Università di Genova | DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

School of Mathematical Physical and Natural Sciences

Bachelor of Science in Computer Science

Academic Year 2022-2023

Binary Instrumentation for runtime monitoring of Internet of Things systems through the Employment of Falco

Federico Elia

Supervisor:

Prof. Davide Ancona

INDEX

CHAPTER I 1

INTRODUCTION...1

1.1 Goals 1

1.2 Motivations 2

1.3 IoT 2

1.4 Thesis structure 3

CHAPTER II 5

INTERNET OF THINGS (IOT): OVERVIEW...5

2.1 IOT: Definition and importance of IoT in the current context.......................5

2.1.1 Overview of IoT applications in various sectors...6

2.1.2 Possible security holes in IoT systems and their impact.............................8

2.2 MQTT: introduction..11

2.2.1 Basic operation of MQTT 12

2.2.2 Importance of MQTT in IoT systems 14

2.2.3 MQTT protocol security issues 15

CHAPTER III...18

BINARY INSTRUMENTATION 18

3.1 Introduction and definition of binary instrumentation 18

3.2 Applications in the field of monitoring..20

3.3 Tools and technologies for binary instrumentation......................................22

3.4 Falco: Overview and Applications in IoT Security 24

3.4.1 Description of Falco 25

3.4.2 Key features and use cases of Falco..26

3.4.3 Falco for strengthening the security of IoT systems 30

3.5 RGB565 31

CHAPTER IV 33
RESEARCH CONTRIBUTION 33
4.1 Analysis of actors and system architecture 33
4.2 Falco User's Guide 35
4.2.1 Installation 35
4.2.2 Startup using custom rules 36
4.3 Developed software 37
4.3.1 Parse-mqttv5-packet library 37
4.3.2 Adapter: essential bridge between Falco and the Monitor 39
4.3.3 Monitor with specifications in RML 42
4.4 Performance Evaluation and Impact on Overhead 44
4.4.1 First test case: Realistic scenario 46
4.4.2 Second test case: Stress test 49
CONCLUSIONS 53
Bibliography 54
Sitography 57

Index of Figures

Figure 1. Some IoT application domains 7
Figure 2. The most popular IoT applications 8
Figure 3. IoT information security: how important is it? 10
Figure 4. Shows the example of using the MQTT protocol 13
Figure 5. Command and control scenario of botnet via MQTT 17
Figure 6. Static binary instrumentation process 19
Figure 7. Label-based instrumentation code. 21
Figure 8. Architecture of Falco 24
Figure 9. Scope of operation of Falco 25
Figure 10. Falco's call handling method 26
Figure 11. How Falco works 27
Figure 12. Call instrumentation system 28
Figure 13. Threat response method 29
Figure 14. Deployment diagram of the system under study 34
Figure 15. Sequence diagram of the Adapter 40
Figure 16. Ensuring sequentiality through FIFO: Flow Chart 42
Figure 17. Diagram on the operation of the Monitor in case of a match 43
Figure 18. Diagram on the operation of the Monitor in case of non-match 44
Figure 19. Example: bash script for level 2 data collection 45
Figure 20. Realistic scenario: CPU utilization in level 0 46
Figure 21. Realistic scenario: CPU utilization in layer 1 46
Figure 22. Realistic scenario: CPU utilization in layer 2 47
Figure 23. Realistic scenario: CPU utilization in layer 3 47
Figure 24. Realistic scenario: comparison CPU utilization by level 48
Figure 25. Realistic scenario: comparison of instructions executed by level 48
Figure 26. Realistic scenario: comparison of clock cycles by level 49
Figure 27. Stress test: CPU utilization in level 0 49
Figure 28. Stress test: CPU utilization in level 1 50
Figure 29. Stress test: CPU utilization in level 2 50
Figure 30. Stress test: comparison of CPU utilization by level 51
Figure 31. Stress test: comparison of instructions executed by level 51
Figure 32. Stress test: comparison of clock cycles by level 52

CHAPTER I

INTRODUCTION

1.1 Goals

This thesis aims to explore the Internet of Things (IoT) universe, initially outlining a comprehensive overview of this technological phenomenon and its importance in the current context. The primary objective is to offer a clear definition of IoT and to illustrate the wide range of applications that this technology enables in different sectors, emphasizing how IoT is revolutionizing how physical objects interact with the digital world. Particular attention is paid to identifying possible security holes in IoT systems, examining their impact on privacy and system security. Next, the thesis delves into the study of the MQTT protocol, a key technology in IoT systems because of its effectiveness in managing communication between devices. A detailed explanation of the basic operation of MQTT and the reasons for its importance is provided, highlighting the security problems that plague it and possible solutions to mitigate them.

This study aims to quantify and understand the effects of overhead caused by binary instrumentation and monitoring of smart devices, evaluating its impact on the performance of a real IoT system. The section then explores the technical background essential to assimilate the tools and methodologies used in the project, with a specific emphasis on binary instrumentation. The concept of binary instrumentation is introduced, its applications in monitoring are analyzed, and relevant tools and technologies are described, laying the foundation for future elaborations of the thesis.

A special focus is given to Falco, an emerging tool in the field of security. A comprehensive overview of Falco is offered, highlighting its main functionalities and application areas, with a focus on the integration of this technology to enhance security in IoT systems. Such integration aims to unveil how Falco can be effective in preventing or mitigating security threats, offering concrete insight into how binary instrumentation interacts with and influences operational IoT systems.

1.2 Motivations

This thesis focuses on exploration and innovation in the field of IoT, a technology that is revolutionizing the way we live, work and interact with the world around us. The IoT, with its vast network of connected devices, offers unlimited possibilities for improving the efficiency, convenience and security of our daily lives. However, with this pervasiveness, significant challenges emerge, particularly with regard to security and user privacy protection. Thus, the main motivation of this thesis is twofold: on the one hand, it aims to provide an in-depth understanding of IoT, highlighting its importance in the current context and exploring its applications in various fields. On the other hand, it aims to investigate and address the inherent vulnerabilities of IoT systems, focusing in particular on the MQTT protocol, which is widely used for communication between devices. This protocol, despite its effectiveness and efficiency, has critical security issues that can be exploited by malicious attackers.

1.3 IoT

The Internet of Things (IoT) is an emerging paradigm that transforms our interaction with the world by enabling electronic devices and sensors to communicate across the network, simplifying and enriching our lives. IoT uses smart devices and the Internet to provide innovative solutions to various challenges and problems related to various business, government and public/private sectors around the world. IoT is gradually becoming an important aspect of our lives that can be felt everywhere around us.

Overall, IoT is an innovation that brings together a wide variety of smart systems, frameworks, and smart devices and sensors.

Smart cities, smart homes, pollution control, energy saving, smart transportation, smart industries are transformations due to IoT.

IoT is the focus of significant research aimed at enhancing existing technologies. Despite advances, IoT still presents complex challenges that limit its full potential. These issues range from technical issues related to security and data privacy to social and environmental aspects. To maximize the benefits of IoT, it is critical to address these challenges by considering various aspects such as specific applications, the technologies involved, and the broader impact on society and the environment. This is possible by providing greater decision-making automated in real time and facilitating tools to optimize such decisions. Integrating renewable energy and optimizing energy use are key drivers for sustainable energy transitions and climate change mitigation, IoT can be employed to improve energy efficiency, increase the share of renewable energy, and reduce the environmental impact of energy use (Ali, 2015).

1.4 Thesis structure

This thesis is divided into three main parts, following a logical structure that guides the reader from an introduction to the context and relevance of IoT in today's technological landscape, through an in-depth discussion of specific technical aspects, to the presentation of an original research contribution of the thesis contribution:

1. The first part introduces the context and relevance of IoT in today's technological landscape, highlighting how its pervasiveness profoundly affects many aspects of modern society. The potential of IoT in various areas, from home automation to smart cities, from Industry 4.0 to healthcare, is discussed, highlighting the incredible opportunities offered by this technology but also the challenges that come with it, especially in terms of security. The paper focuses on MQTT, a key protocol for communication in IoT systems, describing how it works and its critical role in the IoT ecosystem. Security issues related to MQTT are analyzed, highlighting how its implementation can expose systems to significant risks and what measures can be taken to mitigate these vulnerabilities.
2. In the second part, we go into the technical background needed to understand the instruments used in the project, with a focus on binary instrumentation and Falco.
   An overview of binary instrumentation, its applications in monitoring and its importance in detecting and preventing cyber attacks is provided. Next, Falco is introduced, describing its key functionalities and how it can be integrated into IoT systems to strengthen their security.
3. In the third section, the thesis extensively explores its most significant contribution. This part starts with a comprehensive and detailed examination of the binary instrumentation process. The process is meticulously described, highlighting each step and its significance. The implementation of this process is carried out using Falco, and the discussion includes an analysis of how Falco is utilized to achieve the desired results, placing

special emphasis on how the latter is configured with custom rules to monitor specific behaviors or events within IoT systems. In parallel, an Adapter is introduced that acts as an intermediary between Falco and the monitor. This Adapter has the crucial task of processing the data received from Falco so that it can be interpreted effectively by the monitoring system developed in Prolog and in RML (Runtime Monitoring Language). The use of RML (Runtime Monitoring Language) is crucial in this context, as it is used to define the monitor specifications in a simple and concise manner.

This analysis focused on evaluating the effectiveness of the proposed tools for mitigating security risks in IoT systems. Particular attention was paid to measuring the overhead generated by the implementation of these tools, both individually and in combination. The goal was to provide a detailed overview of the implications of the overhead associated with each tool, thus enabling an optimal balance between security and performance.

In summary, this thesis aims to evaluate the effectiveness of such tools and methodologies in optimizing the security of IoT systems, thereby contributing to their increased resilience against cyber threats.

CHAPTER II

INTERNET OF THINGS (IOT): OVERVIEW

2.1 IOT: Definition and importance of IoT in the current context

The IoT constitutes one of the most significant technological revolutions of our time. This concept refers to a global infrastructure that physically connects the digital world to real-world devices and objects. The elements that make up this network-which can range from simple household sensors and actuators to complex industrial systems-are equipped with the technology needed to collect, send and receive data through the network.

This inter-connectivity allows devices to interact not only with each other but also with other platforms and services, facilitating a wide range of applications. For example, in a home environment, IoT can be used to manage temperature, lighting and security automatically and remotely. In the industrial sector, it can improve the efficiency of production processes through real-time collection and analysis of operational data.

IoT devices are equipped with sensors that detect various types of environmental or operational data, such as temperature, humidity, motion or light levels. This data is then transmitted to systems that can process it, often using cloud technologies for the analysis and persistence of voluminous data sets. The information collected and processed allows the system to make autonomous decisions, possibly mediated by human intervention, to operate the actuators and best pursue the intended goals.

The IoT is revolutionizing entire sectors: in industry it enables real-time monitoring of production, supply chain optimization and improved efficiency; in healthcare it enables remote monitoring of patients via wearable devices; in homes it underpins home automation and smart management of appliances and systems (Ali, 2015).

Despite the diversity of research on IoT, its definition remains somewhat fuzzy. There are several definitions reflecting different perspectives; the most important ones in the literature are given below:

- "IoT is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment.

IoT encompasses an ecosystem that includes things, communication, applications and data analytics" (Vuppalapati, 2019).

- "IoT is a group of infrastructures that interconnect sensors and/or actuators with (limited) computational capabilities and enable their management, access and transfer of the data they generate over the Internet without the need for human resources human-human or human-computer interaction" (Dorsemaine et al., 2015).
- "IoT refers to a decentralized system of enhanced devices with sensing, processing and networking capabilities" (Kortuem, 2009).

In industry, IoT plays a crucial role in automating and digitizing processes, enabling detailed, real-time investigation of operations, optimized supply chain management, and increased energy and production efficiency.

In healthcare, wearable devices and other IoT tools provide vital data for remote monitoring of patients, improving the quality of care and speed of intervention.

In the home, IoT transforms spaces into smart environments where every device is able to communicate to improve comfort and security. However, the enormous potential of IoT comes with significant challenges, accentuated by the limited computing power of IoT devices, which often cannot support advanced security protocols or host complex antivirus software. As a result, they become vulnerable targets for cyber attacks, exposing systems to significant risks (Ali, 2015).

2.1.1 Overview of IoT applications in various industries

IoT networks generate a huge amount of aggregated data through smart objects. Every two years, the size of the data doubles and is expected to reach 163 Zettabytes in 2025 (Beecks, 2018).

IoT and cloud computing both serve to increase efficiency for business and industry. IoT generates huge amounts of data, with cloud computing providing storage space and computing power for data analysis. The integration of IoT with medical technologies enables real-time monitoring system and data access to improve patient health (Mubeen, 2018).

For example, IoT devices can be used to track the real-time location of medical equipment such as wheelchairs, oxygen pumps, and other monitoring equipment. IoT systems based on wireless technology are being used to monitor a wide range of sensors in the field that can detect and measure various physical phenomena such as volcanic activity, floods and fires (Amarlingam, 2016).

These examples are just some of the types of applications developed in the IoT field. Figure 1 illustrates some of the areas that benefit from IoT technology (Amarlingam, 2016).

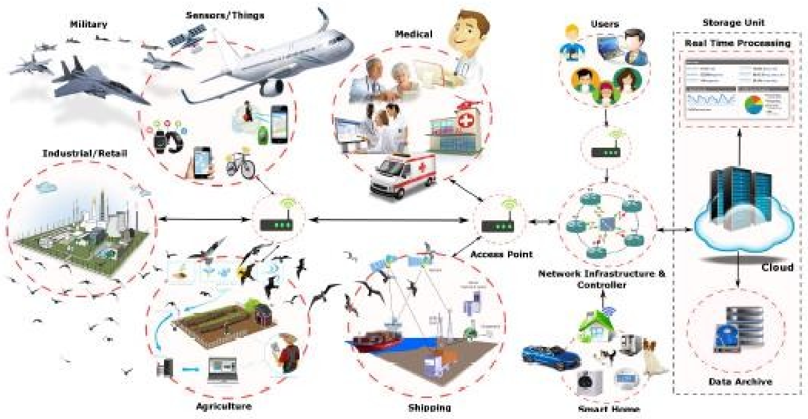


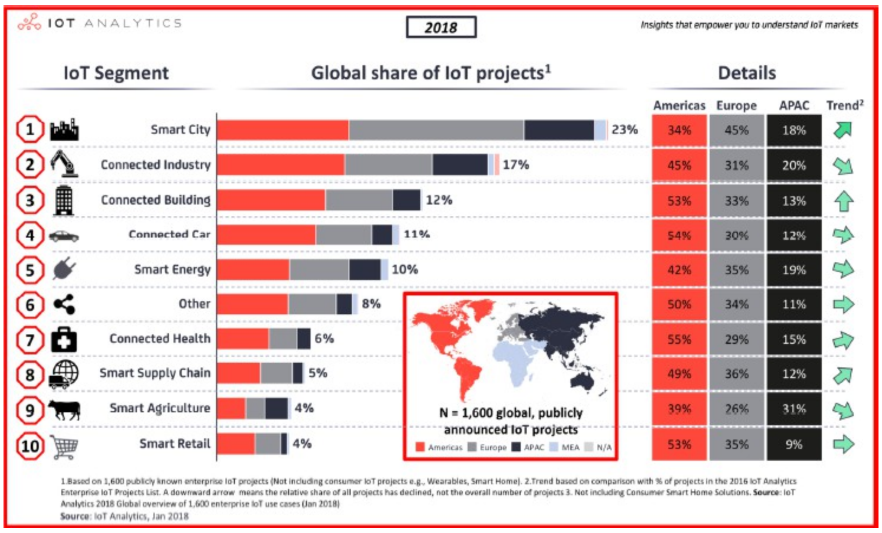
Figure 1. Some IoT application domains.

