

École polytechnique de Louvain

HubOS: Improving and Redesigning a Local-First, Privacy-by-Design Operating System for Smart Home Applications

Author: **Valentin Kola**

Supervisor: **Etienne RIVIERE**

Reader: **François DE KEERSMAEKER, Lionel METONGNON**

Academic year 2025 – 2026

Master [120] in Cybersecurity

Contents

1	Introduction	3
1.1	IoT Device Integration in Smart Homes	3
1.1.1	Manufacturers Proprietary Applications	3
1.1.2	Cloud-Based Smart Home Application	4
1.1.3	Self-Hosted Smart Home System	5
1.2	Scope of This Work	8
1.2.1	HubOS	8
1.2.2	Local-First Applications	9
1.3	Concluding Notes	13
2	Analysis of the First HubOS Prototype	15
2.1	General Overview	15
2.1.1	Module Execution	16
2.1.2	Communication Flows	16
2.1.3	Trigger-Action-Based Access Control (TABAC) Rules	17
2.2	First Prototype Analysis	18
2.2.1	Runtime Overview	19
2.2.2	Module Execution with VM2	21
2.2.3	Communication Flows	25
2.2.4	An Early Prototype	26
2.2.5	Need for Improvement	29
3	New HubOS: Improved Architecture, Features, and Security Model	31
3.1	App Manager	32
3.1.1	Application Integrity Verification	32
3.1.2	User Agreement	35
3.2	MQTT Broker with Access Control	35
3.3	Sandbox Manager	36
3.3.1	Sandboxing Alternatives	36
3.3.2	Docker with GVisor	40
3.4	TABAC Manager	41
3.4.1	Configuration File	41
3.4.2	TABAC-Rule File	42
3.4.3	TABAC to TAR rule	46
3.5	Resource Access Control	47
3.5.1	Proxy Service	48
3.5.2	Proxy Authentication Header	50

3.5.3	Docker Firewall	51
3.5.4	Permission Manager	52
3.6	REST API Service	52
3.7	New Communications Flows	53
3.7.1	At the Source	53
3.7.2	At the Stream Manager	54
3.8	Additional Security Measures	56
3.9	Overview of the Final Architecture	57
4	Software Release Strategy and Functional Demonstration of HubOS	59
4.1	Packaging	59
4.2	Distribution	60
4.2.1	Open-Source Licensing	60
4.2.2	Open-Source philosophy	61
4.3	Example of an Application	63
4.3.1	Manifest.json	64
4.3.2	Config.json	65
4.3.3	Rules.json	66
5	Conclusion	69
5.1	Reflection	69
5.2	Challenges Faced	70
5.3	Next Steps for HubOS	70
5.3.1	Rethinking Stream Access	70
5.3.2	HubOS Web-Based Graphical User Interface	71
5.3.3	Remote Docker Interaction	73
5.4	Final Remarks	73
5.5	Usage of Artificial Intelligence	74
5.6	Acknowledgments	74
A	TABAC Rules	75
A.1	Trigger Conditions	75
A.2	Action Types	78
B	Software Packaging, Licensing, and Distribution Model of New HubOS	81
B.1	List of imported dependencies with their licenses, versions, and utilities	81
C	HubOS Web-Based Graphical User Interface	83

Chapter 1

Introduction

Nowadays, we are witnessing an increase in connected devices in our living spaces. These objects, often referred to as smart or Internet of Things (IoT) devices, are becoming more and more common in our homes, making daily lives easier and enhancing our comfort. This field, commonly known as **home automation** (or **domotics**), encompasses technologies such as connected printers, smart scales, and kitchen machines that can be controlled remotely. Temperature, humidity, presence sensors, as well as security cameras and connected doors, are also part of this technological shift. These devices not only allow us to centralize the control of our environment, but also enable automation of certain tasks, making our homes smarter and more responsive to our needs. Atzori, Lera, and Morabito, among many other researchers, have produced several studies highlighting the broad range of applications and use cases enabled by IoT technology [1, 2, 3].

In addition to their domestic use, these connected devices are also applied in various professional fields such as healthcare, security, and surveillance. For example, connected medical devices can remotely monitor patient's health, while security cameras and motion sensors are used to ensure safety in professional or public environments.

Bugeja et al. [4] define a smart home as being composed of devices, communication protocols, and services. Devices include all hardware components such as sensors, smart objects, and gateways. Communication protocols can vary, including Zigbee, Bluetooth or Wi-Fi. Finally, services refer to software applications hosted either in the cloud or in the local environment.

1.1 IoT Device Integration in Smart Homes

There are three main approaches to integrate such devices and build a smart home environment.

1.1.1 Manufacturers Proprietary Applications

The first and simplest one is to use proprietary devices with their own application. This refers to all devices sold on the market as “connected”. In most cases, the installation of these devices consists of connecting them to the Internet and to the

brand's application. There are many connected devices with their own applications, such as Philips Hue lights, Arlo cameras, connected speakers from Google, Amazon, and Apple, connected kitchen appliances, and many more. Although this solution is easy to use, it remains very restrictive in terms of configuration and automation possibilities. Another limitation is that the application is restricted to the use of devices of that specific brand. Therefore, an average user may end up having to purchase more devices from the same brand, or alternatively, in order to avoid juggling between different apps to manage your devices. Moreover, the price of some devices can be quite high, making this solution potentially expensive in the long run. Additionally, a major security concern arises as these solutions are dependent on third-party companies. This means that user data and device access are handled by external organizations, which impose their own privacy policy and service agreement, which can pose significant privacy risks. The unrestricted and uncontrolled collection of sensitive data from smart homes in the cloud poses a serious threat to users' privacy and in the event of a breach, sensitive data and device control could be exposed to malicious actors. For example, in 2014, it was reported that over 73,000 video cameras were found streaming their live feeds on the web [4]. Figure 1.1 shows the data flow of an IoT device communicating with the Manufacturer's server.

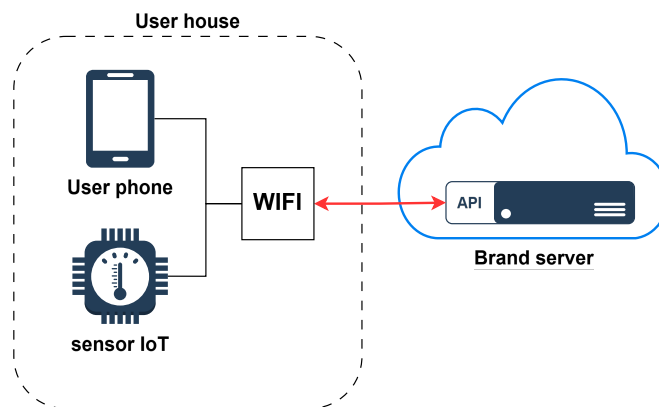


Figure 1.1: IoT Device Data Flow with Manufacturer Proprietary Application [5]

1.1.2 Cloud-Based Smart Home Application

The second option extends the first and helps solve the issue of managing devices across multiple applications. It involves centralizing all devices through an external smart home platform such as Google Home [6], Apple Home [7], or Amazon Alexa [8]. This solution follows a cloud-based model. It means that, the user will have the option to link it to their smart home account. Once the link established, data sent by the device to the proprietary service will also be forwarded to the servers of the cloud-based smart home manager, where it will be processed by applications and trigger automation rules. This architecture offers the advantage of enabling centralized management of devices, simplifying usage and automation. However, this solution has a major downside: the devices must be compatible products, locking the user into a closed ecosystem of compatible devices and services.

Additionally, by using these platforms, we depend on a third-party company not only for controlling our devices but also managing and safeguarding our data, which as mentioned earlier, can raise significant concerns about security and privacy. Several documented cases have shown privacy issues, such as reports revealing that Google and Amazon employees were able to listen to audio recordings from smart speakers of customers [9, 10]. Despite these companies enforcing anonymization methods to protect users, the recordings often contain sensitive information that can potentially identify individuals or reveal private aspect of their lives. Figure 1.2 shows the data flow of IoT data when using a Cloud-based smart home application.

