

[78] Subramanian Balaji, Karan Nathani, and Rathnasamy Santhakumar. «IoT technology, applications and challenges: a contemporary survey». In: *Wireless personal communications* 108 (2019), pp. 363–388 (cit. on p. 16).

[79] *2023 SonicWall Cyber Threat Report*. URL: `https://www.sonicwall.com/2023-cyber-threat-report/` (cit. on p. 16).

[80] Simon Kramer and Julian C Bradfield. «A general definition of malware». In: *Journal in computer virology* 6 (2010), pp. 105–114 (cit. on p. 16).

[81] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub. «A survey on malware and malware detection systems». In: *International Journal of Computer Applications* 67.16 (2013) (cit. on p. 16).

[82] Prajoy Podder, M. Rubaiyat Hossain Mondal, Subrato Bharati, and Pinto Kumar Paul. «Review on the Security Threats of Internet of Things». In: *International Journal of Computer Applications* 176.41 (July 2020), pp. 37–45. ISSN: 0975-8887. DOI: `10.5120/ijca2020920548`. URL: `http://dx.doi.org/10.5120/ijca2020920548` (cit. on p. 16).

[83] Panagiotis I Radoglou Grammatikis, Panagiotis G Sarigiannidis, and Ioannis D Moscholios. «Securing the Internet of Things: Challenges, threats and solutions». In: *Internet of Things* 5 (2019), pp. 41–70 (cit. on p. 16).

[84] Ioannis Andrea, Chrysostomos Chrysostomou, and George Hadjichristofi. «Internet of Things: Security vulnerabilities and challenges». In: *2015 IEEE symposium on computers and communication (ISCC)*. IEEE. 2015, pp. 180–187 (cit. on p. 16).

[85] Mazwa Khawla and Mazri Tomader. «A survey on the security of smart homes: issues and solutions». In: *Proceedings of the 2nd International Conference on Smart Digital Environment*. 2018, pp. 81–87 (cit. on p. 16).

[86] Shruti Kajwadkar and Vinod Kumar Jain. «A Novel Algorithm for DoS and DDoS attack detection in Internet Of Things». In: *2018 Conference on Information and Communication Technology (CICT)*. 2018, pp. 1–4. DOI: `10.1109/INFOCOMTECH.2018.8722397` (cit. on p. 16).

[87] Navdeep Kaur and Maninder Singh. «Botnet and botnet detection techniques in cyber realm». In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. Vol. 3. 2016, pp. 1–7. DOI: `10.1109/INVENTIVE.2016.7830080` (cit. on p. 17).

[88] Liying Sun. «Who Can Fix the Spyware Problem?» In: *Berkeley Tech. LJ* 22 (2007), p. 555 (cit. on p. 17).

[89] Muhammad Aqeel, Fahad Ali, Muhammad Waseem Iqbal, Toqir A Rana, Muhammad Arif, and Md Rabiul Auwul. «A review of security and privacy concerns in the internet of things (IoT)». In: *Journal of Sensors* 2022.1 (2022), p. 5724168 (cit. on p. 17).

[90] Sam L Thomas and Aurélien Francillon. «Backdoors: Definition, deniability and detection». In: *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer. 2018, pp. 92–113 (cit. on p. 17).

[91] KrishnaKanth Gupta and Sapna Shukla. «Internet of Things: Security challenges for next generation networks». In: *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*. 2016, pp. 315–318. DOI: 10.1109/ICICCS.2016.7542301 (cit. on p. 17).

[92] Olivier Brun, Yonghua Yin, Javier Augusto-Gonzalez, Manuel Ramos, and Erol Gelenbe. «Iot attack detection with deep learning». In: *ISCIS Security Workshop*. 2018 (cit. on p. 17).

[93] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. «A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures». In: *IEEE Access* 7 (2019), pp. 82721–82743. DOI: 10.1109/ACCESS.2019.2924045 (cit. on p. 17).

[94] Yuchen Yang, Longfei Wu, Guisheng Yin, Lijie Li, and Hongbin Zhao. «A Survey on Security and Privacy Issues in Internet-of-Things». In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1250–1258. DOI: 10.1109/JIOT.2017.2694844 (cit. on p. 17).

[95] Rwan Mahmoud, Tasneem Yousuf, Fadi Aloul, and Imran Zualkernan. «Internet of things (IoT) security: Current status, challenges and prospective measures». In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. 2015, pp. 336–341. DOI: 10.1109/ICITST.2015.7412116 (cit. on p. 17).