By connecting billions of devices to the Internet, the IoT has created a plethora of applications that touch every aspect of human life (Figure 2); to name just a few:

- **Smart** Cities-the use of sensors, network facilities and cloud-based integration systems to provide citizens with services and infrastructure and give them access to a wide range of real-time information about the urban environment on which to base decisions and actions (Jin et al., 2014).
- **Smart** Grids **-** electricity grids that can integrate the behaviors and actions of all users connected to it in a cost-effective manner to ensure an economically efficient and sustainable electricity system with low losses and high levels of supply quality and security (Guillemin, 2014).
- **Connected Health -** the collection, aggregation and effective processing and of information indicative of physical and mental health (Hassanalieragh, 2015).
- **Smart Homes -** a convenient home configuration in which appliances and devices can be controlled automatically or

remotely from anywhere connected to the Internet using a mobile or other networked device (Ghayvat, 2015).

- **Connected** Cars **-** cars that can connect to the Internet and share various types of data (location, speed, status of car parts, etc.) with back-end applications. useful, for example, for managing car sharing activities or alert systems in case of accidents or traffic problems (Dhall, 2017).



	IoT Segment	Global share of IoT projects	Americas	Europe	APAC
1	Smart City	23%	34%	45%	18%
2	Connected Industry	17%	45%	31%	20%
3	Connected Building	12%	53%	33%	13%
4	Connected Car	11%	54%	30%	12%
5	Smart Energy	10%	42%	35%	19%
6	Other	8%	50%	34%	11%
7	Connected Health	6%	55%	29%	15%
8	Smart Supply Chain	5%	49%	36%	12%
9	Smart Agriculture	4%	39%	26%	31%
10	Smart Retail	4%	53%	35%	9%

1.Based on 1,600 publicly known enterprise IoT projects (Not including consumer IoT projects e.g., Wearables, Smart Home). 2.Trend based on comparison with % of projects in the 2016 IoT Analytics Enterprise IoT Projects List. A downward arrow means the relative share of all projects has declined, not the overall number of projects 3. Not including Consumer Smart Home Solutions. **Source**: IoT Analytics 2018 Global overview of 1,600 enterprise IoT use cases (Jan 2018)
Source: IoT Analytics, Jan 2018

Figure 2. The most popular IoT applications

IoT applications can be divided according to their main requirements into three categories (Dıaz, 2016):

1. **Real-time:** applications that require real-time data monitoring and decision support, as is the case, for example, for the Connected Health, Smart Farming, and Smart Supply Chain domains;
2. **Data analytics:** applications focused on data analytics, e.g., Smart Retail, Smart City, and Smart Grid, which rely on data analytics to optimize business, cities, and power grids, respectively;
3. **Interaction with devices**: applications focus on the relationships between devices. In Smart Home, Wearables and Industrial Internet, interaction with devices is a key objective.

2.1.2 Possible security holes in IoT systems and their impact

The security aspect of this technology is significant as recent surveys and trends have highlighted its importance in this area. According to the National

Institute of Standards and Technology, information assurance is defined as "measures that safeguard and preserve information and information systems by ensuring their secrecy, verification, integrity, availability and non-repudiation." These steps include "the provision of information network restoration by integrating security, detection and response capabilities." Since IoT systems involve physical devices, communication networks and data resources, these five pillars of information assurance are relevant as security requirements (Russell, 2016).

IoT security requirements are critical to ensure the secure operation of interconnected devices and the data they produce. Strong authentication and access control mechanisms to prevent unauthorized access and defend against cyber attacks are essential IoT security requirements. These mechanisms must be able to identify and authenticate users and devices, control access to sensitive data, and provide granular permissions to ensure that only authorized entities can access the system. In addition, data generated by IoT devices should be encrypted and secured to ensure privacy and confidentiality and be protected from tampering to ensure their integrity and authenticity. Network and device security is another vital aspect of IoT security. IoT devices and systems must be protected from network-based attacks. Physical attacks, such as destruction, theft and tampering, should also be prevented by the security mechanisms built into IoT devices (Aldweesh, 2020).

IoT combines the physical and Internet-connected worlds to provide intelligent collaboration between physical entities and their surrounding environments. Typically, IoT devices operate in a variety of environments to achieve a variety of goals. Companies operating in this area must take extremely stringent security measures, both cyber and physical, to protect their devices and networks from potential threats.

The complexity stems not only from the variety and interdisciplinary nature of the components involved-which include hardware, software, communication networks, and data processing algorithms-but also from the vastness and diversity of IoT application scenarios. This complexity exponentially increases the attack surfaces available to attackers, making the task of keeping systems secure more arduous. Attacks can occur on multiple fronts: from the interception of communications wireless, which are often the means by which IoT devices exchange data, to the attempted direct physical access to devices made possible by their deployment in open or easily publicly accessible environments. Effectively addressing these challenges requires an approach to security that is all-encompassing and adaptable to the specific needs of the IoT. This means implementing solutions that not only protect communications and data, but are also designed with the inherent limitations of IoT devices, such as limited processing capacity and low power autonomy, in mind. Such solutions should include the use of lightweight encryption algorithms, robust authentication systems, secure communication protocols, and physical security practices (Aldweesh, 2020).

Consequently, preserving the privacy or security of IoT-based devices is a complex and difficult task that has attracted considerable interest in both academia and industry. Since the main goal of an IoT-based system is to provide easy access to anyone, anywhere, anytime, the possibilities for attack increase. IoT devices generate large amounts of data, which are transmitted over networks, making them vulnerable to cyber threats. Consequently, protecting the network and data in the IoT is essential for the security and protection of the entire system.

Network security measures provide the foundation to protect data in transit, while data security measures safeguard data in transit and at rest. To ensure the safe and secure operation of IoT devices and systems, a comprehensive security strategy must be adopted that includes both network and data security measures (Chaabouni, 2019).

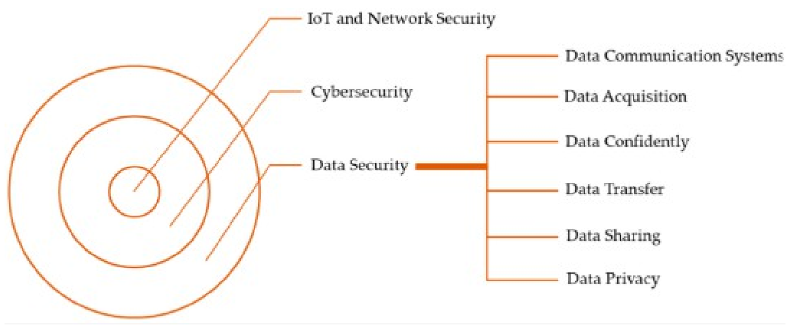


Figure 3. IoT information security: how important is it?

The difficulties in this area are related to three interrelated factors: first, due to the resource-constrained nature and mobility of IoT systems, data security methods must operate in a way that allows for very limited resource consumption.

Second, many IoT facilities are supported by data sharing; however, in data-sensitive contexts, secrecy is of utmost importance, which often presents numerous problems. Third, the need for data security increases significantly, particularly in the case of sensitive IoT services or apps. IoT security is a protection tactic and defense mechanism that protects against attacks that specifically target physically connected IoT devices. Network security protects the network and the data in it from intrusions, assaults and other threats. This is a broad and inclusive term that covers both software and hardware solutions, as well as procedures, guidelines and configurations for network use, accessibility and general threat avoidance. Encryption methods are only one aspect of the topic of data protection. A number of benefits arise from combining IoT with the Industry 4.0 paradigm, including better exploitation of IoT data. This relates to information sharing and other data-dependent operations that could take place anywhere in the system, even outside the boundaries of the organization. Although encryption methods enable preferential data exchange, this part elaborates on other strategies to maintain the confidentiality of IoT data (Figure 3). There are significant connections between IoT and network security, cybersecurity and data security.

Securing IoT devices and networks is critical to preventing cyber attacks. Network security protects the networks that connect IoT devices. Cybersecurity requires defending the entire IoT ecosystem from cyber attacks, including devices, networks, and apps (Taherdoost, 2023).

Data security is the safeguarding of data collected and communicated by IoT devices. This involves encrypting data during transmission and storing it securely. In addition, access restrictions and authentication systems are critical to prevent unwanted access to sensitive data. These data security measures are essential to protect the data collected and transmitted by IoT devices. IoT devices are susceptible to cyber threats and protecting them involves installing network security, cybersecurity and data protection measures (Al-Fuqaha, 2015).

2.2 MQTT: introduction

MQTT (Message Queuing Telemetry Transport) is a lightweight and efficient messaging protocol that has been widely adopted in the Internet of Things (IoT) for its excellent performance and communication model based on publish/subscribe. This model is particularly well suited to the needs of IoT architectures, as it allows devices to send (publish) information without having to establish a direct connection with receivers (subscribers), thus facilitating network scalability and efficiency. IoT devices can thus communicate with each other and with servers asynchronously, optimizing bandwidth usage and reducing energy consumption, crucial aspects for devices often characterized by limited resources (La Marra, 2017).

2.2.1 Basic operation of MQTT

The basic operation of MQTT is structured around two main entities: the client and the broker:

- The MQTT client is an essential element within the MQTT ecosystem, enabling devices to interact with a central broker for message exchange.

  As a publisher, the MQTT client transmits information related to specific topics.
- When it acts as a subscriber, it subscribes to the topics of its interest to receive messages disseminated there. This messaging architecture provides the flexibility for a single client to simultaneously receive information as a subscriber and broadcast it as a publisher, interacting with the same broker.

  In addition, a client can be subscribed to multiple topics, allowing him or her to receive data from multiple sources and effectively segment the information according to his or her interests or operational needs.
- The MQTT broker acts as the beating heart of the MQTT network, having the crucial task of managing message flows. It receives communications from publishers, classifies them according to topics and distributes them to relevant subscribers; this process is critical to ensuring that information is distributed correctly within the network, maintaining the efficiency and effectiveness of communication. In Linux-based environments, there are several MQTT brokers available, many of them open source, such as Mosquitto, EMQX, and HiveMQ.

The advantage of the publish/subscribe system is that publisher and subscriber do not know each other because there is the broker who acts as an intermediary, this creates a time decoupling that makes it impossible for the simultaneous connection of the subscriber and publisher, so that the customer remains to receive the previously delayed data (Atmoko, 2017).

Usually, in the MQTT architecture, different sensors periodically publish the results of their measurements to a specific topic. Each device registered to a specific topic will receive a message from the broker whenever the topic is updated, Figure 4 shows the example of using the MQTT protocol.

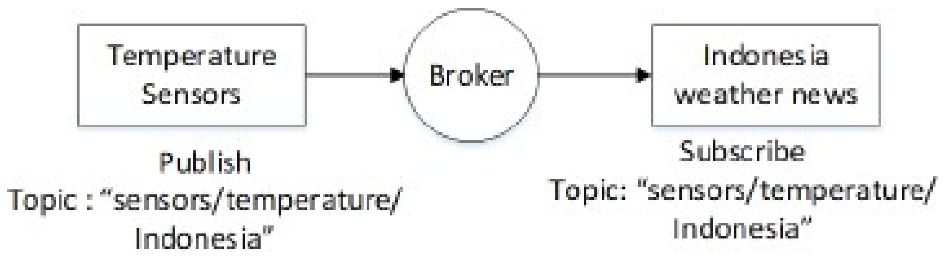

Temperature Sensors
Broker
Indonesia weather news
Publish
Topic : "sensors/temperature/ Indonesia"
Subscribe
Topic: "sensors/temperature/ Indonesia"

Figure 4. Shows the example of using the MQTT protocol.

Topics in MQTT-based messaging systems follow a hierarchical structure similar to paths in file systems. This means that messages can be published and subscribed to using topics that indicate levels of categorization via separators (usually the '/' character). For example, a topic might have the structure house/living room/temperature to indicate the temperature in the living room of a house. Wildcards are special characters used in topic subscriptions to indicate one or more levels of this hierarchy, allowing for more flexible subscriptions. The two commonly used wildcards are (Hwang, 2022):

- + replaces a single topic level. For example, home/+/temp corresponds to both home/living room/temp and home/kitchen/temp.
- # replaces zero or more levels at the end of the topic. For example, home/# corresponds to all posts beginning with home/.

The use of wildcards, however, may have security implications. Here are some considerations:

- **Over-subscription:** Improper use of wildcards could result in a client receiving an excessive volume of messages, including unnecessary or unwanted messages. This can expose the client to sensitive information not intended for him or overload the client and the network.

- **Ineffective filtering:** Too general a subscription may not filter messages effectively, leading to the receipt of irrelevant or potentially dangerous data.
- **Unwanted access:** If security policies were not well configured, a client could use wildcards to access topics that should be confidential or protected, thereby gaining access to sensitive information.

MQTT provides three levels of Quality of Service (QoS) to ensure the reliability of message delivery between client and broker. Here is a summary of the three levels (Atmoko, 2017):

- Maximum one time (QoS 0): The message is sent only once with no guarantee that the recipient will receive it.
- At least once (QoS 1): Ensures that the message is delivered to the recipient at least once.
- Exactly once (QoS 2): This is the most secure level and ensures that each message is received exactly once by the recipient.

2.2.2 Importance of MQTT in IoT systems

Its importance in IoT systems is deeply rooted in its ability to facilitate reliable and efficient communication between devices, often called "things," and the MQTT server or broker that coordinates message distribution. IoT devices, such as sensors or actuators, publish messages on specific topics, which are managed by an MQTT broker. Other devices or services can subscribe to one or more of these topics to receive the relevant messages. This mode of operation improves the scalability of the system and facilitates the integration of new devices. Security is another key issue addressed by MQTT, which offers mechanisms such as SSL/TLS encryption to protect data transmitted between devices and the broker. This is especially important in the IoT era, where data security and protection from manipulation or unauthorized access are critical. MQTT's efficiency in handling communication between a wide range of devices makes it ideal for diverse IoT applications, ranging from home automation to industrial monitoring, from smart agriculture to smart city solutions. For example, in an environmental monitoring application, sensors located at various locations can periodically publish temperature, humidity or air quality data on specific topics managed by an MQTT broker. An analytics application centralized can subscribe to these topics to collect and process data, generating alarms or automatic actions based on the data received. The MQTT protocol is designed to run on top of TCP/IP, providing an efficient and lightweight method for message transmission. One of the distinguishing features of MQTT is its minimal data packet size, with an overhead starting at just 2 bytes. This reduced packet size results in lower bandwidth consumption and, consequently, optimized power consumption, allowing them to remain in "sleep" mode for extended periods. This approach is crucial for IoT scenarios, where devices may be battery-powered and deployed in remote or hard-to-access environments, making their ability to operate for long periods without maintenance essential (Atmoko, 2017) (Mishra, 2020).

2.2.3 MQTT protocol security issues

Security in MQTT can be compromised due to failure to implement robust security measures, leaving devices and data vulnerable to various types of attacks. One of the main security problems with MQTT is insufficient authentication. By default, MQTT does not require strong authentication, allowing anyone who knows the broker's address to post or subscribe to topics, exposing systems to risk of eavesdropping or improper message manipulation. This can be particularly damaging in critical applications, where sensitive data or control commands could be intercepted or altered by malicious actors.

Running heavy encryption algorithms on microcontrollers with limited computational and memory resources may prove impractical or inconvenient, as IoT devices have little computational capacity and restricted memory, making it difficult to implement advanced cryptographic techniques without compromising performance or functionality.

Encryption, although essential for security, requires a balance between the desired level of security and the capabilities of the device. Therefore, adopting lightweight encryption methods or optimizing available resources becomes crucial to maintain both security and operational efficiency of microcontrollers in resource-constrained contexts.

In the case of lack of encryption of data transmitted through MQTT makes the protocol vulnerable to eavesdropping attacks. Without encryption, data sent between devices and the broker can be easily read by anyone on the same network, compromising the confidentiality of the information exchanged. Another significant problem is inadequate topic management (Andy, 2017).

In MQTT, topics are used to filter and direct messages to the appropriate subscribers. However, without appropriate permission management, it is possible for unauthorized users to subscribe to sensitive topics, thereby gaining access to data that should not be available to them.