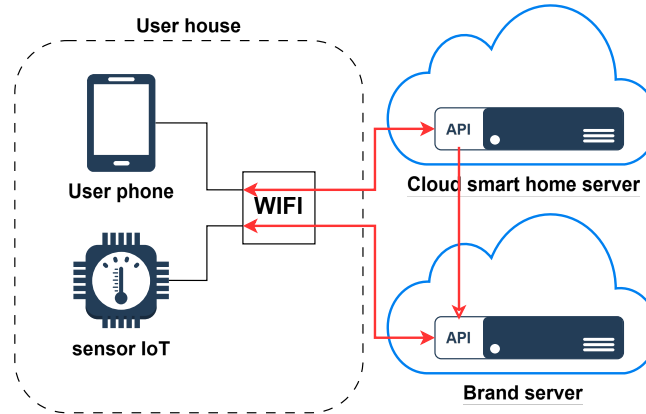


Figure 1.2: IoT device data flow with cloud-based smart home application [5]

1.1.3 Self-Hosted Smart Home System

Finally, in response to the privacy risks and closeness of the cloud-based model, the open source community developed several self-hosted smart home system solutions. Several alternatives exist, such as OpenHAB [11], Home Assistant [12] and Domoticz [13]. These solutions offer maximum flexibility, allowing users to customize their system according to their specific needs. By choosing this option, the user can set up and configure their own home automation server, integrating devices from different brands without being limited by a proprietary ecosystem. Additionally, it provides the possibility to use specific protocols for device communications, such as radio communication with Zigbee [14] and Wi-Fi with MQTT. Although the configuration and management of this type of system require certain technical knowledge, they are designed to be accessible to their users.

This report will focus on the integration of a smart home environment through OpenHAB. The following section provides an overview of OpenHAB, outlining its architecture and core functionalities relevant to its integration.

OpenHAB Open Home Automation Bus (OpenHAB) [11] is an open source home automation system. It allows to control multiple technologies all from a single location. The latest version released to this date is OpenHAB 5. OpenHAB is built around three key objects: **Things**, **Channels**, and **Items**. Things represent physical devices connected to the system. These devices can vary in type: cameras,

smart speakers, connected doors, etc. Each Thing exposes Channels, which correspond to specific data points of the device/Thing (e.g., power status or temperature reading). Finally, Items are elements linked to Channels and used in rules or the user interface, enabling interaction with the system [15]. In addition, the concept of Rules allow the creation of customized automation. These rules can be used to trigger actions when specific event occurs. For example, when the door opens, the living room light automatically turns on. The OpenHAB framework also provides an event bus for inter-component communication. Devices and components can send and receive events through the event bus in an asynchronous way [16].

Despite the advantages of implementing a self-hosted smart home environment, it is not inherently secure under all configurations. In particular, two key security considerations must be kept in mind: enabling remote access to certain resources, and installing third-party applications. These aspects are analyzed in the following sections.

Enabling remote access to certain resources A self-hosted smart home system is primarily local and is based on the use of a smart hub, a computer system to which all smart devices of the home are connected. In this case, two options are available: First, as shown in figure 1.3, we can keep the system completely isolated within the local network, so no external access is authorized, ensuring maximum security by keeping all data and controls within the home network. The second option is to allow access from outside the network. This option can be achieved by enabling a VPN¹, opening specific ports on the router, or using other secure remote access solutions. While remote access provides more flexibility, it also introduces security risks, as each external access point represents a potential entry for cyberattacks.

Therefore, from the perspective of a simple user without technical knowledge in cybersecurity, one question arises: "How can we allow a smart home system to access resources outside of the local network while also being able to send data externally in a secure manner?" It is crucial to find a balance between the flexibility of accessing external resources and protecting personal data. For example, the use of security cameras, which are connected locally but send the video stream in real-time to the owner via a remote application, enables monitoring from anywhere. Accessing the video stream locally generally doesn't pose a security problem, as the data remains within the internal network and under the user's direct control. However, once we made it available externally, it raises critical questions about the security of our data. Can we trust third-party services that manage this access? Is it a good idea to continuously stream video over the Internet, especially considering privacy risks? What would happen if a malicious person took control of our video stream access service? The questions highlight the need for secure solutions to protect sensitive data and prevent unauthorized access.

¹Virtual Private Network

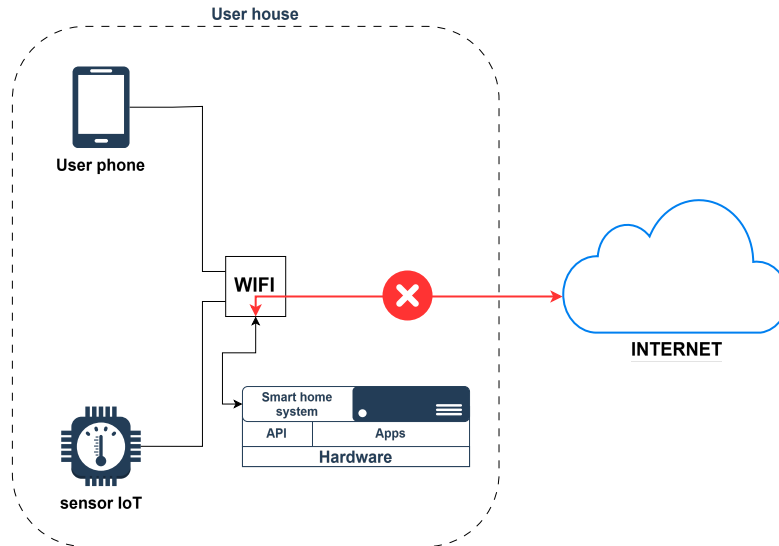


Figure 1.3: IoT device data flow with self-hosted and local smart home system [5]

Installing third-party applications Then, in an effort to expand the capabilities of their home automation environment, the owner may be interested in adding new application or integration in the form of scripts. These application can serve different purposes, such as enabling person detection through security cameras [17], or optimizing plant care using a local machine learning model [18]. Additionally, these scripts can originate from various sources: On one hand, the owner may develop the application scripts themselves (if they have the required technical skills). On the other hand, they can rely on third-party solutions and install pre-existing projects from Git repositories, or even install integrations directly available through their chosen smart home app store.

In the first scenario, where the code is entirely written by the owner, security risks are primarily limited to potential bugs in the source code and vulnerabilities inherent to the smart home platform itself. However, when the source code originates from a third party, the risk landscape becomes more complex due to two possible issues:

1. The developer is well-intentioned, but (similarly to the first scenario) the code may contain security vulnerabilities that could compromise the safety of the smart home environment. For instance, in January 2021, it was discovered that several third-party Home Assistant integrations contained vulnerabilities allowing an attacker to steal any files without logging in [19].
2. In contrast, the developer may be malicious and, under the pretext of providing a legitimate application (for example, a facial recognition door unlocking system), intentionally injects a malware to exfiltrate data or compromise the system. For example, in June 2025, the ReversingLabs Threat Research Team reported a campaign involving more than 60 Github repositories that distributed trojans disguised as Python-based hacking tools [20, 21].

This scenario highlights a critical security concern: access to resources that should not be available by default. Applications should not have unrestricted access to all resources within the environment or on the host machine.

As highlighted by Ali and Awad in their study on security vulnerabilities in IoT-based smart homes [22], whether using cloud-based or self-hosted architectures, smart home solutions inherently pose significant security and privacy risks that must be carefully addressed. Identifying these threats is essential to propose appropriate countermeasures and ensure the protection of users and their data.

To address this security issue, various studies have been conducted to propose secure smart home systems. Some research focuses on the security of the devices, others on the communication protocols between them, and others on mitigating vulnerabilities related to services [4]. This report will focus on the third option - mitigating vulnerabilities related to services - through the presentation of HubOS, a secure-by-design operating system aiming to enable the isolated and secure execution of third-party applications alongside the operation of a smart home system.

1.2 Scope of This Work

This thesis builds upon the previous study "Local-First Smart Home Applications with HubOS" [5], published in the EDCC² paper of 2025, which was carried out by researchers from UCLouvain.

The objective of this work is to develop a concrete system based on the HubOS study and its conceptual design proposal [5], and to revise the technological choices initially made in that context in order to design a more robust and secure solution.

1.2.1 HubOS

HubOS aims to propose an open source operating system that integrates with the existing open source home automation platform, OpenHAB.

Currently, it is very common for services handling domestic data to communicate with external resources, for example, to perform updates, synchronize data with the cloud, or interact with third-party platforms. However, it is often difficult to determine whether certain applications or services are truly trustworthy and whether they might attempt to exploit our data for commercial purposes, out-passing our privacy, or even gain unauthorized access to our local network. Therefore, it is crucial to limit the access of these services strictly to the devices and network segments for which they have been explicitly authorized.