[96] Waqar Ali, Ghulam Dustgeer, Muhammad Awais, and Munam Ali Shah. «IoT based smart home: Security challenges, security requirements and solutions». In: *2017 23rd International Conference on Automation and Computing (ICAC)*. 2017, pp. 1–6. DOI: 10.23919/IConAC.2017.8082057 (cit. on p. 17).

[97] Tidiane Sylla, Mohamed Aymen Chalouf, Francine Krief, and Karim Samaké. «Context-aware security in the internet of things: a survey». In: *International journal of autonomous and adaptive communications systems* 14.3 (2021), pp. 231–263 (cit. on p. 17).

[98] Bogdan-Cosmin Chifor, Ion Bica, Victor-Valeriu Patriciu, and Florin Pop. «A security authorization scheme for smart home Internet of Things devices». In: *Future Generation Computer Systems* 86 (2018), pp. 740–749 (cit. on p. 17).

[99] S. Sicari, A. Rizzardi, L.A. Grieco, and A. Coen-Porisini. «Security, privacy and trust in Internet of Things: The road ahead». In: *Computer Networks* 76 (2015), pp. 146–164. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2014.11.008. URL: https://www.sciencedirect.com/science/article/pii/S1389128614003971 (cit. on p. 18).

[100] Chalwe Musonda, MK Monica, Mayumbo Nyirenda, and Jackson Phiri. «Security, Privacy and Integrity in Internet of Things–A Review». In: *Proceedings of the ICTSZ International Conference in ICTs*. 2019, pp. 148–152 (cit. on p. 18).

[101] Slavko J Pokorni. «Reliability and availability of the Internet of things». In: *Vojnotehnicki glasnik/Military Technical Courier* 67.3 (2019), pp. 588–600 (cit. on p. 18).

[102] Ioannis Antzoulis, Md Minhaz Chowdhury, and Shadman Latiff. «IoT Security for Smart Home: Issues and Solutions». In: *2022 IEEE International Conference on Electro Information Technology (eIT)*. 2022, pp. 1–7. DOI: `10.1109/eIT53891.2022.9813914` (cit. on p. 18).

[103] Issam El Naqa, Ruijiang Li, and Martin J Murphy. «Machine Learning in Radiation Oncology Theory and Applications». In: () (cit. on p. 18).

[104] Eirini Anthi, Lowri Williams, Amir Javed, and Pete Burnap. «Hardening machine learning denial of service (DoS) defences against adversarial attacks in IoT smart home networks». In: *Computers Security* 108 (2021), p. 102352. ISSN: 0167-4048. DOI: `https://doi.org/10.1016/j.cose.2021.102352`. URL: `https://www.sciencedirect.com/science/article/pii/S0167404821001760` (cit. on p. 18).

[105] Akanksha Kaushik, Archana Choudhary, Chinmay Ektare, Deepti Thomas, and Syed Akram. «Blockchain — Literature survey». In: *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. 2017, pp. 2145–2148. DOI: `10.1109/RTEICT.2017.8256979` (cit. on p. 18).

[106] Bandar Alotaibi. «Utilizing Blockchain to Overcome Cyber Security Concerns in the Internet of Things: A Review». In: *IEEE Sensors Journal* 19.23 (2019), pp. 10953–10971. DOI: `10.1109/JSEN.2019.2935035` (cit. on p. 18).

[107] Chu Jay Tan, Junita Mohamad-Saleh, Khairu Anuar Mohamed Zain, and Zulfiqar Ali Abd. Aziz. «Review on Firmware». In: *Proceedings of the International Conference on Imaging, Signal Processing and Communication*. ICISPC 2017. Penang, Malaysia: Association for Computing Machinery, 2017, pp. 186–190. ISBN: 9781450352895. DOI: `10.1145/3132300.3132337`. URL: `https://doi.org/10.1145/3132300.3132337` (cit. on p. 19).

[108] Nai-Wei Lo and Sheng-Hsiang Hsu. «A secure IoT firmware update framework based on MQTT protocol». In: *Information Systems Architecture and Technology: Proceedings of 40th Anniversary International Conference on Information Systems Architecture and Technology–ISAT 2019: Part I*. Springer. 2020, pp. 187–198 (cit. on p. 19).

[109] 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 (cit. on p. 19).

[110] Shahid Ul Haq, Yashwant Singh, Amit Sharma, Rahul Gupta, and Dipak Gupta. «A survey on IoT & embedded device firmware security: architecture, extraction techniques, and vulnerability analysis frameworks». In: *Discover Internet of Things* 3.1 (Oct. 2023), p. 17. ISSN: 2730-7239. DOI: `10.1007/s43926-023-00045-2`. URL: `https://doi.org/10.1007/s43926-023-00045-2` (cit. on p. 19).