The use of wildcards in messaging systems presents potential security and information management risks.Although wildcards offer flexibility in topic subscriptions, allowing clients to receive messages from multiple sources with a single subscription, this feature can expose systems to various dangers. Subscribing to overly broad topics can lead to the receipt of unwanted or sensitive data, increasing the risk of system overload or privacy violations. In addition, misuse of wildcards could facilitate targeted network attacks, such as eavesdropping or message flooding. It is critical, therefore, to adopt stringent security policies, limit the use of wildcards to well-defined use cases, and closely monitor topic access to mitigate these risks and ensure the security of transferred data. This risk is amplified by the tendency of some developers to use predictable naming schemes for topics, making it easy for attackers to guess topic names to subscribe to. Resilience to denial of service (DoS) attacks is another issue. Because MQTT relies on a persistent connection between the device and the broker, an attacker could easily overload the system with malicious requests, making the service unavailable to legitimate users. This type of attack can have devastating consequences, especially in critical systems where continuous data availability is essential (Andy, 2017).

From a local perspective, an attacker can intercept and alter the data exchanged in the network, thereby compromising the privacy, data integrity, and authentication mechanisms of the MQTT protocol. It should be noted that the use of unconventional ports does not increase the security of MQTT. To mitigate such vulnerabilities, it is critical to implement robust security mechanisms for MQTT. The use of TLS is an effective solution, especially for IoT devices that are not resource-constrained. In addition, Singh et al. suggested a security approach based on the ECC encryption algorithm, which is advantageous because of its lower resource demand than TLS and focused on protecting data confidentiality (Andy, 2017).

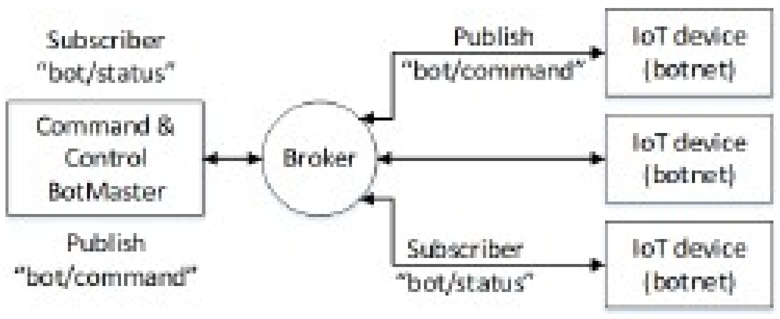


Figure 5. Command and control scenario of botnet using MQTT

Similarly, Mektoubi et al., (2016) conducted a comparative analysis between RSA and ECC, showing that the latter provides better data protection and ensures reliability in non-repudiation. For devices with limited capabilities, Niruntasukrat et al. developed a security mechanism focused on device authentication and authorization, while Katsikeas proposed the use of AES encryption, aimed at safeguarding the confidentiality and authenticity of messages. Despite these advances, the security of the MQTT protocol, particularly for low-resource devices, needs further research and development. Currently, each proposed solution focuses on specific aspects without offering a holistic approach to security.

CHAPTER III

BINARY INSTRUMENTATION

3.1 Introduction and definition of binary instrumentation

Binary instrumentation represents a fundamental concept in the field of software engineering and computer security, which refers to the technique of analyzing and modifying executable programs without accessing their original source code. This practice allows developers and analysts to insert additional code or modify the existing behavior of an application in order to monitor or change its execution in real time. The main goal of binary instrumentation is to provide a deep understanding of software behavior in various execution scenarios, improving software security, performance and quality. Binary instrumentation can be achieved in two ways (Du, 2012):

1. **Dynamic binary instrumentation (DBI):** just-in-time instrumentation of running code. This allows the behavior of the program to be analyzed and modified while it is running.
2. **Static binary instrumentation (SBI):** additional code is inserted directly into the executable file, allowing the behavior of the program to be monitored.

Static binary instrumentation consists of analyzing the program before runtime, first disassembling the code and then detecting points in the program that require instrumentation by going to permanently modify the binary. These points (called instrumentation points) will be used as locations for inserting additional code (called instrumentation code) (Figure 6) (Zhang, 2014).

On the other hand, dynamic binary instrumentation relies on an external monitor that monitors the program at the time of execution. The monitor checks one instruction at a time before it is executed and inserts instrumentation code on the fly. The main advantages of DBI are:

a) overhead in terms of memory costs is avoided, since the original executable is kept unchanged;

b) instrumentation is performed at t h e time of execution, without the need for an expensive offline process.

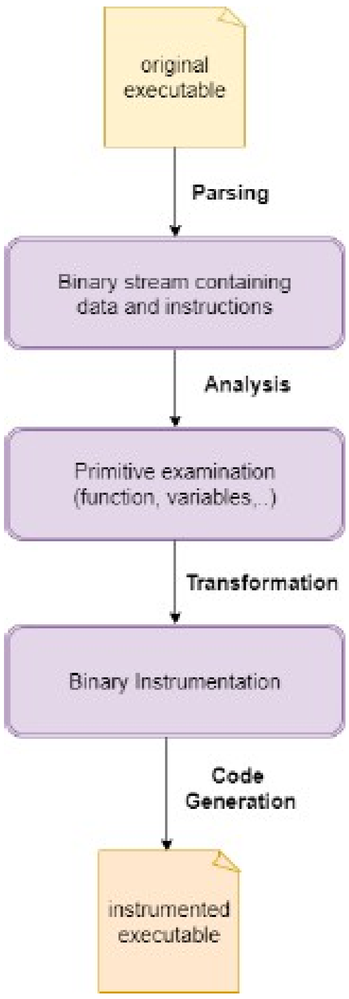


Figure 6. Static binary instrumentation process.

In addition, the real-time approach taken by DBI does not have to worry about code relocation, which must be addressed by SBI during the transformation phase. In contrast, DBI has its disadvantages in the overhead introduced in terms of execution time, which can significantly affect performance. This is a particularly critical feature in real-time applications in embedded systems.

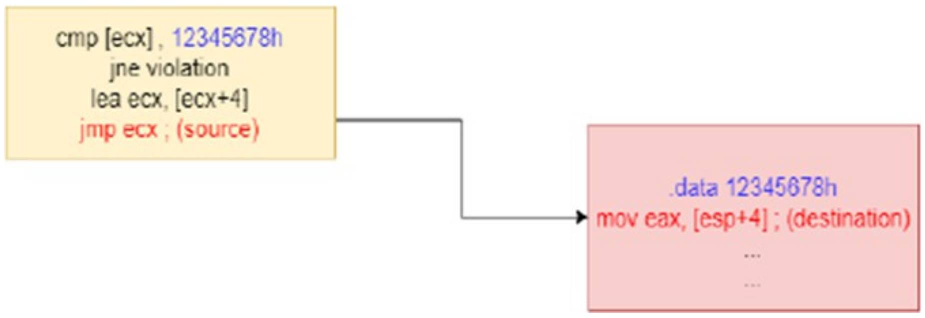
SBI introduces an overhead in terms of performance, but it is only given by the instrumentation instructions if they are entered, i.e., minimal compared to what happens with DBI, where a per-symbol execution takes place. With both strategies, the instrumentation process follows the following steps (Wenzl, 2019):

1. Parsing: the purpose of this step is to obtain the raw instruction stream from the executable and then send it as input to a disassembler. In addition, the contents of global variables and the data section are collected for further analysis;
2. Analysis: at this stage the structure of the program is recovered. The instruction flow is disassembled and the disassembled instructions are grouped into functions. In addition, conditional instructions are detected and a control flow graph (CFG) is produced;
3. Transformation: once the CFG is available, the instrumentation points in the code are collected and integrated with the instrumentation code;
4. Code generation: this is the last step in the process, where changes are integrated into the program to produce a final executable.

3.2 Applications in the area of monitoring

One of the primary applications of binary instrumentation is in software performance monitoring. Through this technique, developers can insert performance counters or timers at specific points in a program to collect detailed execution information, such as the time taken by particular functions or the number of times a certain block of code is executed. This data can then be analyzed to identify performance bottlenecks and guide code optimization. In the area of cybersecurity, binary instrumentation is useful to developers because it allows them to inject checkpoints or triggers into the code at strategic points that can reveal or block suspicious or unauthorized behavior. Through binary instrumentation, it is possible to monitor the program execution flow and intercept exploit attempts, unauthorized access, or other anomalies, contributing significantly to the strengthening of software protection measures. In addition, binary instrumentation can be used to implement additional security checks that verify the integrity of running data or to identify exploit attempts resulting from buffer overflows, code injections, or other attack vectors. By integrating these checks directly into the binary code, developers have the ability to protect applications even in the absence of the original source code or in contexts where the source cannot be modified. This approach makes binary instrumentation an extremely powerful tool for improving not only the performance but also the security of software. The integrity solutions of the software-based control flow are based on control flow monitors implemented exclusively using software techniques. In other words, checks are performed by additional code, executed in parallel through another process or within the program itself, through binary instrumentation (Zhang, 2014).

In Abadi's (2005) original article describing CFI, the authors propose a label-based instrumentation to make the software check the validity of the control flow by itself. The approach is based on storing unique identifiers (labels) at the beginning of each basic block, where a basic block in a CFG corresponds to a set of instructions without jump instructions in between. The instrumentation code is inserted before the indirect branch and checks whether the target basic block ID matches the expected one retrieved from the CFG analysis. In the case of malicious attempts to check the execution flow, the attempt is detected and the integrity of the execution flow is ensured. An example of how the mechanism works is shown in Figure 7 (Abadi, 2005).



```
cmp [ecx], 12345678h
jne violation
lea ecx, [ecx+4]
jmp ecx ; (source)
```

```
.data 12345678h
mov eax, [esp+4] ; (destination)
...
...
```

Figure 7. Label-based instrumentation code.

In this case, the jump destination was labeled with the ID 12345678. Before executing JMP ECX, the destination label is checked for attempts to tamper with the control flow. In the example if the check fails, execution is redirected to the penetration() function, which will stop execution; otherwise, ECX is incremented and execution is successfully redirected to the correct destination (Abadi, 2005).

Another significant application of binary instrumentation is in the field of reverse engineering and malware analysis. Malware reverse engineering performs dynamic code analysis to inspect a program during execution. This typically involves the use of a debugger to monitor a suspicious process. A complementary approach is to query a running process using DBI frameworks. While a debugger allows you to connect to a process, DBI techniques allow you to insert and execute code within a process to examine its internal aspects. Popular DBI frameworks include Pin, DynamoRIO and Intel's Frida. These frameworks are often used to evaluate proprietary programs and assess program performance, but they can also be applied to accelerate malware analysis. They allow analysts to hook functions to observe API calls, evaluate their inputs and outputs, and even modify instructions and data during execution. DBI frameworks target both desktop and mobile operating systems (e.g., Windows, macOS, GNU/Linux, iOS, Android™, and QNX) and provide well-documented APIs to facilitate tool development. Security analysts often use this technique to insert trace or break points into the code of suspicious programs without altering their observable behavior. This allows malware execution to be monitored in controlled environments, analyzing its interactions with the operating system and network to better understand its functionality and develop effective countermeasures (Blogs, 2021).

In addition, binary instrumentation finds use in software testing and verification, allowing developers to automatically insert assertions and validity checks into binary code to detect programming, memory management, or other types of bugs during program execution. This approach can significantly increase test coverage by providing more extensive verification than traditional methods based on source code alone. Overall, binary instrumentation offers an extremely versatile range of applications from improving software performance and security to facilitating error detection and analysis. Its ability to modify and monitor programs at the binary level opens up new possibilities in the field of software engineering, making it a valuable technique for developers, security analysts, and researchers (Brignon, 2019).

3.3 Tools and technologies for binary instrumentation

Several binary instrumentation tools have been presented in the literature, based on the advantages and disadvantages of the instrumentation technique. In the present section

some of the most common tools that follow SBI or DBI will be briefly presented.

- PEBIL (PMaC's Efficient Binary Instrumentation Toolkit for Linux) is a binary instrumentation toolkit that enables the creation of instrumentation tools by producing efficient instrumented executables, taking advantage of the static approach. PEBIL instruments the executable statically by inserting a jump instruction at the instrumentation point that will redirect execution to the instrumentation code. The instrumentation code saves the program state and will execute the actions required by the instrumentation. To support ISAs with variable-length instructions, PEBIL takes care of repositioning and transforming the code for each function in order to make the instrumentation code reachable from the instrumentation points. PEBIL supports only the x86 architecture (Laurenzano, 2010).
- PIN: as a dynamic binary instrumentation framework developed by Intel. Pin aims to provide a platform for creating program analysis tools, called pintools. A pintool consists of three types of routines:
    - instrumentation routines: routines that check application instructions and insert calls to analysis routines;
    - Analysis routines: invoked when the program executes an instrumented instruction;
    - Callback routines: called back when an event occurs;

  The C/C++ APIs provided by PIN can be leveraged to create pintools. PIN will use a Just-In-Time (JIT) compiler to apply the instrumentation at runtime to the application. In addition, PIN supports the probe mode option, which provides higher performance but has a limited set of available callbacks that limit the capabilities of the pintool. PIN is provided for Windows and Linux platforms, but it only works for programs compiled for the Intel architecture, making it unsuitable for embedded systems based on ARM architectures (Bach, 2010).
- Dynist is a binary instrumentation and analysis framework that leverages both static and dynamic approaches to code instrumentation. Dynist provides a representation of the program in terms of higher-level structures, such as basic function blocks, and the user exploits this representation to instrument the code anywhere in the binary. Dynist emphasizes instrumentation at any point in time by providing the ability to statically instrument the code (binary modified so

permanent) or to instrument it at the time of performance (dynamic instrumentation). The user can change or remove instrumentation at any time. In addition, the analysis techniques performed allow new instructions to be inserted without affecting the uninstrumented code, which can run at native speed, so as to minimize overhead in performance (Bernat, 2011).

3.4 Falco: Overview and Applications in IoT Security

In a digital age of rapidly evolving cyber threats, the importance of sophisticated security tools emerges, including Falco, an open source initiative originally developed by Sysdig, and now part of the prestigious Cloud Native Computing Foundation (CNCF). Its essence, deeply rooted in cloud-native runtime protection, manifests itself through its unique ability to scrutinize, with a watchful eye, out-of-the-ordinary behavior and potential pitfalls that threaten the integrity of systems, with a particular focus on Linux environments (Falco, 2024).

Falco works by intercepting and analyzing system calls through a technique known as "system call tracing," and executing a set of rules that allow anomaly detection (Figure 8). Falco simplifies the monitoring of Linux kernel system calls and enriches these events with information from Kubernetes and the rest of the cloud-native stack. Falco has a rich set of ready-to-use security rules created specifically for Kubernetes, Linux, and the cloud. Since its inception, Falco has been downloaded more than 100 million times, growing more than 480% in the past two years.

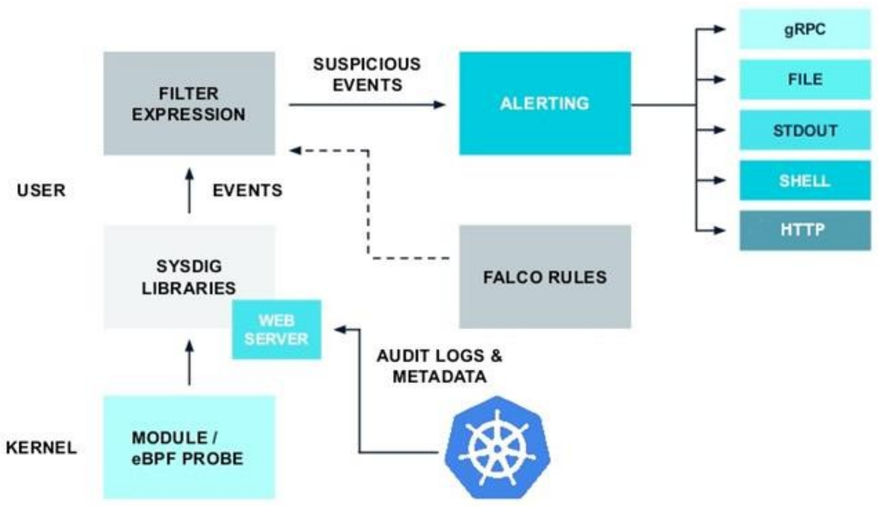


Figure 8. Architecture of Falco

3.4.1 Description of Falco

Falco is a runtime security tool designed to monitor and protect Linux operating systems from anomalous behavior and potential security threats in real time, this makes it in effect a dynamic binary instrumentation (BDI) tool. Falco is widely used in production environments by various organizations. For example, GitLab, a well-known software lifecycle management platform, uses Falco to detect suspicious activity and ensure the security of their infrastructure and services; similarly, Shopify, a leading e-commerce platform, relies on Falco to protect their customers' sensitive data and prevent possible security breaches. Its main functionality is based on observing system events, such as system calls (syscalls), and integrating metadata from the container runtime and Kubernetes (Falco, 2024).