The goal of HubOS is to provide a secure by design additional layer of security for the outgoing data flows from our smart home applications. For example, is it necessary for the video stream from our living room surveillance camera to be accessible at all times on an external network? Wouldn't we prefer to allow access to this video stream only when a specific case authorizes it?

²European Dependable Computing Conference

HubOS aims to address this need with the concept of **local-First** design. This concept refers to types of application that prioritize processing data locally (i.e., within the user’s local/internal network) and, if needed, rely on an external service to address a specific need, for a short period of time. As Kleppmann et al. describe in their study on local-first applications, in cloud-based apps [23], user data is primarily stored on the company’s servers, clients can only view or modify their data by issuing first a request to the server which responds with a copy of the data. This approach is very common in current models of video surveillance services or smart assistants. In contrast, a local-first model reverses this logic: data is primarily stored and processed locally, and only sent to specific servers when explicitly required by the user.

HubOS enables specific control over the data flow from smart devices by allowing users to define specific conditions under which certain data streams can be accessed. This ensures that sensitive information, such as video feeds or personal data, is only accessible when necessary and under predefined circumstances, according to the principle of least privilege. This critical task within the HubOS system is handled by OpenHAB and its rule engine. By implementing this level of control, HubOS enhances the overall security of smart homes and IoT devices, preventing unauthorized access while ensuring that legitimate use cases are supported.

The system aims to engage both developers and users with limited technical knowledge. Regular user will be able to install and launch local-first applications with minimal effort.

In the other way, developers can create local-first application and set of rules that specify when and how data is processed and transmitted, creating a more secure and flexible environment for managing their devices. This secure by design approach not only protect users privacy but also helps build trust in the interconnected systems that are increasingly prevalent in modern homes.

1.2.2 Local-First Applications

In the this section, we present three examples of local-first applications, taken from the initial research on HubOS, to illustrate the system’s potential use cases [5].

Front Door Smart Security Camera

Description The first application is a facial recognition service. A video camera continuously send a video stream to the service, which automatically detects and identifies individuals at the front door. If the person is not recognized, a notification is sent to the homeowner along with a snapshot of the video.

Cloud-based implementation In a cloud-based implementation of this service, the entire video stream would typically be sent to an external server for analysis, where the AI facial recognition model used to identify authorized individuals is also stored. A leak of any of these elements would represent a major privacy risk for the client.

Local-first implementation A local-first implementation of the smart security camera service could be designed by separating it into three distinct components:

- The first component is responsible for facial recognition. It implements the AI-based recognition model and has direct access to the surveillance camera's video stream. This component is strictly confined to the local network, ensuring network isolation. When this component detects an unrecognized individual, it triggers an event.
- At that point, a second component (authorized to communicate outside the local network) handles the task of sending a notification to the homeowner, along with video snapshot, to an external or cloud server.
- The third component is the external server responsible for storing these snapshots. It has no direct access to the local network or the camera feed, and only receives the data explicitly sent by the second component, maintaining a clear separation of responsibilities.

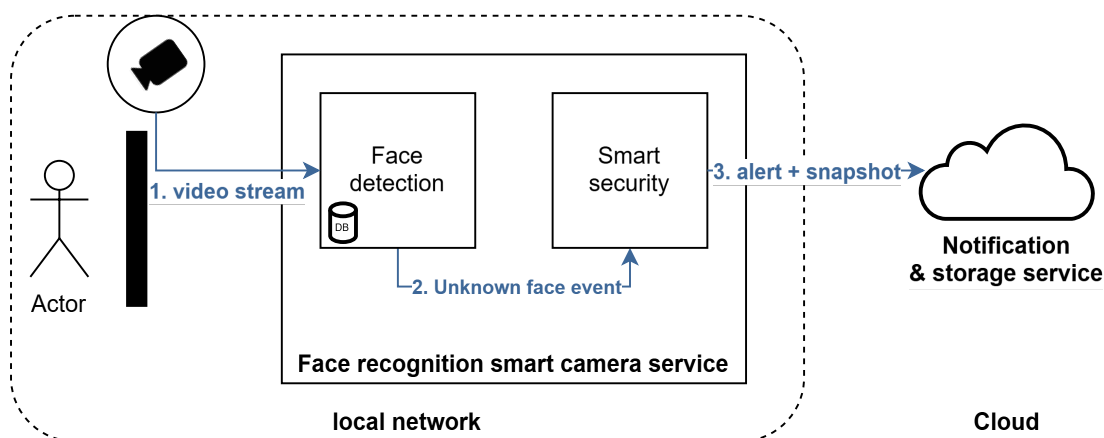


Figure 1.4: Front door smart security camera use case with a local-first implementation [5]

Voice-Controlled Butler

Description The second example corresponds to the implementation of a voice control system, such as Amazon Alexa [8], Google Nest [24], or Siri [25]. This type of system allows users to control a wide range of devices using voice commands. For instance, playing music through smart speakers, ask for the weather forecast, or adjust the thermostat.

Cloud-based implementation In a cloud-based implementation of this concept, the smart speaker is programmed to activate when hearing a specific wake word or phrase, such as "Hey Siri" or "Okay Google". The device is designed to locally recognize this trigger phrase. Once detected, it sends the subsequent audio recording to a cloud server for processing in order to fulfill the user's request.

It is important to highlight that such recordings can pose a significant privacy risk if leaked. Moreover, users have no clear assurance that these recordings are not being used for other purposes by the company providing the service, nor that they cannot be intercepted or misused by a malicious actor.

In addition, complex speech-to-text models can also be used to recognize or infer a user's identity. Audio recordings are known to carry much more information than it may appear: they can reveal characteristics such as a person's age, sex, nationality, accent, or even certain health conditions [26, 27, 28]. A leakage of speech recordings therefore introduces additional risks, including identity theft, as audio samples can be misused to train generative models capable of impersonating user [29].

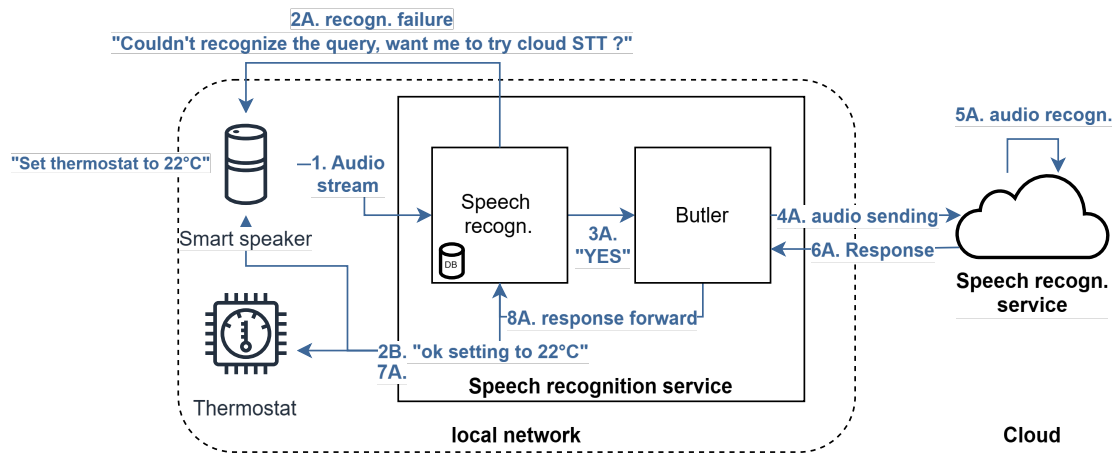


Figure 1.5: Voice-controlled butler use cas with a local-first implementatione [5]

Local-first implementation To implement a local-first version of this concept, it is necessary to design a system that prioritizes local speech recognition for basic voice commands, and relying on external resources for more difficult request. A first component would be responsible for accessing the audio stream from the smart speaker and attempting to recognize the user's request locally and process it. However, since this processing occurs on local hardware with limited computational and linguistic capabilities, it cannot match the accuracy or flexibility of cloud-based voice assistants. By consequent, when the local component is unable to interpret the request, it can delegate the task to a second component.