[111] Daojing He, Hongjie Gu, Tinghui Li, Yongliang Du, Xiaolei Wang, Sencun Zhu, and Nadra Guizani. «Toward hybrid static-dynamic detection of vulnerabilities in IoT firmware». In: *IEEE Network* 35.2 (2020), pp. 202–207 (cit. on p. 19).

[112] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. «Using Static Analysis to Find Bugs». In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: `10.1109/MS.2008.130` (cit. on p. 19).

[113] Denis Gopan and Thomas Reps. «Guided static analysis». In: *International Static Analysis Symposium*. Springer. 2007, pp. 349–365 (cit. on p. 19).

[114] Sreeja Nair, Raoul Jetley, Anil Nair, and Stefan Hauck-Stattelmann. «A static code analysis tool for control system software». In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 459–463. DOI: `10.1109/SANER.2015.7081856` (cit. on p. 19).

[115] David Evans. «Static detection of dynamic memory errors». In: *ACM SIGPLAN Notices* 31.5 (1996), pp. 44–53 (cit. on p. 19).

[116] Pär Emanuelsson and Ulf Nilsson. «A comparative study of industrial static analysis tools». In: *Electronic notes in theoretical computer science* 217 (2008), pp. 5–21 (cit. on p. 19).

[117] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. «Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube». In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 209–219. DOI: `10.1109/ICPC.2019.00040` (cit. on p. 19).

[118] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. «A critical comparison on six static analysis tools: Detection, agreement, and precision». In: *Journal of Systems and Software* 198 (2023), p. 111575 (cit. on p. 19).

[119] Devin Kester, Martin Mwebesa, and Jeremy S. Bradbury. «How Good is Static Analysis at Finding Concurrency Bugs?» In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 2010, pp. 115–124. DOI: `10.1109/SCAM.2010.26` (cit. on p. 20).

[120] Andreas Ibing and Alexandra Mai. «A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops». In: *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. 2015, pp. 44–51. DOI: `10.1109/HASE.2015.16` (cit. on p. 20).

[121] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. «An uncaught exception analysis for Java». In: *Journal of systems and software* 72.1 (2004), pp. 59–69 (cit. on p. 20).

[122] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. «A static analysis framework for detecting SQL injection vulnerabilities». In: *31st annual international computer software and applications conference (COMPSAC 2007)*. Vol. 1. IEEE. 2007, pp. 87–96 (cit. on p. 20).

[123] Hossain Shahriar and Mohammad Zulkernine. «Classification of Static Analysis-Based Buffer Overflow Detectors». In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*. 2010, pp. 94–101. DOI: `10.1109/SSIRI-C.2010.28` (cit. on p. 20).

[124] P. Louridas. «Static code analysis». In: *IEEE Software* 23.4 (2006), pp. 58–61. DOI: `10.1109/MS.2006.114` (cit. on p. 20).

[125] Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977 (cit. on p. 20).

[126] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee, and Brittany Johnson. «Evaluating how static analysis tools can reduce code review effort». In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017, pp. 101–105. DOI: `10.1109/VLHCC.2017.8103456` (cit. on p. 20).

[127] Peng Li and Baojiang Cui. «A comparative study on software vulnerability static analysis techniques and tools». In: *2010 IEEE International Conference on Information Theory and Information Security*. 2010, pp. 521–524. DOI: `10.1109/ICITIS.2010.5689543` (cit. on p. 21).

[128] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. «User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics». In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2016, pp. 100–107. DOI: `10.1109/QRS.2016.22` (cit. on p. 22).

[129] Miroslaw Ochodek, Krzysztof Durczak, Jerzy Nawrocki, and Miroslaw Staron. «Mining Task-Specific Lines of Code Counters». In: *IEEE Access* 11 (2023), pp. 100218–100233. DOI: `10.1109/ACCESS.2023.3314572` (cit. on p. 22).

[130] Vinny Kristina Sihombing, Mario ES Simaremare, Deni Josua Samosir, and Ventina Otani Limbong. «Improving Prometer, A Measure For Programmer Performance». In: *2022 IEEE International Conference of Computer Science and Information Technology (ICOSNIKOM)*. IEEE. 2022, pp. 01–06 (cit. on p. 22).

[131] Seymour V. Pollack and Ron K. Cytron. «Statement». In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 1683–1685. ISBN: 0470864125 (cit. on p. 22).