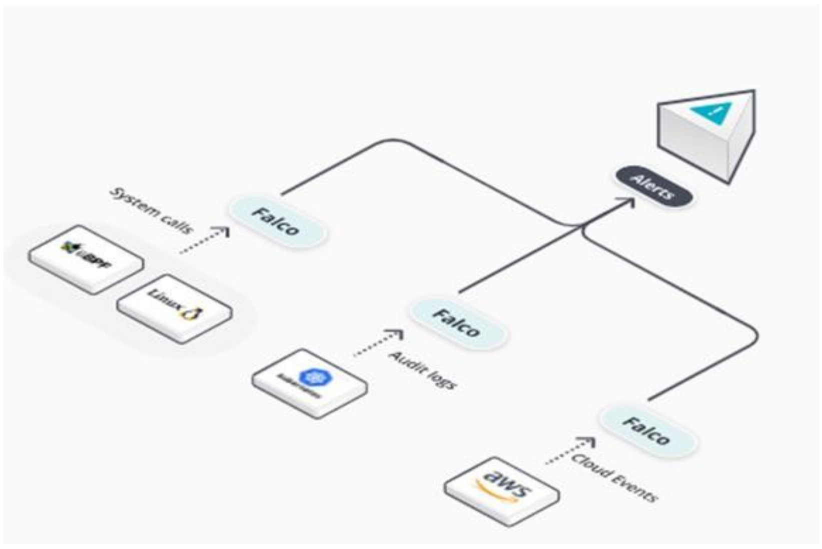


Figure 9. Scope of operation of Falco

Falco operates in both kernel and user space. In kernel space, Linux system calls (syscalls) are collected by a driver, such as the Falco kernel module or the Falco eBPF probe. Next, the system calls are placed in a ring buffer from which they are moved into user space for processing. Events are filtered using a rule engine with a set of Falco rules. Falco comes with a default set of rules, but operators can modify or disable these rules and add their own. If Falco detects suspicious events, they are forwarded to various endpoints.

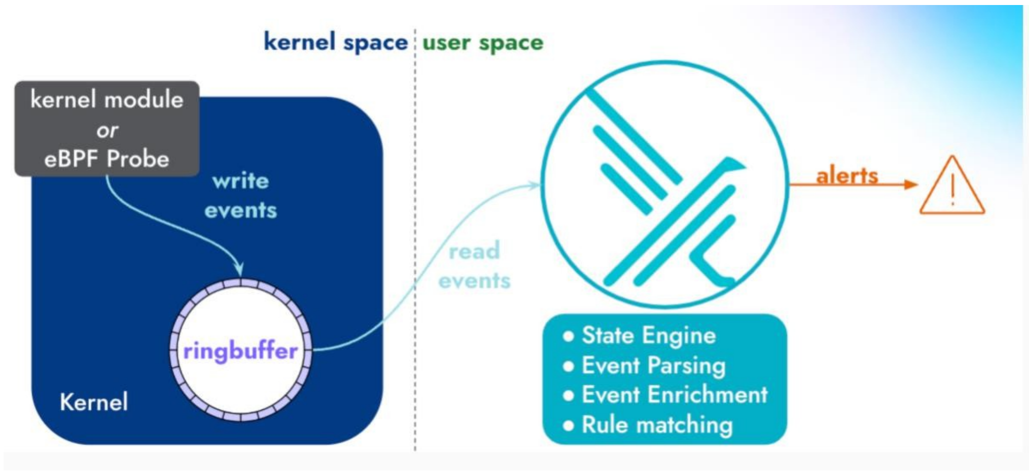


Figure 10. Falco's call handling method.

3.4.2 Key features and use cases of Falco

The main functionalities of Falco can be summarized as follows (Falco, 2024):

- Real-time monitoring: Falco monitors system calls in the Linux kernel in real time, providing an in-depth view of system activity. In the context of cybersecurity and system monitoring, tools such as Falco, developed by Sysdig, are designed to detect anomalous or potentially malicious behavior within operating systems, primarily through the observation of system calls (syscalls). System calls are critical to the interaction between running software and the operating system core, allowing low-level operations such as reading or writing files, memory management, and network communication.
- Customizable rules: Users can define specific rules that describe undesirable or potentially dangerous behavior, allowing Falco to generate alerts when these conditions are met.
- Extensibility via plugins: Falco's architecture supports plugins that extend its monitoring capabilities beyond syscalls, allowing it to include additional event sources such as Kubernetes audit logs, AWS CloudTrail events, and more. Falco offers an extensible architecture that allows a wide range of events to be monitored, not limited solely to system calls (syscalls). This extensibility is made

possible through the use of plugins and the ability to integrate specific "probes" into the code you wish to monitor.

- Integration with the Cloud-Native ecosystem: Falco integrates closely with cloud-native technologies such as Kubernetes, offering monitoring and alerting capabilities specific to these platforms.
- Configurable outputs and alerts: Falco-generated alerts can be sent to various recipients and formats, including standard logs, syslogs, HTTP[S] endpoints, and gRPC APIs, facilitating integration with event management and incident response systems.

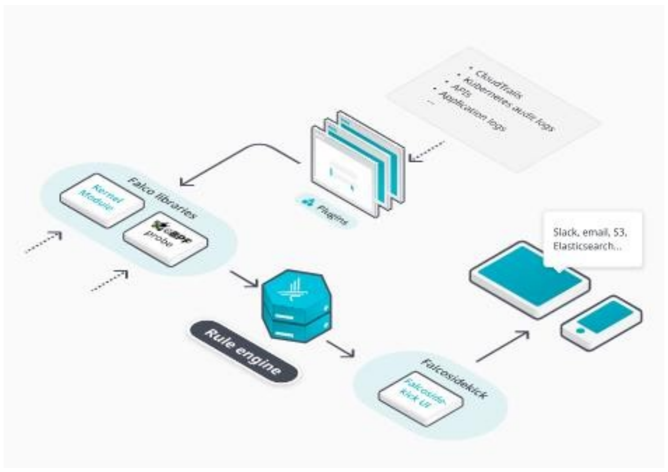


Figure 11. How Falco works

Observing system calls is critical to performance, and there are two ways Falco achieves this: an eBPF probe or a kernel module. eBPF is a revolutionary technology that allows us to run sandbox programs within an operating system. eBPF scripts are flexible, secure and run extremely fast, making them perfect for acquiring runtime security. This makes it ideal for instrumentation system calls for Falco. Before eBPF emerged, kernel modules were the norm for extending functionality in the Linux kernel. They run in privileged mode and are written in C, making them efficient and an excellent option for performance-critical jobs. Falco offers a kernel module for situations where eBPF is not the best solution (Falco, 2024).

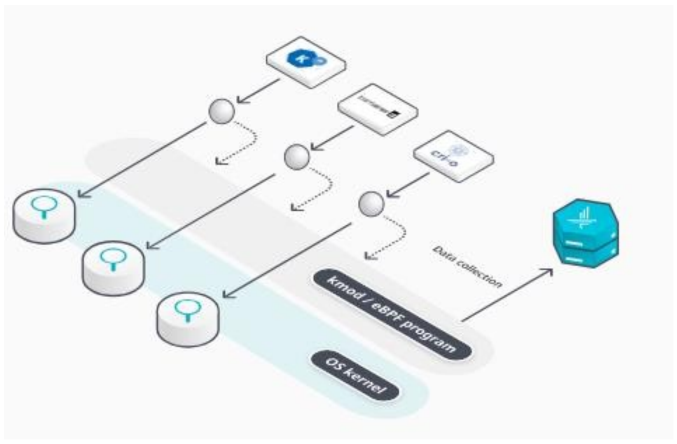


Figure 12. Call instrumentation system

Unlike kernel modules, which require kernel-level code to be compiled and loaded directly into the kernel, eBPF runs in an isolated space within the kernel, providing a secure and controlled execution environment for monitoring and data manipulation code. One of the main advantages of eBPF is security. Because eBPF code runs in a controlled virtual environment with limited access to kernel resources, the risk of system crashes or unwanted behavior is significantly reduced compared to kernel modules. In addition, eBPF imposes strict code checking before execution, ensuring that only verified and safe code can be executed. From an efficiency point of view, eBPF is designed to be extremely performant. eBPF code is compiled into bytecode and then dynamically optimized by the kernel's Just-In-Time compiler, resulting in very fast executions with low overhead. This makes it ideal for real-time monitoring and performance analysis without significantly impacting system performance. Flexibility is another key advantage of eBPF. It allows developers to write programs that can run at various hook points within the kernel, providing the ability to monitor and influence virtually any aspect of system behavior (Scholz, 2018).

With Falco and Falcosidekick, it is also possible to forward suspicious events to serverless systems to trigger actions and remediate threats. Falcosidekick is a complementary application to Falco that forwards Falco events. It allows you to distribute events to more than 50 systems, such as e-mail, chat, message queues, serverless functions, databases and more. It is easy to configure and use both locally and within Kubernetes (Falco, 2024).

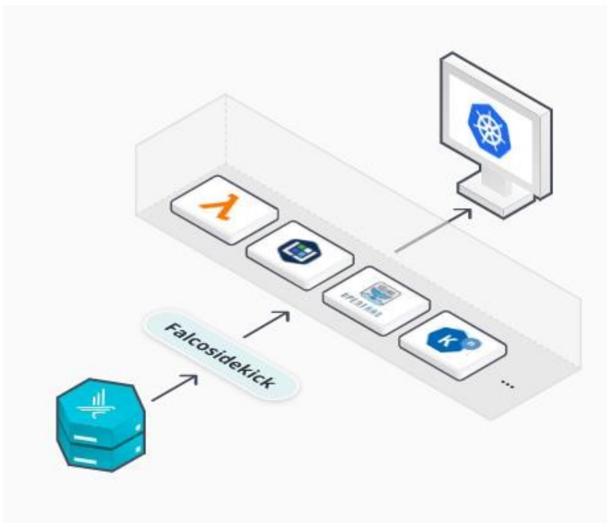


Figure 13. Threat response method

Falco, being a versatile and powerful tool for real-time security monitoring, finds application in a wide range of real-world scenarios. These use cases demonstrate how Falco can be employed to strengthen security in different environments and contexts. Here are some concrete examples of how Falco is being used for (Falco, 2024):

- Monitoring ubernetes environments: In an organization that uses Kubernetes to manage its containerized applications, Falco can be configured to monitor and alert on suspicious events or events that do not comply with corporate security policies. For example, it can detect unauthorized access attempts to sensitive Kubernetes pods, unexpected changes to container configurations, or attempts to escalate privileges within the cluster. This helps security teams intervene quickly to mitigate potential threats.
- Cloud infrastructure protection: An enterprise operating in the cloud can use Falco to monitor suspicious activity in its computing instances, such as abnormal access or unauthorized changes to critical configuration files. Falco can be configured to generate real-time alerts when activities that could indicate a compromise are detected, enabling security teams to investigate and respond promptly.
- Compliance and auditing in regulated environments: In heavily regulated industries, such as financial, health care, or public, Falco can be used to ensure that system activities comply with regulations on the

privacy and data protection. By tracking access to sensitive data and recording system activity, Falco provides an effective mechanism for auditing and demonstrating compliance with regulations, such as GDPR, HIPAA, or PCI-DSS.

- Real-time anomaly detection: An Internet service provider can use Falco to monitor its servers for anomalous behavior that could indicate a DDoS attack, intrusion attempts, or malware. By configuring specific rules, Falco can identify unusual traffic patterns or suspicious system activity, allowing the security team to act quickly to mitigate the attack and maintain continuity of service.
- Integration with incident response systems: Falco can be integrated with incident response orchestration platforms (SOARs) and security event and information management systems (SIEMs) to automate the response to security alerts. In an organization with a security operations center (SOC), Falco alerts can be automatically sent to SIEM for analysis and, if necessary, trigger automated workflows in the SOAR to isolate compromised devices, block suspicious IP addresses, or perform other mitigation actions.
- IoT security and device monitoring: In an IoT context, where connected devices are often deployed at large scale and in critical environments, Falco can be used to monitor the behavior of these devices. By detecting anomalies such as unauthorized access attempts, unexpected configuration changes, or suspicious network communications, Falco helps protect the IoT ecosystem from potential vulnerabilities and attacks.

3.4.3 Falco for strengthening the security of IoT systems

In the context of IoT security, the importance of tools such as Falco becomes increasingly apparent as the number and complexity of connected devices continues to grow. These devices, often deployed on a large scale and in critical environments, can become critical targets for cyber attacks. Against this backdrop, Falco emerges as a useful tool for strengthening the security of IoT devices and infrastructure, thanks to its real-time monitoring capability. Falco offers system administrators the ability to identify suspicious activity or anomalies in real time that could indicate a security compromise.

This includes detection of privilege escalation attempts, which could indicate an attack in progress to gain deeper control over the system, or unauthorized access to critical files and directories, which could suggest attempts to steal data or manipulate the system. By implementing custom rules in Falco, operators can define what behavior to consider dangerous or unauthorized, enabling targeted and highly effective monitoring (Falco, 2024).

A key element of Falco is its ability to use custom rules, which allow users to define specific behaviors to be monitored. This flexibility in configuration rules significantly increases the granularity of monitoring, enabling detailed and targeted surveillance that is crucial in IoT environments, where each device may have different security needs. Falco is therefore useful for identifying misconfigurations, unauthorized access or execution of unusual commands, helping to protect IoT infrastructure from potential threats. This introspection capability ensures the security and integrity of the infrastructure.

Falco's ability to generate detailed and configurable alarms makes it an ideal candidate for integration with automated incident response systems. These alerts can trigger immediate corrective actions, such as isolating compromised devices to prevent further damage. This type of integration enables rapid incident response, reducing exposure time to potential threats and minimizing the impact of attacks. Data security and privacy regulations are becoming increasingly stringent, especially in critical industries such as healthcare, finance and utilities, where IoT devices are widely used. Falco's monitoring and logging capabilities help meet these compliance and auditing requirements for IoT systems by providing detailed records of system activity and security events. This not only allows demonstration of compliance with current regulations but also provides valuable forensic information in case of security incidents (Falco, 2024).

3.5 RGB565

The RGB565 format represents an efficient and popular method for encoding colors in digital devices, especially those with memory or bandwidth limitations. This format uses 16 bits to represent a single color, distributed among the red, green and blue channels, with a distribution of 5 bits for red, 6 bits for green and 5 bits for blue. The choice of this bit distribution takes into consideration the greater sensitivity of the human eye to changes in green, thus allowing more accurate color representation while maintaining a compact format (RGB565 Color Picker, 2024).

Among the advantages we can mention (RGB565 Color Picker, 2024):

- Memory efficiency: Using only 16 bits per color, the RGB565 format significantly reduces memory usage, compared to the 24 bits required by the more widely used RGB888 format. This translates into a 33% space saving, critical in resource-constrained devices.
- Improved performance: Reducing the amount of data to be processed can lead to improved performance in real-time applications, such as video games or rendering animated graphics, where processing speed is essential.
- Reduced power consumption: In portable devices, where power conservation is critical to extend battery life, the use of a more efficient color format can help reduce overall power consumption.

For disadvantages, however (RGB565 Color Picker, 2024):

- Limited color depth: With only 65,536 colors available, the RGB565 format can be limited for applications that require a wide color gamut or need to handle delicate hues. This can lead to undesirable effects such as banding, where color transitions are no longer smooth.
- Absence of transparency: Unlike other formats that include an alpha channel to handle transparency, the RGB565 format does not natively support this feature, limiting its use in contexts where transparency is not necessary.

Conversion from RGB888 to RGB565 and vice versa is a common operation in many graphics applications. This process involves reducing or extending color values to fit the new format (RGB565 Color Picker, 2024).