This second component is responsible for securely transmitting the audio to a cloud-based speech recognition server, enabling more advanced processing only when necessary. Importantly, before any audio is sent, a voice-scrambling mechanism is applied to reduce the risk of user re-identification and to prevent the construction of accurate generative models that could impersonate the user [30, 31, 32]. Finally, the result returned by the cloud server is sent back to the local speech recognition module for execution.

It is important to note that, just like in the first application, the two modules differ in terms of network access. The first module is strictly confined to the local

network, while the second is allowed to communicate with external servers. By consequent, if the the speech recognition module needs to access resources that require an internet connection, such as retrieving the weather forecast, the request must be relayed through the second module.

To simplify the model, this application is limited to recognize only voice commands for changing the temperature of a thermostat.

Fall Detection and Remote Assistance

Description In the context of elderly care, the risk of falls is a major concern. To help ensure their well-being, motion sensors, cameras, or health monitoring devices can be integrated within the home and made accessible to healthcare services, enabling the detection of such incidents and the delivery of immediate assistance when needed.

Cloud-based implementation In a cloud-based implementation, all measurement device data is continuously sent to the company responsible for the healthcare service for analysis. Unfortunately, as with previous applications, entrusting third parties with the management of such sensitive data poses a significant privacy risk.

Local-first implementation A local-first solution breaks this service into two separate modules. The first module is restricted to the local network and is responsible for intercepting data from monitoring devices and processing it to detect potential falls. After detecting such an incident, it generates an event that is sent to the second module, which is authorized to temporarily transmit the video stream from surveillance cameras to the healthcare provider for a limited time. This allows the healthcare company to review the footage and determine the appropriate course of action.

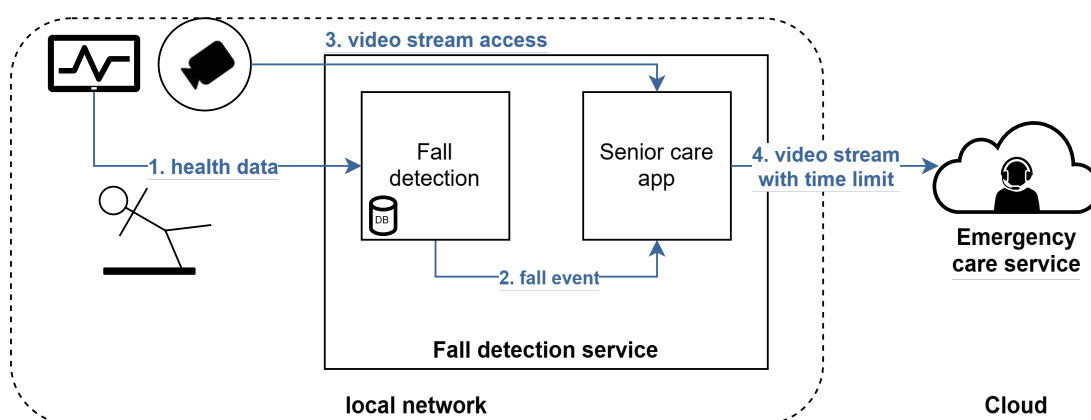


Figure 1.6: Fall detection and remote assistance use case with a local-first implementation [5]

Separation of Network Access

Our implementation of a local-first model requires the system to be composed of multiple modules or components, each with specific responsibilities, in order to enforce a strict separation of network access. A module that requires regular access to local data will not be allowed to communicate with the external network. Conversely, a module that needs to interact with external services will have limited or no access to internal resources. This architectural separation helps reduce the risk of data leakage and ensures that only explicitly authorized data leaves the local environment.

1.3 Concluding Notes

We have seen that smart home systems represent a vast and diverse field, with configurations varying significantly from one household to another. Additionally, three main categories of smart home integration can be distinguished: Manufacturers Proprietary Applications, Cloud-based Smart Home Applications, and Self-Hosted Smart Home Systems. Each offering specific advantages but also presenting certain limitations.

Throughout this chapter, we reviewed the challenges raised by cloud-based configurations, particularly in terms of data privacy and reliance on external services. As a way to address these concerns, we introduced the concept of a local-first application model, where data is processed locally first, before being granted access to external resources if necessary. To manage such a system, we presented HubOS, which provides a secure-by-design additional layer of protection for outgoing data flows originating from smart home applications.

In the next chapter, we will examine in detail the logic behind the first version of this system and identify the key components that require revision or enhancement in order to deliver a refined and production-ready version of the concept.

Chapter 2

Analysis of the First HubOS Prototype

This chapter begins with a high-level overview of HubOS as presented in its original study, followed by a discussion of its core concepts. Subsequently, we analyze the first implementation of the system, developed as part of the original research, highlighting key features introduced by the previous developer and identifying areas requiring further refinement or updates.

We consider a smart home to be equipped with multiple connected devices, sensors, and appliances, all linked to a central hub that acts as the single point of connection for the entire system. As a result outgoing communications from the devices are directed to the hub, rather than to the cloud.

2.1 General Overview

As mentioned above, HubOS allows users to develop and install applications. These applications consist of 3 main components: a set of **rules**, a set of **modules**, and a **manifest.json** file that links all the modules included in it. Additionally, if a module requires access to resources that depend on the environment of the host user, the developer can define a **.env** file located at the root of the module. This file specifies the necessary environment variables, which the user must complete before using the application.

In this section, we present a high-level overview of HubOS and focus on three key concepts:

1. The execution of **module scripts**;
2. The **communication flow**;
3. The concept of **Trigger-Action-Based Access Control (TABAC) rules**;

2.1.1 Module Execution

In the example presented earlier, each application is composed of two distinct **modules**, each with a specific role: a **Service** module and a **Handler** module. The Service module is responsible for processing data locally, with fewer restrictions on access to the local network. Its role includes handling data and publishing events that, if they match the defined rules, will authorize the handler module to process the request and communicate with authorized external services.

Modules are executable code that can take three possible forms: Simple interpreted scripts using the JavaScript language and runtime; Binary module supporting compiled code; and WebAssembly [33], a portable low-level bytecode with a compact representation, efficient validation, and fast decoding and execution, designed to be safe and efficient without significant overhead.

Each application must comply with the principles of **separation of duties** and **least privilege**, ensuring that modules remain isolated and only have access to the strictly necessary resources for their operation. Each module performs a single operation and only requests access to the resources necessary for its purpose. On the other hand, the rules specify which types of access can be granted based on contextual conditions and events. These rules must be approved by human users.

Moreover, to prevent potential data leaks between applications or even between modules of the same application, each module must be isolated, with no access to system resources other than those explicitly granted.

2.1.2 Communication Flows

Currently, HubOS highlights three communication flows for modules:

- Between one module and another;
- Between a module and a local device;
- Between a module and external web services.

As part of an initiative to apply a least-privilege model to HubOS, our modules need access only to the minimum resources required to perform their tasks. Consequently, access to these resources must be restricted by default and only permitted when explicitly authorized by HubOS (and, by extension, the user-defined rules).

Following this logic, HubOS adopts two distinct communication buses to manage interactions between components: one for streams and one for events. In addition, it provides authorized modules with a network client to enable communication with approved web resources and domain names.

Event Bus This communication bus allows modules to notify HubOS of an event. If the event triggers a rule, HubOS is responsible for executing the corresponding actions (for example, granting a designated module access to a specific device).

Flow (or Stream) Bus HubOS does not allow application modules to connect directly to local devices. This task is delegated to the stream bus, which allows modules (depending the rules) to register for and receive continuous streams of data based on their name. (e.g., to receive the video stream from a surveillance camera).

Network client By default, modules do not have direct access to web resources. When a rule allows the execution of a module with access to these resources, HubOS grants the module access to its `NetworkClient` interface, which is responsible for performing the requested web query.

2.1.3 Trigger-Action-Based Access Control (TABAC) Rules

In the previous section, we discussed that, in accordance with the principle of least privilege, access to resources is restricted by default and that access permissions are granted only when the situation allows it. Therefore, it is necessary for a user to be able to define (or at least approve) these specific situations in which modules are granted access rights. HubOS ensures this through the creation of a rule model named `Trigger-Action-Based Access Control` or `TABAC` for short. Moreover, following the philosophy of HubOS, it is essential to pay close attention to the structure and content of these rules. In fact, previous research has shown that permission screens are rarely read by the users [34, 35], and that there can be a significant gap between a user's understanding of smart home devices and the device's actual behavior [36, 37]. Therefore, it is important for HubOS to ensure that its rules follow a philosophy focused on simplicity and user comprehension with clear and concise definitions, helping users make informed and secure decisions.