[132] Ayman Madi, Oussama Kassem Zein, and Seifedine Kadry. «On the improvement of cyclomatic complexity metric». In: *International Journal of Software Engineering and Its Applications* 7.2 (2013), pp. 67–82 (cit. on p. 22).

[133] DI De Silva, N Kodagoda, and H Perera. «Applicability of three complexity metrics». In: *International Conference on Advances in ICT for Emerging Regions (ICTer2012)*. IEEE. 2012, pp. 82–88 (cit. on p. 22).

[134] DI De Silva, N Kodagoda, and H Perera. «Applicability of three complexity metrics». In: *International Conference on Advances in ICT for Emerging Regions (ICTer2012)*. IEEE. 2012, pp. 82–88 (cit. on p. 22).

[135] G. Ann Campbell. «Cognitive complexity: an overview and evaluation». In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 57–58. ISBN: 9781450357135. DOI: `10.1145/3194164.3194186`. URL: `https://doi.org/10.1145/3194164.3194186` (cit. on p. 23).

[136] Jitender Kumar Chhabra. «Code cognitive complexity: a new measure». In: *Proceedings of the World Congress on Engineering*. Vol. 2. International Association of Engineers Hong Kong, China. 2011, pp. 6–8 (cit. on p. 23).

[137] Anjana Gosain and Ganga Sharma. «A Survey of Dynamic Program Analysis Techniques and Tools». In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Ed. by Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal. Cham: Springer International Publishing, 2015, pp. 113–122. ISBN: 978-3-319-11933-5 (cit. on p. 25).

[138] Michael D. Ernst. «Static and dynamic analysis: Synergy and duality». In: *WODA 2003: Workshop on Dynamic Analysis*. Portland, OR, USA, May 2003, pp. 24–27 (cit. on p. 25).

[139] Sanjay Kumar Singh and Amarjeet Singh. *Software testing*. Vandana Publications, 2012 (cit. on p. 26).

[140] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. «Code coverage and test suite effectiveness: Empirical study with real bugs in large systems». In: *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE. 2015, pp. 560–564 (cit. on p. 26).

[141] Brian Marick, John Smith, and Mark Jones. «How to misuse code coverage». In: *Proceedings of the 16th Interational Conference on Testing Computer Software*. 1999, pp. 16–18 (cit. on p. 26).

[142] TW Williams, MR Mercer, JP Mucha, and R Kapur. «Code coverage, what does it mean in terms of quality?» In: *Annual reliability and maintainability symposium. 2001 Proceedings. International symposium on product quality and integrity (Cat. No. 01CH37179)*. IEEE. 2001, pp. 420–424 (cit. on p. 26).

[143] *Codecov Coverage Thresholds*. URL: `https://about.codecov.io/blog/identify-coverage-holes-with-the-codecov-sunburst-chart/` (cit. on p. 26).

[144] *Rust*. URL: `https://github.com/rust-lang/rust` (cit. on p. 26).

[145] Chat Room. «Rust (Language)». In: *system* 5.32 (2022), p. 23 (cit. on p. 27).

[146] William Bugden and Ayman Alahmar. «Rust: The programming language for safety and performance». In: *arXiv preprint arXiv:2206.05503* (2022) (cit. on p. 27).

[147] Nicholas D. Matsakis and Felix S. Klock. «The rust language». In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 103–104. ISBN: 9781450332170. DOI: `10.1145/2663171.2663188`. URL: `https://doi.org/10.1145/2663171.2663188` (cit. on p. 27).

[148] Tunç Uzlu and Ediz Şaykol. «On utilizing rust programming language for Internet of Things». In: *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*. 2017, pp. 93–96. DOI: `10.1109/CICN.2017.8319363` (cit. on p. 28).

[149] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. «Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller». In: *Electronics* 12.1 (2022), p. 143 (cit. on p. 28).

[150] Eireann Leverett, Richard Clayton, and Ross Anderson. «Standardisation and Certification of the 'Internet of Things'». In: *Proceedings of WEIS*. Vol. 2017. 2017 (cit. on p. 28).

[151] André Cirne, Patrícia R Sousa, João S Resende, and Luís Antunes. «IoT security certifications: Challenges and potential approaches». In: *Computers & Security* 116 (2022), p. 102669 (cit. on p. 28).

[152] Gianmarco Baldini, Antonio Skarmeta, Elizabeta Fourneret, Ricardo Neisse, Bruno Legeard, and Franck Le Gall. «Security certification and labelling in Internet of Things». In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. 2016, pp. 627–632. DOI: `10.1109/WF-IoT.2016.7845514` (cit. on p. 28).

[153] *OWASP IoT Top 10*. URL: `https://owasp.org/www-project-internet-of-things/` (cit. on p. 28).