CHAPTER IV

RESEARCH CONTRIBUTION

The contribution of this research focuses on using techniques, including binary instrumentation, for runtime monitoring of IoT devices and investigating the corresponding overhead introduced.

Through a quantitative assessment, the impact and specific metrics of overhead on performance and security are examined. The research includes tests and benchmarks to quantify overhead in various scenarios.

Practical guidance for integrating monitoring systems is also offered, The work contributes to the understanding of the interplay between security and performance in IoT systems.

4.1 Analysis of actors and system architecture

The structure of the system under consideration is described with a detailed UML Deployment diagram (Figure 14) that illustrates the various components and their interactions within the network.

Specifically, the system consists of five main hardware nodes:

- **Raspberry Pi 4 (RPI4):** This node serves as the hub of the system, hosting a number of essential features:
    - FALCO's implementation, explored in depth in the previous chapter. Its output is subsequently routed to the Adapter through a pipe mechanism.
    - The Adapter allows FALCO's output data to be interfaced with the MONITOR component, converting it into a more high-level, easily usable form for monitoring.
    - The subscriber 'clientSubSenseHat.js' is responsible for lighting the LEDs of the Sense-Hat with the color it receives through the MQTT protocol from the client 'clientPub.js'.
- The **Sense-Hat** is an expansion board designed to interact directly with the Raspberry Pi through the use of GPIO pins and the I2C protocol.

  Although equipped with a variety of sensors, in our experiment the focus is solely on the 8x8 LED array of the Sense-Hat.

In Linux, this array is treated as a system object, which allows the operating system to interact with it through a file-based interface.

The specific use of the syscall 'pwrite' allows writing directly to the file associated with the Sense-Hat. This method facilitates the monitoring of commands sent to the array via writes.

- **HiveMQ server**, that entity operates as an MQTT broker.
- **Device 1** which hosts:
  - The 'clientPub.js' client, i.e., an MQTT publisher in charge of creating and sending MQTT messages representing commands to be executed on the LED array of the Sense-Hat.
- **Device 2,** which hosts:
  - Monitor artifact, developed in Prolog, this component receives messages from the Raspberry's "adapterLight" entity via the WebSocket protocol and verifies their correctness.

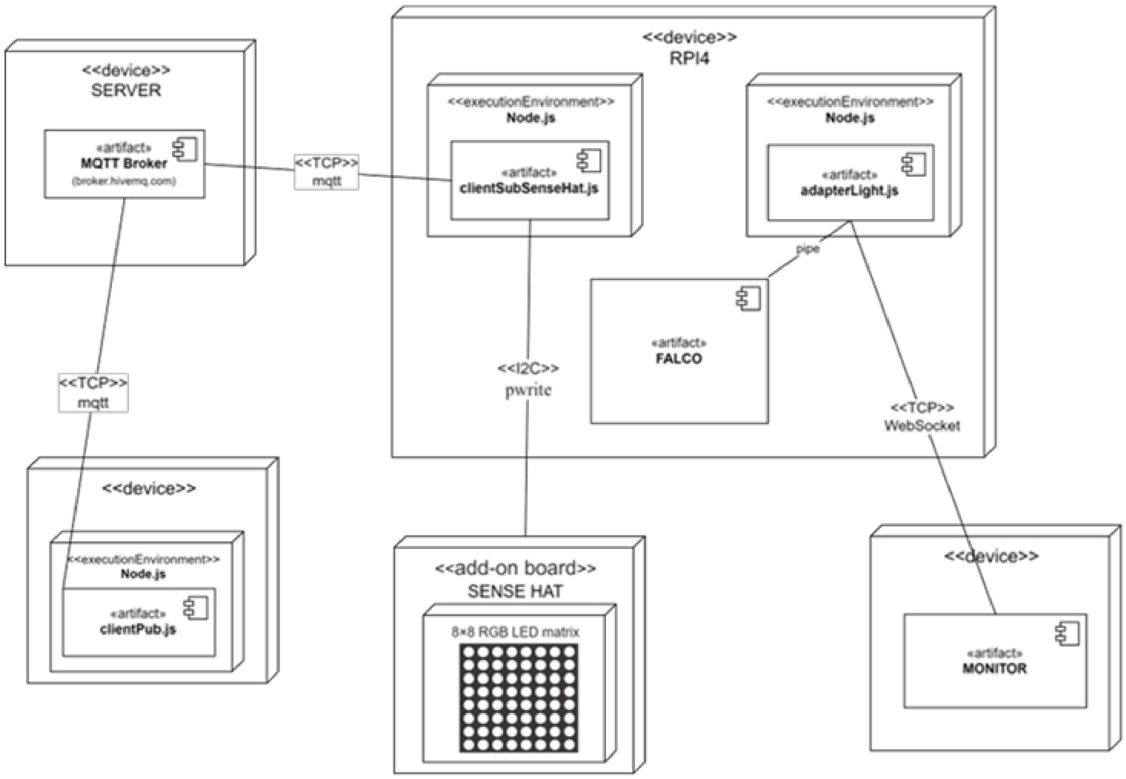


Figure 14. Deployment diagram of the system under study

In our test scenario, the client 'clientPub.js' plays a crucial role by sending commands (simple MQTT messages) where the topic corresponds to the order to be executed while the payload contains the color in RGB888 format. These commands are intended for the 'clientSubSenseHat.js,' which is in charge of interpreting them to check the

LED matrix of the Sense-Hat. However, writing the values to the Sense-Hat is done in hexadecimal format and thus in RGB565, thus introducing additional complexity into the process.

The primary objective of this system is to ensure that changes to the LED matrix of the Sense-Hat exactly match the commands received. This verification, carried out by the MONITOR through cross-analysis, is crucial to confirm that the system is functioning correctly, ensuring that each instruction sent is interpreted and displayed faithfully on the device.

4.2 Falco User's Guide

4.2.1 Installation

In the specific case of our study, the focus is on using Falco on a Raspberry Pi 4 node running Ubuntu 20.04, which represents a possible implementation in IoT contexts. As we explained in the previous chapter, the importance of dynamic instrumentation lies in its ability to provide a real-time view of system-level activities, enabling detection and response to potential security threats.

Falco, being a runtime security tool, allows monitoring and logging system calls in real time, in this specific case its main role will be to filter and log only TCP syscalls coming from the MQTT broker and writes executed on the Sense-Hat.

Installing Falco on Debian-like distributions, such as Ubuntu, requires a number of key steps to ensure proper and secure integration into the system. Initially, it is crucial to entrust the falcosecurity GPG key, a key step in ensuring the authenticity of the installed software. This is achieved by downloading and installing the GPG key, which certifies that the packages received come from a trusted source.

For our case study, we chose kmod because of its high level of efficiency and accuracy in monitoring kernel-level activities, as well as its ability to provide detailed information in real time.

However, it is critical to consider the potential disadvantages associated with this approach. The close association of kmod with the host kernel means that any changes in kernel versions, architecture or operating systems can introduce significant complexities. This interdependence requires that, with each kernel update, Falco's kmod module may need recompilation or adjustments to maintain compatibility and ensure optimal performance. Furthermore, because kmod operates at such a deep level within the system, any malfunctions or bugs in the module can directly affect the stability and security of the host system. Therefore, while kmod offers indisputable advantages in terms of monitoring capabilities, it also requires careful and proactive management to ensure that system evolutions do not compromise Falco's functionality (eBPF vs. KMod Performance, 2024).

A typical installation on Ubuntu requires the installation of dependencies needed to build the kernel module, a critical aspect of using Falco's kmod mode.

4.2.2 Startup using custom rules

The configuration of Falco represents a critical operation that significantly affects the behavior of the software and its interaction with the host system. After installation, it is critical to customize the 'falco.yaml' file, as it allows systems administrators to calibrate monitoring according to the characteristics of the environment in which the software operates.

During initialization, it is possible to specify the configuration file to be used, the rules to be applied, and the mode of Falco execution through the use of defined parameters. For example, the command we used 'sudo falco -c falco/falco.yaml -r falco/log_server.yaml -r falco/sense-hat.yaml -b -S 400' is an example of a custom startup, where we outline the specific configuration files and rules.

Specifically, the rule contained in 'sense-hat.yaml' focuses on monitoring 'pwrite' write operations to the '/dev/fb0' file, which is directly connected to the Sense-Hat display, playing an essential role in identifying interactions with the device. In parallel, the rule defined in 'log_server.yaml' monitors the TCP syscalls associated with the clientSubSenseHat.js script, the specified argument '-S 400' assumes great importance because '-S' allows us to increase the maximum size (in bytes) of the 'env.buffer' field of FALCO, in our case where multiple MQTT packets can be found in a single TCP syscall it is essential to increase the maximum size of the buffer (the standard one is only 80 bytes).

The implementation of such rules in our IoT system is not random, but rather results from a strategic selection aimed at verifying the accuracy with which commands, received through the MQTT protocol, are executed on the Sense-Hat display, thus assuring us that the system is functioning properly.

4.3 Developed software

4.3.1 Parse-mqttv5-packet library

The parse-mqttv5-packet library is a tool designed specifically for analyzing MQTT packets; it plays a key role in interpreting raw data, transforming them into structured objects that can be easily manipulated, thus facilitating their analysis.

The need for such a library arose from the observation that existing solutions were not optimally suited to the specific needs of our project. Therefore, the decision was made to develop a dedicated library that could offer accurate and flexible parsing of MQTT packets, providing developers with an intuitive and powerful interface for working with the protocol's messages.

The Parser class is designed to be extremely versatile; it can be configured at creation time by providing its constructor with the MQTT protocol version of the packets to be parsed. Once instantiated with the specified version, the parser is ready to perform its task.

The operational core of Parser lies in its parse method; the method takes as input a byte buffer, which may contain one or more consecutive MQTT packets, recognizes the type of each packet with specific contents, and decrypts the "remaining length" field (Figure 33).

The "remaining length" field in MQTT packets is crucial because it indicates how many bytes to read to complete the parsing of the packet in question. The parse() method uses this information to identify the boundaries of each packet within the buffer, allowing the method to correctly position itself in the buffer to begin parsing the next packet after finishing that of the current one.

"Remaining length" encoding is implemented in this way: each byte read contributes to the total value, with the first 7 bits of each byte representing the value and the eighth bit indicating whether there are additional bytes to be read. This information is essential for the parser to determine precisely where a packet ends and begins the next, allowing accurate handling of concatenated packet flows without error or ambiguity.

Once the packet type is recognized, parse() proceeds with the creation of an object of type Packet, populating its attributes with the specific packet data extracted from the buffer.

The Packet class, is the heart of the parse-mqttv5-packet library, serves as the abstract base class for all the different types of MQTTv5 packets. It defines the common attributes and interface that each packet has, establishing a structure shared by the child classes; this inheritance hierarchy ensures modular, intuitive organization and code reuse. Each Packet child class represents a specific type of MQTT packet:

- **Connect:** Handles and represents packets of type CONNECT, containing information to establish a connection between the client and the server.
- **ConnAck**: Handles and represents CONNACK packets, which are the server's responses to CONNECT packets.
- **Publish** Deals with and represents PUBLISH packages, used to distribute messages to subscribed clients.
- **PubAck**: Handles and represents PUBACK packets, sent as a response to PUBLISH packets with QoS 1.
- **PubRec**: Handles and represents PUBREC packets, which are used in the QoS 2 flow to signal the receipt of a PUBLISH packet.
- **PubRel**: Handles and represents PUBREL packets, which are part of the delivery assurance process for packets with QoS 2.
- **PubComp**: Handles and represents PUBCOMP packets, which complete the exchange cycle for PUBLISH packets with QoS 2.
- Subscribe: Deals with and represents SUBSCRIBE packages, used by clients to request subscription to specific topics.
- **SubAck**: Handles and represents SUBACK packets, sent by the server in response to SUBSCRIBE packets.
- **UnSubscribe**: Deals with and represents UNSUBSCRIBE packets, used by clients to unsubscribe.
- **UnSubAck**: Handles and represents UNSUBACK packets, which are the server's responses to UNSUBSCRIBEs.

Each of these classes inherits the attributes of the Packet class and introduces additional details and specific functionality dependent on the type of packet it represents.

Each child class has a constructor that is responsible for parsing the rest of the buffer that has not yet been interpreted, extracting and populating the specific fields in the package, through the definition of new properties.

This approach offers the great advantage to users of this library the realization of generic functions capable of accepting as input an instance of Packet and operating with any of its subclasses thanks to polymorphism.

In summary, this architecture not only makes the library extremely flexible but also allows polymorphic interoperability, enabling the development of generic functions that can operate on any subclass of Packet. This gives developers the ability to extend the library to support new packet types or to adapt to future versions of the MQTT protocol, while maintaining efficient and consistent handling of various types of communications.

The library is publicly accessible on GitHub, thus enabling interested developers to contribute, customize and integrate this solution into their analysis applications. For further details, exploration and contributions, you can

visit the repository parse-mqttv5-packet on GitHub .

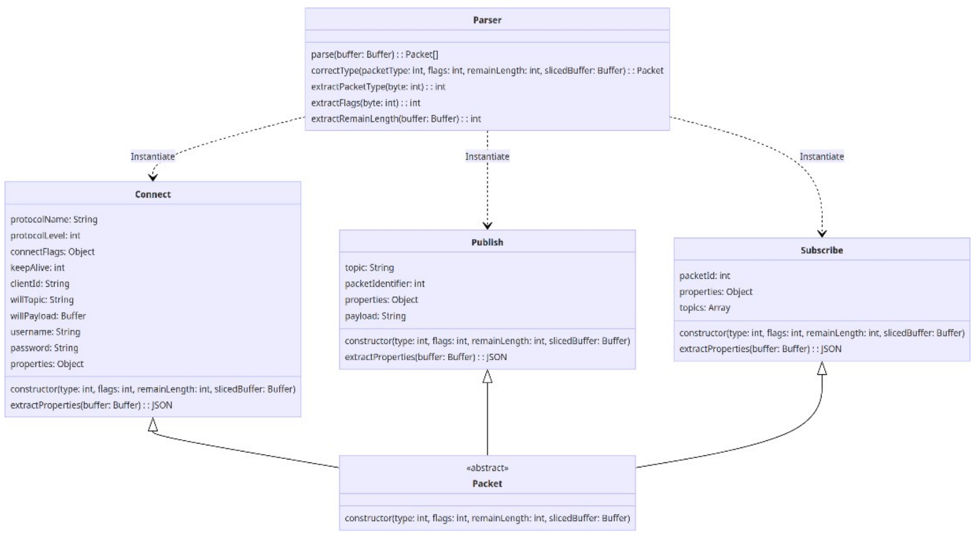

Parser
parse(buffer: Buffer) : : Packet[]
correctType(packetType: int, flags: int, remainLength: int, slicedBuffer: Buffer) : : Packet
extractPacketType(byte: int) : : int
extractFlags(byte: int) : : int
extractRemainLength(buffer: Buffer) : : int
Instantiate
Instantiate
Instantiate
Connect
protocolName: String
protocolLevel: int
connectFlags: Object
keepAlive: int
clientId: String
willTopic: String
willPayload: Buffer
username: String
password: String
properties: Object
constructor(type: int, flags: int, remainLength: int, slicedBuffer: Buffer)
extractProperties(buffer: Buffer) : : JSON
Publish
topic: String
packetIdentifier: int
properties: Object
payload: String
constructor(type: int, flags: int, remainLength: int, slicedBuffer: Buffer)
extractProperties(buffer: Buffer) : : JSON
Subscribe
packetId: int
properties: Object
topics: Array
constructor(type: int, flags: int, remainLength: int, slicedBuffer: Buffer)
extractProperties(buffer: Buffer) : : JSON
«abstract»
Packet
constructor(type: int, flags: int, remainLength: int, slicedBuffer: Buffer)

Figure 33. Class diagram parse-mqttv5-packet

4.3.2 Adapter: essential bridge between Falco and the Monitor

This section reviews the process of designing and implementing the Software Adapter, a key component designed to optimize the integration and flow of data between FALCO and the MONITOR.