The concept of TABAC rules is presented as "a novel access control model targeting flexibility and human understandability"[5], extending the traditional `Trigger-Action Rule (TAR)` standard, which is commonly used in most smart home platforms such as `OpenHAB` [5]. Unlike access control models such as `Role-Based Access Control (RBAC)` [38] or `Attribute-Based Access Control (ABAC)` [39], which rely on predefined conditions, the goal of TABAC is to offer a more dynamic form of access control. In other words, certain permissions can depend on the result of computation performed by a module, making access decisions more context-aware and flexible.

By using TABAC rules, users can define specific conditions under which a module is permitted to access sensitive data, connect to the internet, or send commands to smart devices. Each application is expected to declare in its manifest the accesses each of its modules needs for fulfilling its purposes and that the user must accept for the application to operate. All other accesses are disabled by default.

```
1 rule <name>
2 IF
3     <trigger condition> [or <trigger condition> [or ...]]
4 THEN
5     <action> [ and <action> [and ...<action>]
```

Listing 2.1: Simple structure of a TAR rule

Just like in the structure of a traditional TAR rule (as shown in Figure 2.1, the first element is a unique name that should be descriptive enough to help reader understand the purpose of the rule. Next, the trigger section defines the boolean expressions to determine the conditions under which the rule is triggered. These expressions are based either on events published on the event bus (originating from devices or other modules) or on environmental (time of day, date, etc.). Finally, the last section of the structure defines which resources or services should be granted or denied access, optionally specifying a period of time for which the access remains in the effect.

```
1 TABAC rule <name>
2 IF
3     <trigger condition> [or <trigger condition> [or ...]]
4 THEN
5     <allow/deny> <access> <resource> (<period>) [ and ...]
```

Listing 2.2: Simple structure of a TABAC rule

The range of resources to which the **Then** section of TABAC rules applies in HubOS includes:

- The access to network services (to specific IP or DNS name);
- The ability to subscribe to a stream or produce one;
- The access to all or a subset of the API of a smart device;
- The right to use and access local storage;

Of course, once the defined time period has elapsed, the granted accesses are automatically revoked, and the system reverts to its default state, where those accesses remain denied.

2.2 First Prototype Analysis

In parallel with the HubOS study, the original team developed an initial version of the system. This first implementation, developed in JavaScript and executed using Node.js, served primarily as a demonstration of the theoretical foundation of HubOS. It was a basic proof of concept, heavily relying on a simple implementation specifically built around the three application examples presented earlier.

To introduce the next chapter, which focuses on the new implementation of HubOS, it is important to revisit its first implementation in order to analyze how the previous developer implemented the core concepts presented previously and to determine whether they need to be modified or updated. This includes examining the implementation of the sandboxing module, the communication buses, the service responsible for interactions with external services, as well as the limited integration of OpenHAB and its rule engine. The implementation of the proposed solutions, however, is reserved for the following chapter.

2.2.1 Runtime Overview

Next, to better understand the implementation of the core components, it is useful to present an overview of the HubOS runtime as presented in its first report in order to identify how these components are utilized within the system.

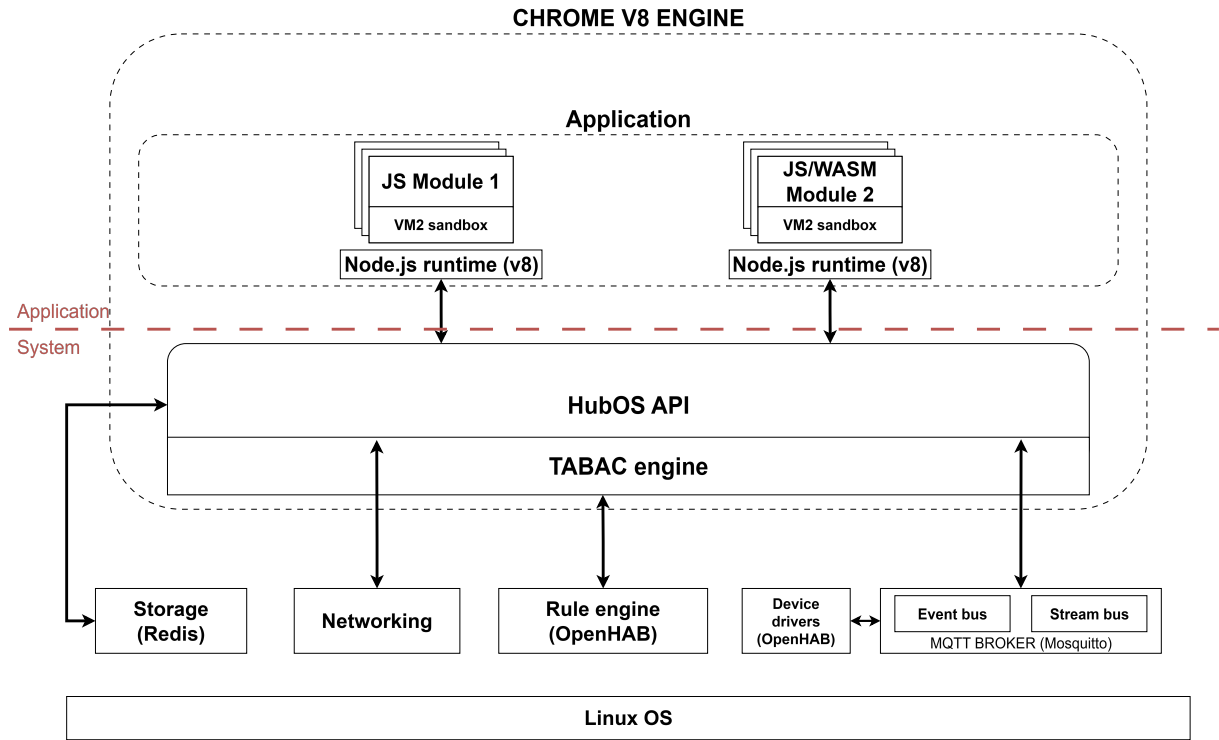


Figure 2.1: The architecture and interactions of the original HubOS components [5]

Figure 2.1 illustrates the main components of HubOS and their interactions, as defined in the previous research work. The system uses GNU/Linux as the hub base operating system, with its core components operating as standard Linux processes within shared namespaces. In contrast, application components are executed in separate namespaces with enforced resource limitation. Interaction between the modules and HubOS services is facilitated via a small shim layer embedded within the modules, which exposes the HubOS APIs.

When HubOS is executed, the service checks the applications manually added by the user, and for each of these applications, the program performs a series of actions. To illustrate this process, we refer to the use case and TABAC analysis previously introduced in sections 1.2.2 and 2.1.3 concerning the front door smart security camera. This process is also illustrated in Figure 2.2, which provides a step-by-step overview of all the stages involved, as will be described in the following paragraphs.