[154] Ryan Johnson, Zhaohui Wang, Corey Gagnon, and Angelos Stavrou. «Analysis of android applications' permissions». In: *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*. IEEE. 2012, pp. 45–46 (cit. on p. 29).

[155] Zainab Masood, Rashina Hoda, Kelly Blincoe, and Daniela Damian. «Like, dislike, or just do it? How developers approach software development tasks». In: *Information and Software Technology* 150 (2022), p. 106963 (cit. on p. 30).

[156] Philipp Straubinger and Gordon Fraser. «A Survey on What Developers Think About Testing». In: *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 2023, pp. 80–90 (cit. on p. 30).

[157] Thomas Borchert. *Code profiling: Static code analysis*. 2008 (cit. on p. 34).

[158] Soma Pal. «Properties of Profile-Guided Compiler Optimizations with GCC and LLVM». PhD thesis. University of Kansas, 2022 (cit. on p. 34).

[159] Alexandre Bergel, Felipe Banados, Romain Robbes, and Walter Binder. «Execution profiling blueprints». In: *Software: Practice and Experience* 42.9 (2012), pp. 1165–1192 (cit. on p. 34).

[160] Alexandre Bergel, Felipe Banados, Romain Robbes, and David Röthlisberger. «Spy: A flexible code profiling framework». In: *Computer Languages, Systems & Structures* 38.1 (2012), pp. 16–28 (cit. on p. 34).

[161] Jasmin Blanchette. «The little manual of API design». In: *Trolltech, Nokia* (2008) (cit. on p. 41).

[162] Joshua Bloch. «How to design a good API and why it matters». In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 506–507 (cit. on p. 41).

[163] *Rust Builder Pattern*. URL: `https://rust-unofficial.github.io/patterns/patterns/creational/builder.html` (cit. on p. 42).

[164] P. Koopman. *Better Embedded System Software*. Drumnadrochit Education, 2010. ISBN: 9780984449002. URL: `https://books.google.it/books?id=mxCtngEACAAJ` (cit. on p. 52).

[165] *CRAP*. URL: `https://testing.googleblog.com/2011/02/this-code-is-crap.html` (cit. on p. 52).

[166] *Skunk*. URL: `https://www.fastruby.io/blog/code-quality/introducing-skunk-stink-score-calculator.html` (cit. on p. 54).

[167] *Skunk Conference Presentation*. URL: `https://www.youtube.com/watch?v=ZyU6K6eR-_A&t=1492s` (cit. on p. 55).

[168] Karin Koogan Breitman, Marco Antonio Casanova, and Walter Truszkowski. «Ontology in computer science». In: *Semantic Web: Concepts, Technologies and Applications* (2007), pp. 17–34 (cit. on p. 62).

[169] Robert Stevens, Alan Rector, and Duncan Hull. «What is an ontology?» In: *Ontogenesis* (2010) (cit. on p. 62).

[170] Victor Charpenay, Sebastian Käbisch, and Harald Kosch. «Introducing Thing Descriptions and Interactions: An Ontology for the Web of Things.» In: *SR+ SWIT@ ISWC*. 2016, pp. 55–66 (cit. on p. 63).

[171] Prateek Jain, Pascal Hitzler, Peter Z Yeh, Kunal Verma, and Amit P Sheth. «Linked data is merely more data». In: *2010 AAAI Spring Symposium Series*. 2010 (cit. on p. 64).

[172] Peter Neish. «Linked data: what is it and why should you care?» In: *The Australian Library Journal* 64.1 (2015), pp. 3–10 (cit. on p. 64).

[173] Christian Bizer, Tom Heath, and Tim Berners-Lee. «Linked data-the story so far». In: *Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web*. 2023, pp. 115–143 (cit. on p. 64).

[174] Sander Stolk, Wouter Lubbers, Freek Braakman, and Sander Weitkamp. «Ontologies and JSON-LD at TenneT: The Use of Linked Data on EU-303 Projects.» In: *LDAC@ ESWC*. 2022, pp. 20–31 (cit. on p. 64).

[175] Kevin Angele23 and Jürgen Angele. «JSON towards a simple Ontology and Rule». In: (2021) (cit. on p. 64).

[176] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. «JSON-LD 1.1». In: *W3C Recommendation, Jul* (2020) (cit. on p. 65).

[177] Xinyang Feng, Jianjing Shen, and Ying Fan. «REST: An alternative to RPC for Web services architecture». In: *2009 First International Conference on future information networks*. IEEE. 2009, pp. 7–10 (cit. on p. 70).