The following sequence diagram (Figure 15), illustrates the architectural aspects of the system:

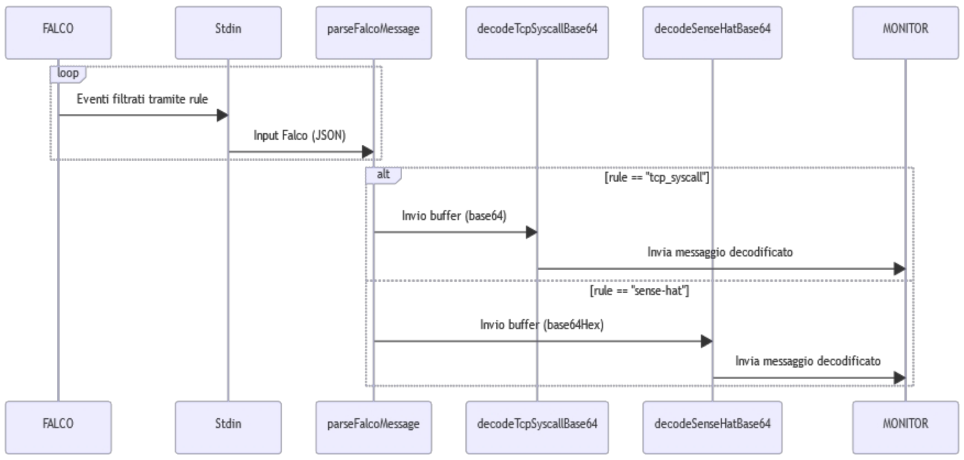


Figure 15. Sequence diagram of the Adapter (capitalized external actors).

The diagram clearly illustrates the operational flow, showing how events, initially filtered by Falco according to well-defined rules, are sent via a pipe in JSON format to the Adapter, which processes them using the parsing function called parseFalcoJSON.

A key aspect at this stage is the parsing of the 'rule' field present in the JSON data, which can be 'tcp_syscall' or 'sense-hat'; in fact, parsing is not only a data interpretation operation, but also identifies which Falco-specific rule has been triggered. This distinction is critical because it directs the event to the appropriate decoding function: decodeTcpSyscallBase64 for 'tcp_syscall' or decodeSenseHatBase64 for 'sense-hat'.

The decodeSenseHatBase64 function transforms the base64 data found in 'evt.buffer' into a hexadecimal string representing a color. This data is then converted to a standard RGB565 object via the hexToRGB565 function, which isolates the red, green and blue components.

Two versions of the Adapter have been implemented: a light version and a full version, which differ in the way they process and send data to the monitor, the main difference being precisely in the decodeTcpSyscallBase64 function and in the way the two versions transform the received RGB888 message:

The full version, in addition to analyzing MQTT messages, also modifies them, converting messages from RGB888 format to RGB565 format.

The light version, on the other hand, is limited only to parsing MQTT messages while maintaining the RGB888 format. This will result in more effort for the

MONITOR in the next step; in fact, it will have to perform additional operations independently in order to compare the color data displayed on the Sense-Hat with those received from the MQTT broker.

A major challenge in the development of the Adapter was managing the asymmetry between 'tcp_syscall' and 'sense-hat' packets; when the Adapter receives 'tcp_syscall' packets, it may face a situation where 'evt.buffer' contains multiple concatenated MQTT messages. On the other hand, on the other hand, 'sense-hat' packets do not have this complexity since they are tied to single events and never contain multiple messages in a single 'evt.buffer'.

The asymmetry thus arises from the fact that the Adapter must handle two different data streams: one (from 'tcp_syscall') potentially consisting of multiple MQTT messages and the other (from 'sense-hat'). The goal is to be able to ensure that the data is processed and forwarded to the monitor in a synchronized and accurate manner. The Adapter adopts a FIFO (First In, First Out) queue as a strategic solution to deal with the mismatch between the 'tcp_syscall' and 'sense-hat' packets, a crucial aspect for the proper functioning of the system.

The queue is used to store decoded MQTT messages from the 'evt.buffer' field in 'tcp_syscall' packets, ensuring sequentiality in their processing by queuing MQTT messages. The Adapter, therefore, does not proceed to immediately forward messages to the monitor; rather, it waits for the arrival of a 'sense-hat' packet to start processing. This synchronized approach ensures that MQTT messages are processed sequentially with 'sense-hat' events, avoiding potential mismatches in the processed data. Upon receipt of a 'sense-hat' packet, the Adapter begins forwarding messages from the FIFO queue to the monitor. This process continues until an MQTT message is identified that aligns or matches with the 'sense-hat' event in question. This method ensures that data are processed and forwarded in an orderly manner, meeting the necessary sequencing and synchronization for effective monitoring (Figure 16).

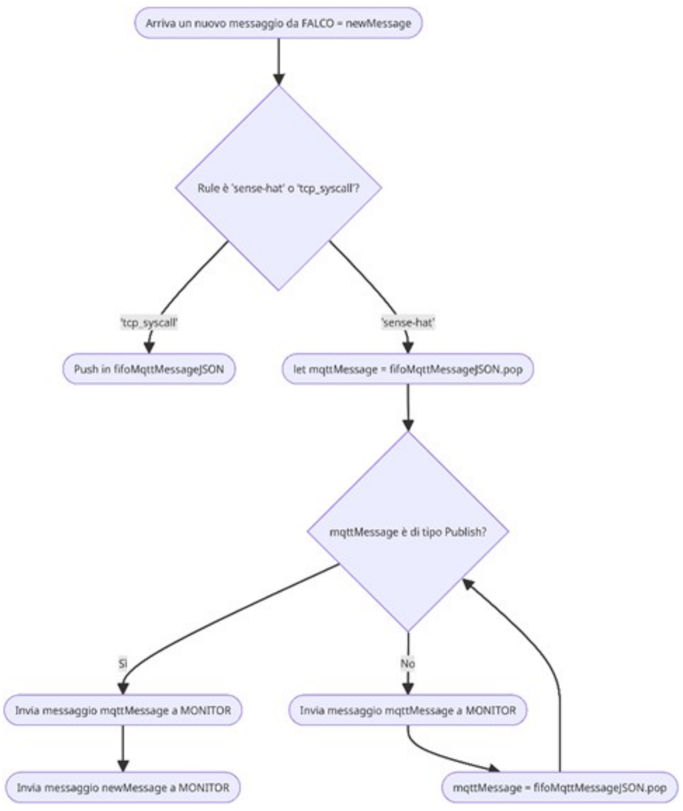

Arriva un nuovo messaggio da FALCO = newMessage
Rule è 'sense-hat' o 'tcp_syscall'?
'tcp_syscall'
'sense-hat'
Push in fifoMqttMessageJSON
let mqttMessage = fifoMqttMessageJSON.pop
mqttMessage è di tipo Publish?
Sì
No
Invia messaggio mqttMessage a MONITOR
Invia messaggio mqttMessage a MONITOR
Invia messaggio newMessage a MONITOR
mqttMessage = fifoMqttMessageJSON.pop

Figure 16. Ensuring sequentiality through FIFO: Flow Chart

It is important to note that should the monitored code command the sense-hat independently of the MQTT messages, the MONITOR would be able to detect it later.

4.3.3 Monitor with specifications in RML

The monitor, called the RMLatDIBRIS Monitor, is a crucial component in our system, charged with observing and evaluating system activities in real time. It functions as a sentinel, constantly verifying that program execution conforms to established specifications. Thanks to its monitoring activity, any abnormal or noncompliant behavior during program execution can be detected, allowing immediate (or even automatic) action to be taken to make the necessary corrections (RMLatDIBRIS Monitor, 2024).

Being implemented in Prolog, it requires the installation of SWI-Prolog version 7 or higher to function. In addition, to define the rules and conditions that events must meet for the system to be considered compliant, a specification must be provided as input.

The monitor can be used in two modes: offline and online; in the offline mode, the monitor reads a trace of sequences from a file, and based on the defined specification, generates reports on whether or not the specified conditions have been met; the offline mode is useful for analyzing past event sequences or for testing the monitor's behavior with known data.

In the online mode, however, the monitor receives events in real time via a TCP interface; this mode is ideal for systems that require real-time monitoring, allowing events to be intercepted and analyzed as they occur (RML Dibris, 2024).

To define monitoring specifications, the Runtime Monitoring Language (RML), which is specially designed for this purpose, is used. Developers can easily define rules using RML to ensure compliance of program execution traces (Ancona, 2021).

To use RML, you must use the RMLatDIBRIS Compiler, which transforms RML specifications into Prolog code.

For our system, two separate RML specifications were developed which, while similar have crucial differences in the mechanism for comparing the 'tcp_syscall' and 'sense-hat' message.

The specification 'lightAdapterRule.rml' used in conjunction with the AdapterLight and the monitor is called upon to perform additional calculations to realize a cross-check between 'sense-hat' and 'tcp_syscall'. Messages of the former type include data in RGB565 format, while the latter are in RGB888 format (Figure 17). To establish the correlation between these two types of messages, the monitor must verify that the following equations are satisfied: $((red888 / 8) - red565) \leq 1$, $((green888 / 4) - green565) \leq 1$ and $((blue888 / 8) - blue888) \leq 1$.

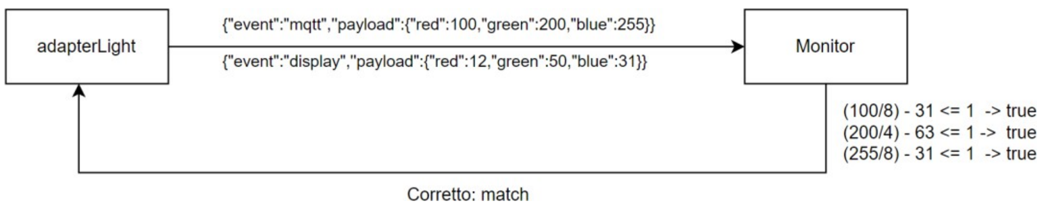

adapterLight
{"event":"mqtt","payload":{"red":100,"green":200,"blue":255}}
{"event":"display","payload":{"red":12,"green":50,"blue":31}}
Monitor
(100/8) - 31 <= 1 -> true
(200/4) - 63 <= 1 -> true
(255/8) - 31 <= 1 -> true
Corretto: match

Figure 17. Diagram on the operation of the Monitor in case of a match

In contrast, the "fullAdapterRule.rml" specification aims to facilitate the monitor's task. Since adapterFull deals with the conversion from RGB888 to RGB565, the monitor only needs to verify the equality of the two messages.

In this second scenario (Figure 18), the main goal is to increase the workload on the Adapter to lighten the workload on the monitor.

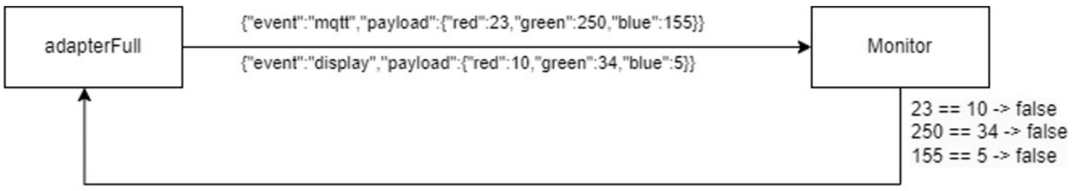

adapterFull
{"event":"mqtt","payload":{"red":23,"green":250,"blue":155}}
{"event":"display","payload":{"red":10,"green":34,"blue":5}}
Monitor
23 == 10 -> false
250 == 34 -> false
155 == 5 -> false

Figure 18. Diagram on the operation of the Monitor in case of non-match

4.4 Performance Evaluation and Impact on Overhead.

The overhead and performance impact analysis of the entire monitoring system is divided into four distinct levels, each examining the incremental impact of the tools used on overall system performance. At **Level 0**, the system is examined in its basic configuration, devoid of

any monitoring mechanism implemented. This initial condition serves as a baseline, allowing comparative evaluation of the effectiveness of subsequent monitoring tools. At this stage, system performance is analyzed in terms of resource utilization, latency, thus providing a baseline for future evaluations.

Next, **Level 1** introduces Falco. In this phase, the focus is on the impact of Falco on overall system performance, evaluating changes in resource utilization, latency, and throughput, then analyzing the influence of the event detection process.

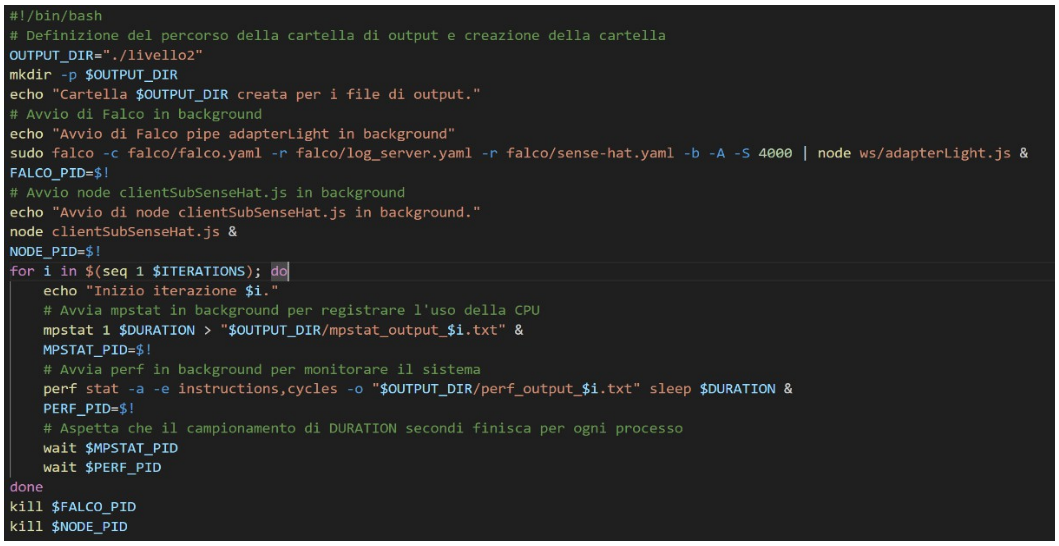
Level 2 further extends the analysis by incorporating a complete monitoring stack that includes Falco, AdapterLight, and the Monitor. The goal of this phase is to evaluate how the synergistic interaction between these components affects the overall performance of the system. Particular attention is paid to the role of the Adapter in data management, including decoding MQTT messages and synchronizing data streams, in order to determine their overall effect on performance.

Finally, **Level 3** replicates the configuration of Level 2, but replaces AdapterLight with AdapterFull, allowing a direct comparison between the two configurations to determine the impact of the full versus light version.

In the context of system performance analysis, a specially designed Bash script was used to systematically orchestrate the collection of performance data. The script was designed to monitor the impact on the entire system resulting from the execution of one or more programs. Execution of the script requires two parameters: the number of iterations (<number_iterations>) and the duration of each iteration (<iteration_duration>). After starting, the script proceeds to create a dedicated directory (./leveli) for storing the generated output files.

A monitoring strategy that is fragmented into separate sessions was adopted to allow the impact on system-wide performance to be assessed, while ensuring the ability to calculate averages for each set of iterations. This provides a balanced and normalized representation of performance, reducing the risk of external events or atypical fluctuations affecting results. Once the process 'clientSubSenseHat.js' plus the monitoring stack relative to the layer under analysis is started, two parallel monitoring processes follow in each iteration: the first uses 'mpstat' to track CPU usage, while the second employs 'perf' to collect statistics on executed instructions and processor cycles.