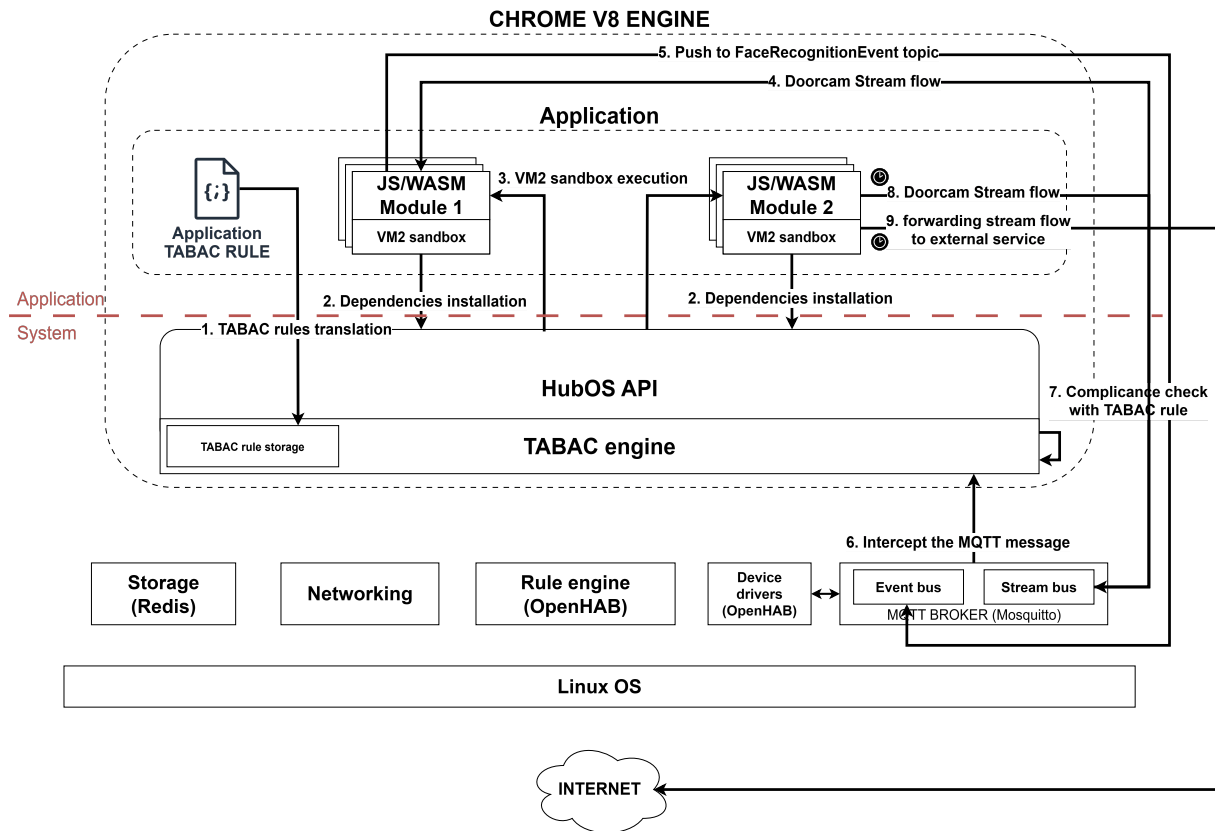


Figure 2.2: Runtime example of the use case front door smart security camera

1. Development An optional first step is the development of an application composed of modules. In this step, the developer needs to define various configuration files, such as, a **rules.json**, a **manifest.json**, and a **.env** file as illustrated in Figure 2.3.

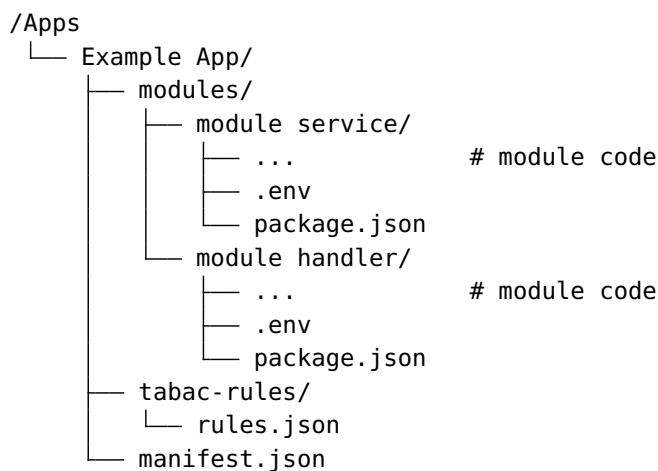


Figure 2.3: File Structure of an App

2. Initialization The first phase of the system is the initialization of the applications and their modules. This step consists in defining the paths of the different applications, extracting the data specified in their manifest file as well as any available data located in the '<application-name>/resources' folder, and storing them in a Redis database.

3. TABAC Engine (step 1) The second step consists of extracting the **TABAC** rule file. The TABAC Engine will be responsible for checking whether an event is triggered and executing the corresponding actions.

4. Script execution (step 2-3) After that, HubOS extracts the scripts of each application's modules, installs their dependencies and executes them inside an **isolated** sandbox.

5. Stream request (step 4) In this use case, the service module continuously accesses the video from the doorcam via the stream bus and analyzes the images in order to detect any unknown faces.

6. Event triggering (step 5-9) If the service module detects an unknown face, it reports this event via the event bus. HubOS then intercepts the event and delegates the task of access control to the TABAC engine. In this case, the handler module is granted access to both the external network and the doorcam for a duration of 30 seconds. From that point on, the handler module accesses the doorcam's video stream through the stream bus and forward it to an external service, allowing the user to view the video stream remotely. Once the 30 second period has elapsed, access is automatically revoked by the TABAC engine, and the handler module has no longer permission to access these resources.

2.2.2 Module Execution with VM2

The original project uses an open source Node.js library named VM2 [40] to run applications developed by external developers in an isolated manner. It runs untrusted code securely in a sandbox with whitelisted Node's built-in modules. Unfortunately, the library is currently discontinued due to critical security issues.

During the development of an application, evaluating the security posture of third-party libraries is crucial, as existing vulnerabilities could compromise the application's integrity. One effective way to check for such vulnerabilities is by referring to **CVEs (Common Vulnerabilities and Exposures)** [41] that have been reported and cataloged in public databases. According to Aggarwal [42], the CVE system tracks and catalogues vulnerabilities in both software and hardware. It is widely used as a standard by security professionals to identify, evaluate, and reference known vulnerabilities. Once a vulnerability is identified, its severity is assessed using the **CVSS (Common Vulnerability Scoring System)**, resulting in a score ranging from 0 (no impact) to 10 (critical impact).

In July 2023, two critical vulnerabilities were discovered in the latest version of the VM2 library. Although different in execution, both share a common trait: they allow attackers to escape the sandbox and execute arbitrary code. The first vulnerability **CVE-2023-37466** [43] leverages a manipulation of the **Promise[species]** behavior to break out of the sandbox and gain unauthorized execution capabilities. The second vulnerability **CVE-2023-37903** [44] exploit Node.js's **custom inspect** function to inject and execute arbitrary code during object inspection.

In the following we will present the two critical vulnerabilities as demonstrated by Lee and Leesh3288 [45, 46].

Sandbox Escape via Promise[species] [46]

First of all, as host exceptions in async context (Promise) may leak host objects into the sandbox, the vm2 library override `Promise.prototype.then` with a proxy to sanitize the arguments before invoking the user-provided `onRejected` handler.

To bypass that, the exploit take advantage of the JavaScript mechanism called **@@species** (also known as `Symbol.species`), which allows overriding the constructor used when performing operations on Promises (such as `Promise.then()`). By leveraging this feature, the attacker can inject a custom constructor that gets automatically invoked by the JavaScript v8 engine.

This v8 engine is the name of the JavaScript engine developed by Google. It is used in Node.js to compile and execute JavaScript efficiently with WebAssembly support.

Listing 2.3 illustrates this exploit in practice. To better understand how it works, let us describe the key steps involved in the attack:

- **Step 1:** An asynchronous function is declared that produces a rejected Promise by throwing an exception;
- **Step 2:** The Promise's constructor is overwritten with a custom object defining a `Symbol.species` accessor, which returns a malicious constructor ;
- **Step 3:** When the `then()` method is called, the JavaScript engine follows the ECMAScript 2022 specification for `Promise.prototype.then` [47], retrieving the constructor via `@@species`. The malicious constructor is triggered, and the provided executor receives **resolve** and **reject** callbacks;
- **Step 4:** The malicious executor abuses the **reject** callback to access the host process object and run arbitrary code;

Therefore, even though vm2 attempts to sanitize the arguments of `then()` using a proxy, the use of `@@species` allows attackers to bypass this protection and execute a malicious constructor before the proxy comes into play.

```

1  const {VM} = require("vm2");
2  const vm = new VM();
3
4  const code = `
5      async function fn() {
6          (function stack() {
7              new Error().stack;
8              stack();
9          })();
10     }
11     p = fn();
12     p.constructor = {
13         [Symbol.species]: class FakePromise {
14             constructor(executor) {
15                 executor(
16                     (x) => x,
17                     (err) => { return err.constructor.constructor('return process')
18                             ().mainModule.require('child_process').execSync('touch pwned
19                             '); }
20                 )
21             }
22         };
23     };
24     p.then();
25 console.log(vm.run(code));

```

Listing 2.3: Sandbox Escape in vm2@3.9.19 via Promise

Sandbox Escape via Custom Inspect Function [45]

Node.js allows a custom inspect function to be used instead of the default formatter by defining it as `util.inspect.custom` property. This mechanism enables developers to override the default behavior of `util.inspect()`. As a result, when Node.js tries to inspect an object (for example, during error handling) it will invoke the custom inspect function if present. However, if this overridden function receives an `inspect()` callback that originates from the host environment, it can be abused.