At the end of the iterations, the script terminates monitoring and indicates the conclusion of data collection, regardless of the level of monitoring considered. This methodological approach ensures the accuracy and reproducibility of measurements, which are crucial elements for comparative performance analysis in diverse scenarios.



```bash
#!/bin/bash
# Definizione del percorso della cartella di output e creazione della cartella
OUTPUT_DIR="./livello2"
mkdir -p $OUTPUT_DIR
echo "Cartella $OUTPUT_DIR creata per i file di output."
# Avvio di Falco in background
echo "Avvio di Falco pipe adapterLight in background"
sudo falco -c falco/falco.yaml -r falco/log_server.yaml -r falco/sense-hat.yaml -b -A -S 4000 | node ws/adapterLight.js &
FALCO_PID=$!
# Avvio node clientSubSenseHat.js in background
echo "Avvio di node clientSubSenseHat.js in background."
node clientSubSenseHat.js &
NODE_PID=$!
for i in $(seq 1 $ITERATIONS); do
    echo "Inizio iterazione $i."
    # Avvia mpstat in background per registrare l'uso della CPU
    mpstat 1 $DURATION > "$OUTPUT_DIR/mpstat_output_$i.txt" &
    MPSTAT_PID=$!
    # Avvia perf in background per monitorare il sistema
    perf stat -a -e instructions,cycles -o "$OUTPUT_DIR/perf_output_$i.txt" sleep $DURATION &
    PERF_PID=$!
    # Aspetta che il campionamento di DURATION secondi finisca per ogni processo
    wait $MPSTAT_PID
    wait $PERF_PID
done
kill $FALCO_PID
kill $NODE_PID
```

Figure 19. Bash script for layer 2 data collection.

4.4.1 First test case: Realistic scenario

In this test scenario, we examined a sample of 10 sessions, each lasting 100 seconds; the publisher (clientPub.js) sends a message every 2 seconds.

- **Level 0**: Analysis revealed that the average number of instructions processed amounted to 0.8×10^9, while the average number of processor cycles amounted to 1.1×10^9, resulting in an overall average CPU utilization of 0.21% (Figure 20).

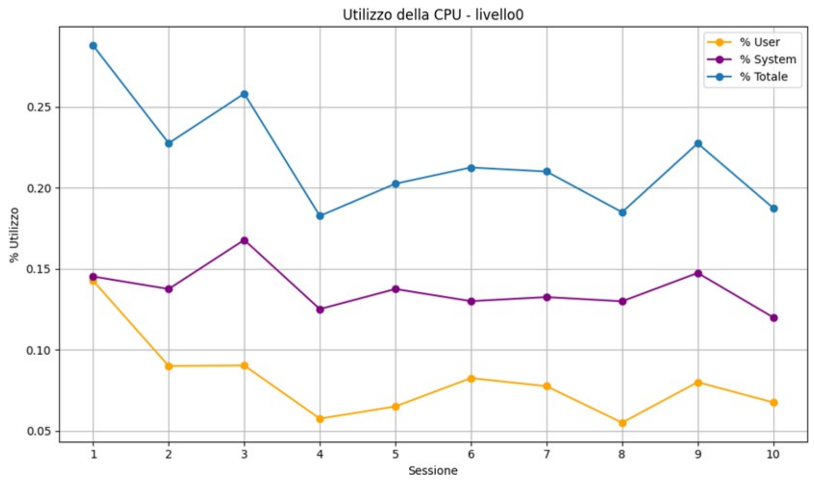


Figure 20. Realistic scenario: CPU utilization in layer 0

- **Level 1:**
  The average number of instructions processed was 1.38×10^9, and the average number of processor cycles was 1.8×10^9.
  The average CPU utilization for User was 0.14% while for System it was 0.27%, with an overall average CPU utilization of 0.41% (Figure 21).

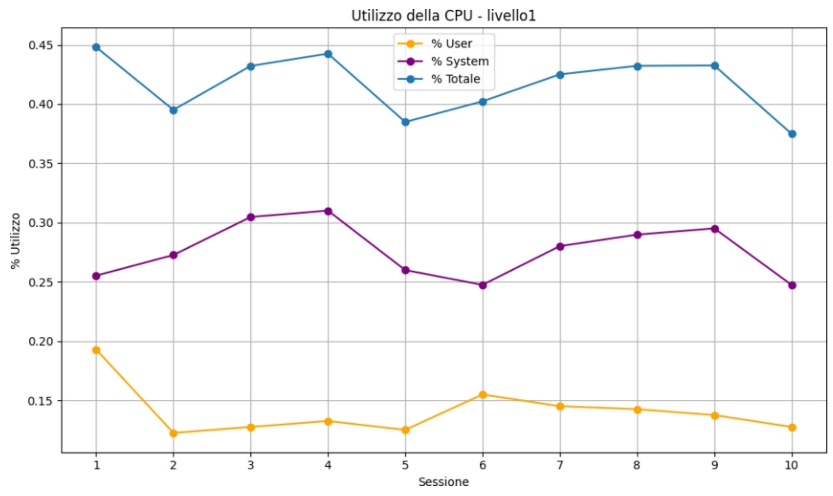


Figure 21. Realistic scenario: CPU utilization in layer 1

- **Level 2:** The average number of instructions processed was 2.06×10^9, and the average number of processor cycles was 2.82×10^9, showing a further increase that underscores a continuous increase in complexity and workload in the system.
  The average CPU utilization for User was 0.23% while for System it was 0.41% for a total average CPU utilization of 0.64% (Figure 22).

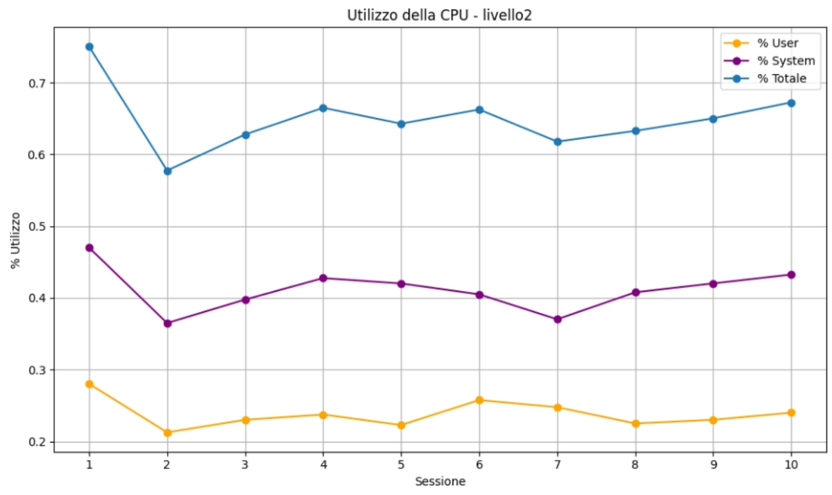


Figure 22. Realistic scenario: CPU utilization in layer 2

- **Level 3:** The average number of instructions processed was 2.04×10^9, and the average number of processor cycles was 2.80×10^9, showing a further increase that underscores a continuous increase in complexity and workload in the system. The average CPU utilization for User was 0.24% while for System it was 0.40% for a total average CPU utilization of 0.64% (Figure 23).

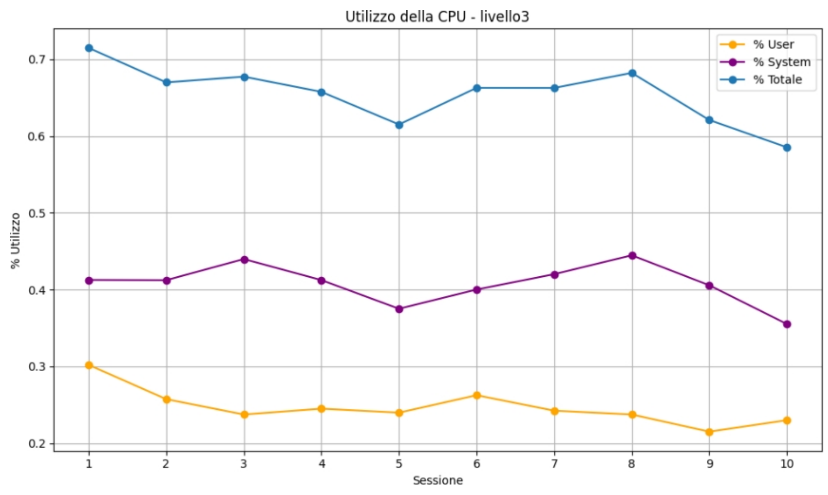


Figure 23. Realistic scenario: CPU utilization in layer 3

A gradual increase in total CPU utilization rates is observed, denoting an increase in iterations and greater load on the CPU (Figure 24).

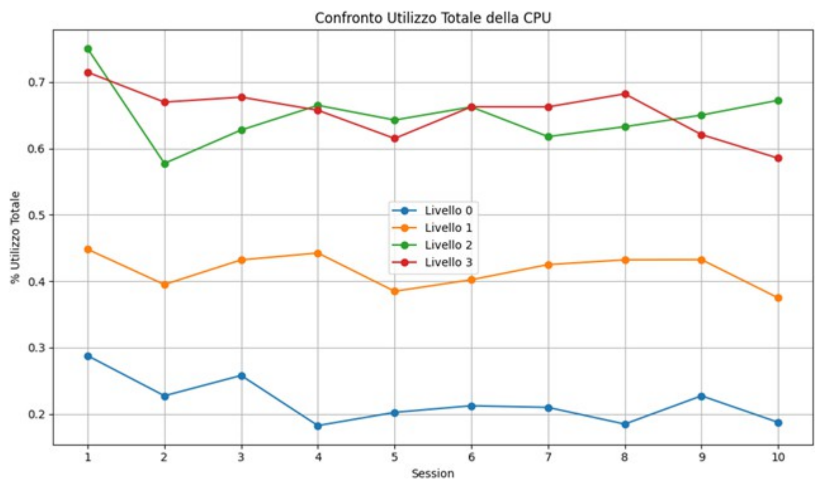


Figure 24. Realistic scenario: comparison CPU utilization by level

The same was noted in the processed instructions and clock cycles (Figure 25-26):

- The transition from Level 0 to Level 1 shows a percentage increase in instructions of about 58.30% and cycles of about 62.93%%.
- The transition from Level 1 to Level 2 shows a percentage increase in instructions of about 62.93 percent and cycles of about 51.52 percent.
- The transition from Level 2 to Level 3 shows a percentage decrease in instructions of about 0.84% and cycles of about 0.74%.

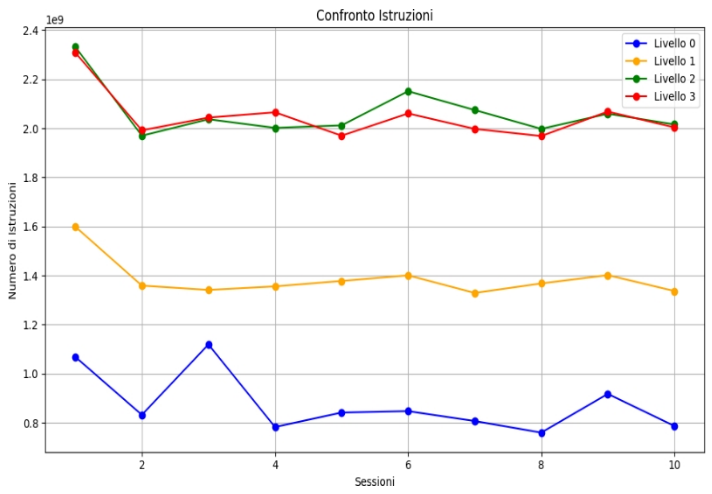


Figure 25. Realistic scenario: comparison of instructions executed by level

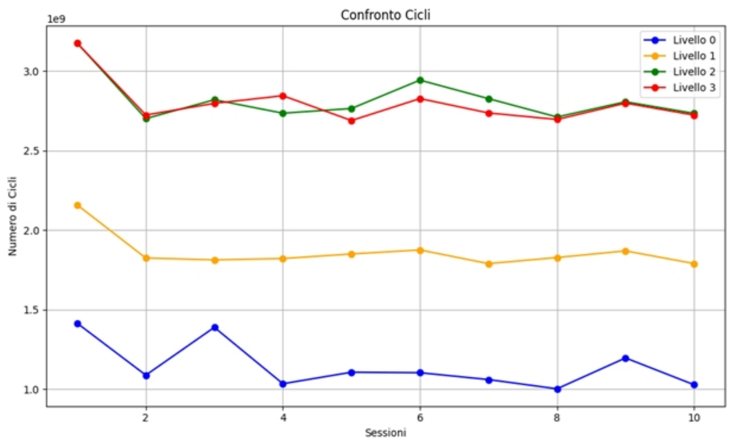


Figure 26. Realistic scenario: comparison of clock cycles by level

Instructions executed: The first graph shows the number of instructions executed in each test for the three levels. It is observed that as the level increases, the number of instructions tends to increase. This can be attributed to the additional complexity or functionality implemented in the higher levels.

Clock cycles: Similarly, the second graph highlights the number of clock cycles for each test, also showing an increase as one moves from one level to the next. The increase in cycles reflects the increased workload and potential increased complexity in the higher levels.

4.4.2 Second test case: Stress test

In this test scenario, we examined a sample of 10 sessions, each lasting 100 seconds, clientPub.js sends a message every 100 milliseconds.

- **Level 0:** Analysis revealed that the average number of instructions processed amounted to 2.8×10^9 and the average number of processor cycles to 4.1×10^9, resulting in an average CPU utilization of 0.87% (Figure 27).

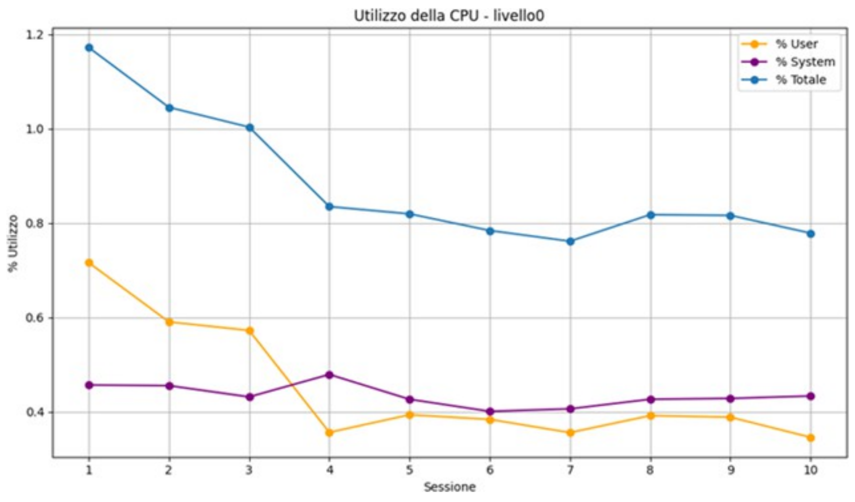


Figure 27. Stress test: CPU utilization in level 0

- **Level 1:** The average number of instructions processed was 6.2×10^9, and the average number of processor cycles was 9.2×10^9. These results indicate an increase from Level 0, reflecting an increase in workload and complexity in the system.
  The average CPU utilization for User was 1.12% while for System it was 1.26%, with an overall average CPU utilization of 2.38% (Figure 28).

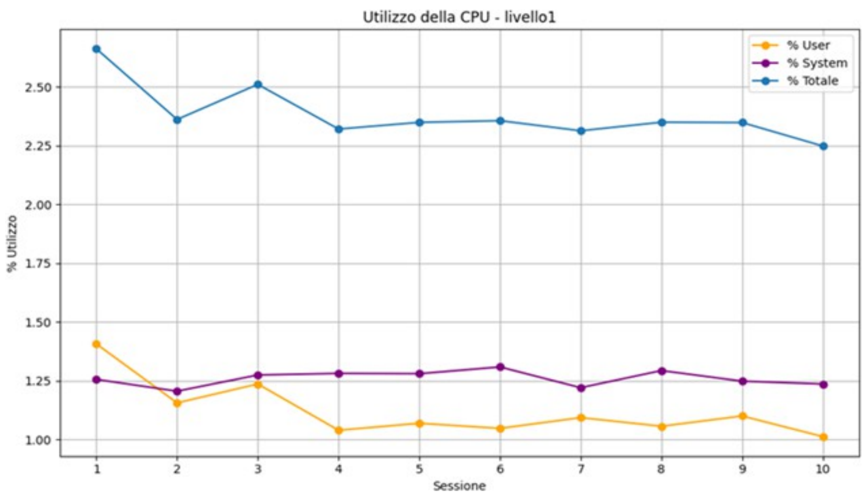


Figure 28. Stress test: CPU utilization in level 1

- **Level 2:** The average number of instructions processed was 8.0×10^9, and the average number of processor cycles was 11.8×10^9, showing a further increase that underscores a continuous increase in complexity and workload in the system. The average CPU utilization for User was 1.39% while for System it was 1.72% for a total average of 3.11% (Figure 29).