In the context of this exploit, the attacker began by identifying internal use cases of the `inspect()` function within the Node.js codebase. This revealed multiple occurrences, particularly within the `determineSpecificType()` function [48]. This segment is reachable by various types of errors, for example `ERR_INVALID_ARG_TYPE` and `ERR_INVALID_ARG_VALUE`.

Additionally, the error `ERR_INVALID_ARG_TYPE` is thrown within the `wasmStreamingCallback()` function [49], which can be triggered via `WebAssembly.compileStreaming()`.

Listing 2.4 illustrates this exploit in practice. In this example, the attacker creates an object with a custom inspect function specifically designated to execute arbitrary code. When the code calls `WebAssembly.compileStreaming()` with this crafted object, it triggers a `ERR_INVALID_ARG_TYPE` error. This error leads the execution flow to the `determineSpecificType()` function, which attempts to inspect the object. As a result, the custom inspect function is invoked, and the malicious code is executed.

```
1 const {VM} = require("vm2");
2 const vm = new VM();
3
4 const code = `
5     const customInspectSymbol = Symbol.for('nodejs.util.inspect.custom');
6
7     obj = {
8         [customInspectSymbol]: (depth, opt, inspect) => {
9             inspect.constructor('return process')().mainModule.require('
10                 child_process').execSync('touch pwned');
11         },
12         valueOf: undefined,
13         constructor: undefined,
14     }
15
16     WebAssembly.compileStreaming(obj).catch(()=>{});
17 `;
18 vm.run(code);
```

Listing 2.4: Sandbox Escape in vm2@3.9.19 via custom inspect function

In conclusion, the HubOS system relies on the execution of scripts in the form of modules. These scripts can be deployed directly by the host user or provided by a third-party developer. In the latter case, a crucial question arises: how can we ensure the user's security against potentially malicious scripts designed to carry out harmful actions on the host machine?

The original contributors addressed this concern by proposing the isolated execution of scripts within a sandbox using the library VM2. However, following the discovery of critical vulnerabilities in the chosen solution and its discontinuation, it has become essential to rethink this approach and adopt an alternative solution that addresses at least the same security concerns.

We reserve the discussion of the different alternatives to this library for the following chapter.

2.2.3 Communication Flows

In this section, we address the implementation of the communication buses (event and stream) as well as the network client.

Event and Stream Bus

The **event bus** and the flow **bus** are both implemented over the Mosquitto publish/subscribe broker (as used also by OpenHAB, and implementing the MQTT protocol that is common in smart environments). In our case, most communications occur through Mosquitto channels, including those between OpenHAB and HubOS, and between HubOS modules of the same application.

Mosquitto [50] is an open-source broker widely used in home automation systems and IoT applications for its ability to efficiently handle a high volume of real-time messages while maintaining a minimal memory footprint.

MQTT Protocol The Message Queuing Telemetry Transport (MQTT) protocol is a lightweight messaging protocol primarily used for communication between devices in IoT environments. It relies on a publish/subscribe model, where "subscribers" **subscribe** to **topics** and receive messages **published** by "publishers".

Topics in MQTT are channels used to organize communication between clients on a MQTT broker. Each message published in MQTT is associated with a topic, which acts as a communication channel. Topics are hierarchical, allowing for better organization of data flows. For example, `home/livingroom/humidity` might define the topic that receives and displays the humidity value of the living room in the house. Similarly, `home/livingroom/temperature` can define the topic that receives and shows the temperature level of the room.

MQTT is designed to be simple, and reliable. It operates using a central server called a **broker**, which manages communication between clients and ensures that messages are delivered correctly.

Flow (or Stream) Bus To prevent direct interactions between the modules and the devices or sensors, both rely on suitable drivers. HubOS leverages OpenHAB's driver support, requiring only minimal modifications to integrate with its APIs. For example, camera drivers stream video through the stream bus, preventing direct access by modules. In addition, drivers enable the transmission of commands to devices via HubOS's event bus.

For example, referring to the example of the smart door camera application presented in section 1.2.2 and illustrated in Figure 2.4, the service module continuously analyzes a video stream and generates events for the handler module whenever an unknown face is detected. If permitted by the rules, the handler module may then access the video stream and forwards it to an external service.

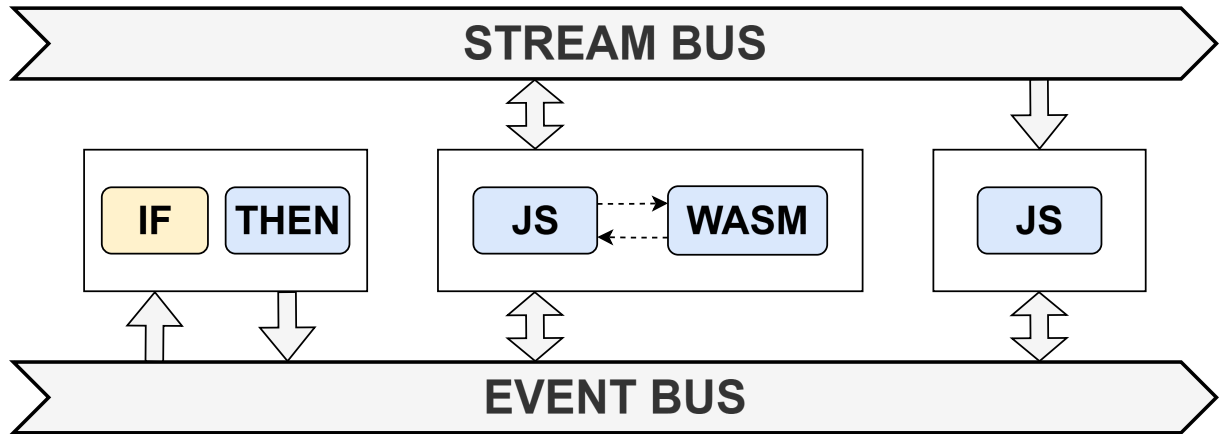


Figure 2.4: Flow and Event Bus communication between application processes [5]

Data Stream over MQTT Sending continuous data streams, such as video, over the MQTT protocol is uncommon, as MQTT was not originally designed for this purpose. Consequently, there is no definitive answer regarding its impact on performance of either the broker or the clients when sending or receiving messages. Therefore, one could consider maintaining this approach. However, as discussed in a later section, we adopt a more traditional method of access using TCP-based requests, leveraging OpenHAB’s capabilities.

Access Control In the current HubOS configuration, the developer does not provide any access control for the MQTT communication flow. This means that a Module has no restrictions on read or write access to the available topics. Consequently, the module can publish to topics reserved for other modules and read information transmitted over the stream bus, which is particularly concerning since the communication is neither secured nor encrypted.

Network Client

Access to external resources by modules is restricted by default and delegated to the `NetworkClient`. As its name suggests, this client defined by HubOS is attached to the module’s sandbox instance before execution. It acts as a middleware, in the sense that the client provides modules with several functions enabling them to perform GET, POST, PUT, and other types of requests, and returns the result of the executed request to the module. However, the current implementation of the client does not verify whether the destination URL belongs to a domain authorized by any of the defined rules. As a result, a module can communicate with any web service without restriction.

2.2.4 An Early Prototype

We analyzed the first HubOS prototype developed by the first team. This analysis allowed us to understand the underlying philosophy of HubOS, along with its vari-

ous components and mechanisms forming the foundation of the system. However, certain aspects of this version require significant revision or redesign. Despite the remarkable work done by the initial contributors on the development of HubOS, it is more appropriate to consider this version as a prototype, or rather a proof-of-concept based on the three use cases introduced earlier (Front Door Smart Security Camera, Voice-Controlled Butler, and Fall Detection and Remote Assistance).

In this section, we discuss three elements of HubOS that demonstrate the need for further refinement and redesign.