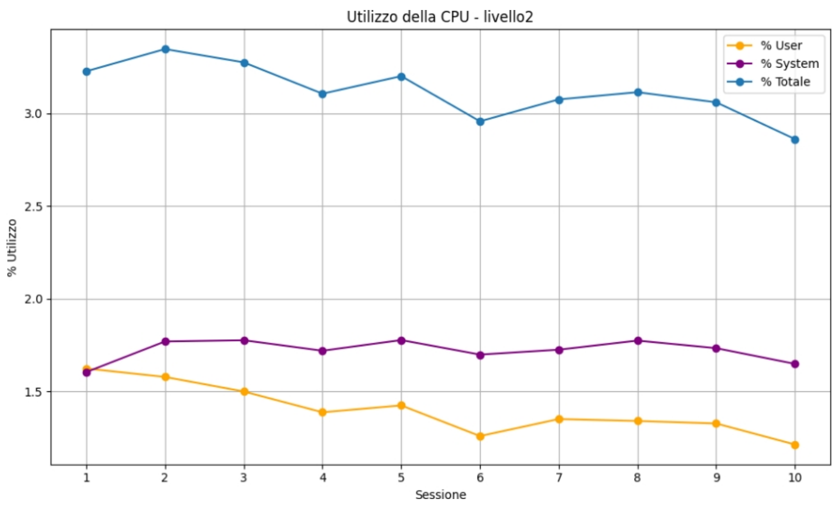


Figure 29. Stress test: CPU utilization in level 2

- **Level 3:** The results are omitted because, similar to what was observed for the previous test, they are on the same average as the antecedent level and, therefore, are found to be insignificant.

Thus, a gradual increase in total CPU utilization rates was observed, denoting an increased load on the CPU (Figure 30).

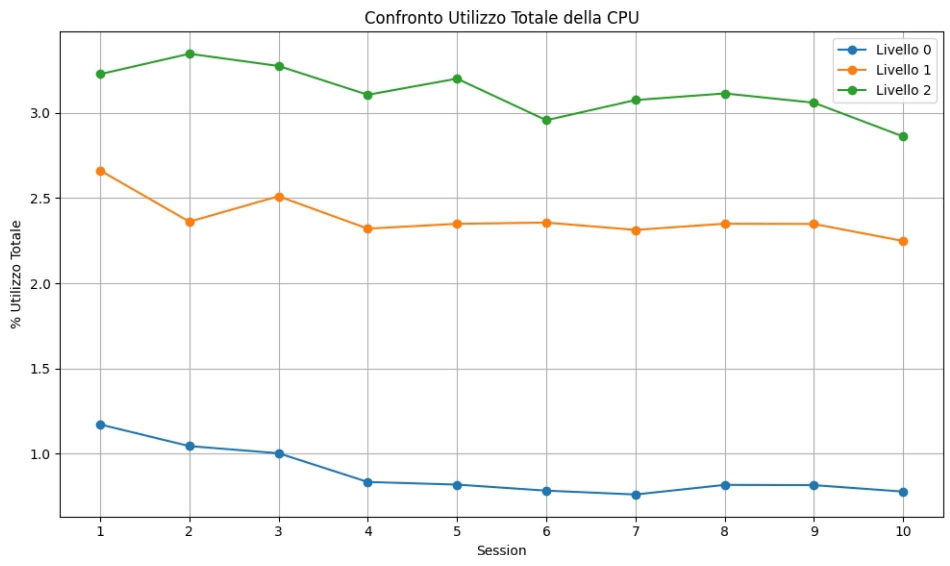


Figure 30. Stress test: comparison of CPU utilization by level

The same was noted in the processed instructions and clock cycles (Figure 31-32):

- The transition from Level 0 to Level 1 shows a percentage increase in instructions of about 116.60% and cycles of about 123.61%
- he transition from Level 1 to Level 2 shows a percentage increase in instructions of about 28.06% and cycles of about 28.59%.

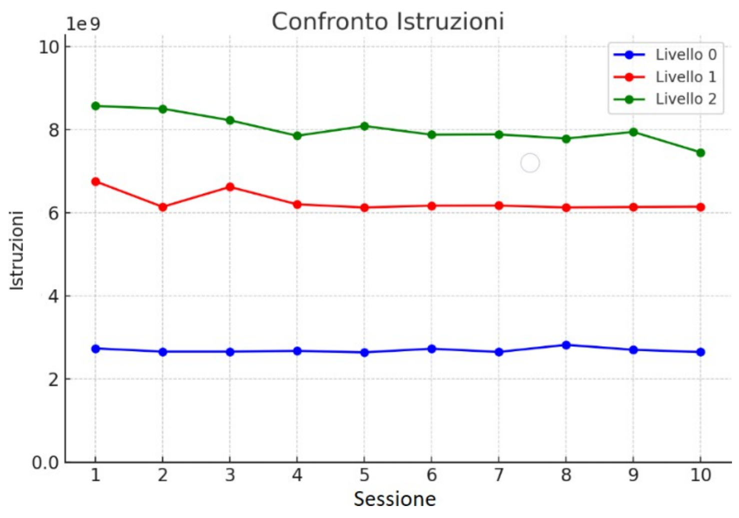


Figure 31. Stress test: comparison of instructions executed by level

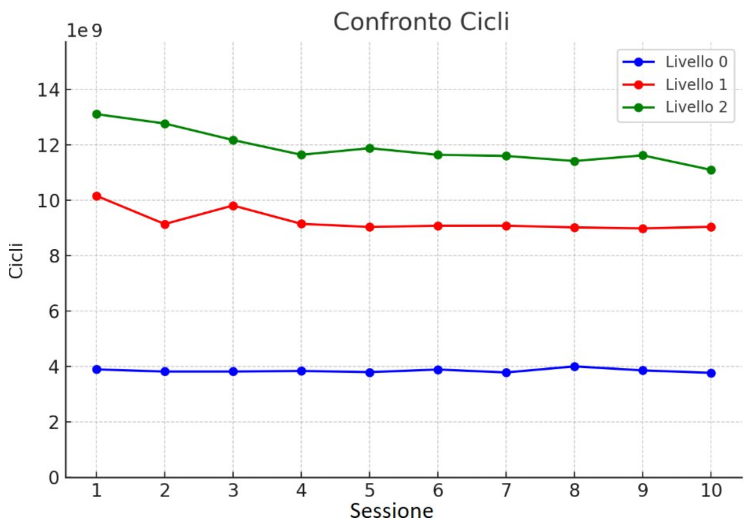


Figure 32. Stress test: comparison of clock cycles by level

Instructions executed: The first graph shows the number of instructions executed in each test for the three levels. It is observed that as the level increases, the number of instructions tends to increase. This can be attributed to the additional complexity or functionality implemented in the higher levels.

Clock cycles: Similarly, the second graph highlights the number of clock cycles for each test, also showing an increase as one moves from one level to the next. The increase in cycles reflects the increased workload and potential increased complexity in the higher levels.

CONCLUSIONS

This project paves the way for many future developments in the field of real-time monitoring of IoT devices. The flexibility of the proposed approach allows further exploration of the field with studies that rely on devices with limited computational capabilities, such as Raspberry Pi.

So far, our work has focused on publish-type MQTT packets. However, the use of Quality of Service (QoS) level 2 introduces a significant variety of packets (such as PUBREL and PUBACK) that can be leveraged to perform more sophisticated cross-checks. This not only broadens the scope of analysis but also increases the accuracy of monitoring, allowing more accurate assessment of communications between devices.

The introduction of tighter controls to analyze the overhead generated by these packages offers an interesting perspective. By examining how overhead varies with increasing control complexity, we can gain useful insights into the efficiency and scalability of monitoring solutions. A comparison of monitoring tools, such as Falco's current kmod and alternatives based on eBPF and modern BPF, may reveal significant differences in performance and analysis capabilities. This comparative analysis could guide the development of more effective and less invasive solutions. The creation of a Falco plugin dedicated to parsing MQTT packets represents another promising direction. This would allow data to be processed directly at the source, potentially reducing latency and improving the efficiency of the monitoring system. Investigating the possibility of applying instrumentation not only to subscribers but also to MQTT brokers could offer performance benefits.

An innovative version of the Adapter, optimized to further reduce overhead, can be developed. This could be achieved through the adoption of a producer-consumer model implemented on a queue, thereby refining the efficiency of the system and minimizing the propensity for snags. In conclusion, the potential for evolution for this project is numerous and diverse. By exploring these avenues, we can not only extend our understanding of real-time IoT device monitoring but also significantly improve the effectiveness and efficiency of existing solutions.

Bibliography

1. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity." In: Proceedings of the 12th ACM conference on Computer and communications security. ACM. 2005, pp. 340-353.
2. Aldweesh, A.; Derhab, A.; Emam, A.Z.. Deep learning approaches for anomaly- based intrusion detection systems: A survey, taxonomy, and open issues. Knowl.-Based Syst. 2020, 189, 105124.
3. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. IEEE Commun. Surv. Tutor. 2015, 17, 2347-2376.
4. Ali, Zainab H., Hesham A. Ali, and Mahmoud M. Badawy. "Internet of Things (IoT): definitions, challenges and recent research directions." International Journal of Computer Applications 128.1 (2015): 37-47.
5. Amarlingam M, P. K. Mishra, K. V. V. D. Prasad, and P. Rajalakshmi. Compressed sensing for different sensors: A real scenario for WSN and IoT. In 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), pp. 289-294, 2016.
6. Ancona D, Luca Franceschini, Angelo Ferrando, Viviana Mascardi, RML: Theory and practice of a domain specific language for runtime verification, Science of Computer Programming, Volume 205, 2021
7. Andrea, I.; Chrysostomou, C.; Hadjichristofi, G. Internet of Things: security vulnerabilities and challenges. In Proceedings of the 2015 IEEE Symposium on Computers and Communication (ISCC), Larnaca, Cyprus, July 6-9, 2015; pp. 180-187
8. Andy, Syaiful & Rahardjo, Budi & Hanindhito, Bagus. (2017). Attack scenarios and security analysis of MQTT communication protocol in IoT system.
9. AR. Bernat and BP. Miller. "Anywhere, any-time binary instrumentation." In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools. 2011, pp. 9-16.
10. Atmoko, Rachmad & Riantini, Rona & Hasin, M. (2017). IoT real time data acquisition using MQTT protocol. Journal of Physics: Conference Series. 853. 012003.
11. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A.

Jaleel, C. Luk, G. Lyons, H. Patil, et al. "Analyzing parallel programs with pins.

In: Computers 43.3 (2010), pp. 34-41

12. Beecks C, A. Grass, and S. Devasya. Metric Indexing for Efficient Data Access in the Internet of Things. In 2018 IEEE International Conference on Big Data (Big Data), pp 5132-5136, Dec. 2018.
13. Brignon and L. Pierre, "Assertion-Based Verification through Binary Instrumentation," 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019, pp. 988-991,
14. Chaabouni, N.; Mosbah, M.; Zemmari, A.; Sauvignac, C.; Faruki, P. Network Intrusion Detection for IoT Security Based on Learning Techniques. IEEE Commun. Surv. Tutor. 2019, 21, 2671-2701
15. Colitti W, Steenhaut K, De Caro N 2011 Integrating Wireless Sensor Networks with the Web Proc. IP+SN Chicago, USA
16. Dhall R. and Vijender Solanki. "An IoT Based Predictive Connected Car Mainte-nance." In: International Journal of Interactive Multimedia & Artificial Intelligence 4.3 2017.
17. Dıaz M, Cristian Martın, and Bartolomé Rubio. "State-of-the-art, Challenges, and Open Issues in the Integration of Internet of Things and Cloud Computing." In: Journal of Network and Computer applications, pp. 99-117, 2016.
18. Dorsemaine B., et al. "Internet of Things: A Definition & Taxonomy." In: 2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies. IEEE, pp. 72-77, 2015.
19. Du, Kunping & Kang, Fei & Shu, Hui & Dai, Li. (2012). Dynamic Binary Instrumentation Technology Overview.
20. Ghayvat H. et al. "WSN and IoT-based Smart Homes and their Extension to Smart Buildings." In: Sensors, pp. 10350-10379, 2015.
21. Haidhar Athir Mohd Puat and Nor Azlina Abd Rahman 2020 J. Phys.: Conf. Ser. 1712 012009
22. Hassanalieragh M. et al. "Health Monitoring and Management Using Internet-of- Things (IoT) Sensing with Cloud-based Processing: Opportunities and Challenges." In: 2015 IEEE International Conference on Services Computing, pp. 285-292, 2015.
23. Jin J. et al. "An Information Framework for Creating a Smart City Through Internet of Things." In: IEEE Internet of Things Journal, pp. 112-121, 2014.

24. Kortuem G., et al. "Smart Objects as Building Blocks for the Internet of Things."

In: IEEE Internet Computing, pp. 44-51, 2009.

25. Lampkin V et al. 2012 Building smarter planet solutions with MQTT and IBM WebSphere MQ telemetry IBM, ITSO
26. Laurenzano, MM. Tikir, L. Carrington, and A. Snavely. "Pebil: Efficient static binary instrumentation for linux." In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). IEEE. 2010, pp. 175-183.
27. Mektoubi, H. L. Hassani, H. Belhadaoui, M. Rifi and A. Zakari, "New approach for securing communication over MQTT protocol A comparison between RSA and Elliptic Curve," 2016 Third International Conference on Systems of Collaboration (SysCo), Casablanca, 2016, pp. 1-6.
28. Minerva R, Abyi Biru, and Domenico Rotondi. "Towards a Definition of the Internet of Things (IoT)." In: IEEE Internet Initiative, pp. 1-86, 2015.
29. Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. "A Platform for Secure Static Binary Instrumentation." In: SIGPLAN Not. 49.7 (2014), 129-140. issn: 0362- 1340.
30. Mishra and A. Kertesz, "The Use of MQTT in M2M and IoT Systems: A Survey," in IEEE Access, vol. 8, pp. 201071-201086, 2020,
31. Mubeen S. A. Asadollah, A. V. Papadopoulos, M. Ashjaei, H. Pei-Breivold, and M. Behnam. Management of Service Level Agreements for Cloud Services in IoT: A Systematic Mapping Study. IEEE, pp. 30184-30207, 2018.
32. Russell, B.; Van Duren, D. Practical Internet of Things Security; Packt Publishing Ltd: Birmingham, UK, 2016
33. Taherdoost, H. Security and Internet of Things: Benefits, Challenges, and Future Perspectives. Electronics 2023, 12, 1901.
34. Vermesan O., et al. "Internet of Things Strategic Research Roadmap." In: Internet of Things-Global Technological and Societal Trends, pp. 9-52, 2011.
35. Vuppalapati, Chandrasekar. Building Enterprise IoT Applications. CRC Press, 2019.
36. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. "From hack to elaborate technique- a survey on binary rewriting." In: ACM Computing Surveys (CSUR) 52.3 (2019), pp. 1-37.
37. Zhang, Mingwei & Qiao, Rui & Hasabnis, Niranjan & Sekar, R.. (2014). A Platform for Secure Static Binary Instrumentation. ACM SIGPLAN Notices.

Sitography

Blogs, Malware Analysis with Dynamic Binary Instrumentation Frameworks, 2021. [accessed online 04/03/2024].

https://blogs.blackberry.com/en/2021/04/malware-analysis-with-dynamic-binary-instrumentation-frameworks

Falco, The Falco Project, 2024. [accessed online 07/03/2024]

https://falco.org/docs/

RMLatDIBRIS Monitor, 2024. [accessed online 12/03/2024]

https://github.com/RMLatDIBRIS/monitor

RML Dibris, 2024. [accessed online 12/03/2024]

https://rmlatdibris.github.io/

RGB565 Color Picker, 2024. [accessed online 16/03/2024]

https://rgbcolorpicker.com/565

eBPF vs. KMod Performance. [accessed online 16/03/2024]

https://github.com/falcosecurity/libs/issues/267