Static References

First, alongside the development of HubOS itself, three applications were also built by the initial HubOS team to demonstrate practical use cases for the platform. As a result, some values and references were hard-coded directly into the project's source code, which can be particularly striking:

Fixed device type definitions When extracting data from the `Manifest.json` in the `extractManifestData()` function (located in `system/core/AppRuntime.js`, see Listing 2.5), the developer restricted support to only cameras, lights, thermostats, and voice assistants, significantly limiting interaction possibilities.

```
1 module.exports = class AppRuntime {
2   ...
3   async extractManifestData(appDir,appName) {
4     ...
5     let vm = new NodeVM({
6       ...
7       sandbox: {
8         ...
9         Camera : dependencies.includes("Camera") ? new Camera("service") :
10            null,
11         Thermostat : dependencies.includes("Thermostat") ? new Thermostat("
12            service") : null,
13       ...
14     }
15   }
16   ...
17   processEvent(eventType,event) {
18     ...
19     actions.forEach(action => {
20       if (action['access']=='NetworkClient') {
21         sandbox['NetworkClient'] = new NetworkClient(action);
22       }
23       ...
24       if (action['access']=='Thermostat') {
25         sandbox['Thermostat'] = new Thermostat(action);
26       }
27     }
28   }
29   ...
30 }
```

Listing 2.5: Source code of HubOS showing the fixed device type definitions

Static MQTT topic subscription When initializing the EventEngine class (in `system/core/EventEngine.js`, see Listing 2.6), the system subscribes exclusively to the topics used by the three use cases (FaceRecognitionEvent, FallDetectionEvent, and SpeechRecognitionEvent).

```
1 module.exports = class EventEngine {
2   constructor(appRuntime) {
3     ...
4     this.client.subscribe('FaceRecognitionEvent', (err) => {
5       ...
6     })
7
8     this.client.subscribe('FallDetectionEvent', (err) => {
9       ...
10    })
11    ...
12  }
```

Listing 2.6: Source code of HubOS showing the static MQTT topic subscription

Consequently, this design results in two major issues:

1. It is impossible for any application to interact with devices beyond the four predefined categories, such as smart locks or connected power outlets.
2. HubOS is inherently incompatible with applications other than the three original ones, as it does not subscribe to their respective MQTT topics.

These limitations significantly limit the system’s scalability and prevent smooth integration of future applications, restricting the overall adaptability of HubOS to evolving IoT environments.

Two Modules Limitation

Then, the application presented and their implementation within HubOS are limited to only two modules (the Service module and the Handler module). In addition, the TABAC rule logic is also restricted to these two modules, as the composition of a rule always results in actions being assigned to the Service module. While there is nothing inherently wrong with this design choice, it introduces limitations with respect to the principle of separation of duties that HubOS aims to promote. Indeed, HubOS is intended to support the definition of multiple modules within the same application. Therefore, it is necessary to revisit this aspect, not only to enable the execution of more than two modules, but also to specify, when creating TABAC rules, which module should be targeted by each action.

Limited Integration of OpenHAB and its Rule Engine

After that, as mentioned in the previous research, HubOS was originally intended to delegate TABAC rule management to OpenHAB’s engine, thereby leveraging its mature trigger-handling model. Indeed, OpenHAB, through its rule engine, provides a powerful and flexible rule management framework that reinforces the principle

of separation of duties. Instead, the current version of HubOS re-implements this functionality internally. While this is not an error, it raises questions regarding architectural consistency and potential duplication of existing, well-tested capabilities.

No Permissions Management

Finally, an important aspect to highlight for the continuation of this project is the way HubOS handles the assignment of access permissions to various resources (both web based and hardware-related). As previously discussed, HubOS relies on its custom drivers to enable its VM2 instances to access hardware resources such as video or audio streams.

In addition to this, in order to allow instances to access resources available on the web, HubOS provides a dedicated interface for web communication. This interface offers full access to external web resources and is assigned to the instance at runtime, remaining active until the end of the instance's lifecycle.

However, this approach introduces a critical limitation: it does not guarantee that modules will interact only with trusted or authorized web resources. In its current form, this mechanism resembles an open door left unattended, it allows traffic to flow freely in and out, but provides no visibility or control over what is actually being exchanged. As a result, it becomes nearly impossible to monitor or restrict undesired behaviors.

Therefore, it is essential to rethink this model and implement a more complete permission system, one that only authorizes access to resources when explicitly allowed based on the destination of the request. This would enable HubOS to enforce strict outbound communication policies and significantly reduce the system's exposure to malicious or unintended interactions.

2.2.5 Need for Improvement

Throughout this section, we presented the first implementation of HubOS. This allowed us to highlight that some of its components require improvements or updates to better align with the intended architecture and security principles. These improvements include, among others, an alternative to the sandboxing module, the implementation of access control at the MQTT broker level, the modification of the access control model for web services, and the delegation of TABAC rule management to OpenHAB. Additionally, we also note that this version of HubOS is more comparable to a proof-of-concept than to a production-ready release. Continuing development on this version would be of little value, as it would still require a complete re-implementation of certain components. Consequently, we will revisit the implementation with a new version designed to meet the newly identified requirements and be suitable for production deployment. The following chapter addresses this redesign implementation and its key architectural decisions.

Chapter 3

New HubOS: Improved Architecture, Features, and Security Model

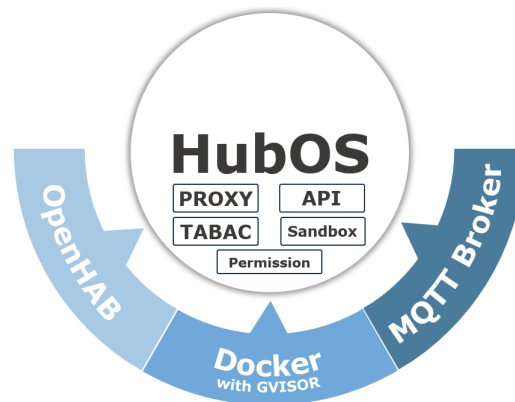


Figure 3.1: New HubOS main components

This new implementation of HubOS represents a complete redesign of the system architecture. For clarity throughout this document, we will refer to this redesigned version as "**New HubOS**", distinguishing it from the initial proof-of-concept implementation.

As discussed in the previous part of this report, a system like HubOS must meet several essential requirements. First and foremost, HubOS is designed to allow its users to create or import applications that serve a specific purpose. The execution of these applications must be carried out securely, within a sandboxed environment that strictly prevents any form of unauthorized escape or interaction with the host system.

In addition, it is crucial that the entire HubOS platform be developed according to a privacy-by-design approach. It should be inherently difficult for a malicious actor to develop an application capable of altering or compromising the user's environment once imported into HubOS.

This chapter presents the modifications and newly added features that distinguish the updated version of HubOS from the original project. It details the ar-

chitectural and functional enhancements implemented, with particular emphasis on the security improvements they introduce. These developments contribute to a more resilient, scalable, and secure system, better aligned with the requirements of a modern and privacy-conscious application platform.

The objective of this redesign is structured around three main pillars, First, to address the issues highlighted in the previous analysis; Second, to develop New HubOS with a strong emphasis on maximizing security through architectural and functional improvements; and finally, to realize the author’s vision by delivering a production-ready version of HubOS that is both robust and privacy-by-design.

While New HubOS introduces major evolutions, it retains a structure that remains largely consistent with its predecessor. As shown in Figure 3.1, The system is still based on three main entities, the New HubOS core system, OpenHAB, and the MQTT Broker. However, a fourth key component has now been integrated into the architecture: Docker, together with its gVisor layer.

The core system of New HubOS is composed of several distinct services, each responsible for a specific task within the system. The following sections provide an in-depth description of these components:

- The App Manager
- The MQTT Broker with Access Control
- The Sandbox Manager
- The TABAC Manager
- The Proxy Service
- The Permission Manager
- The REST API Service
- The New Communication Flows

3.1 App Manager

This section presents the App Manager, the controller responsible for verifying the integrity and validity of imported applications during the configuration phase. When a user imports an application from a third party, it is essential to perform an initial verification step. This process checks for correct naming conventions, proper file formats and structures, as well as the expected directory layout.

3.1.1 Application Integrity Verification

The goal of this verification is not only to ensure that the modules will execute correctly within the system, but also to reduce the potential attack surface by rejecting improperly structured or potentially malicious applications.

The first verification step consists in ensuring that the application's file structure is correctly built. As shown in Figure 3.2, the App Manager checks whether the application contains all required files and directories. Specifically, it verifies the presence of a **config.json**, a **manifest.json**, and a **rules.json** file within the **tabac-rules/** directory, as well as a **modules/** directory containing each module in its own subdirectory.

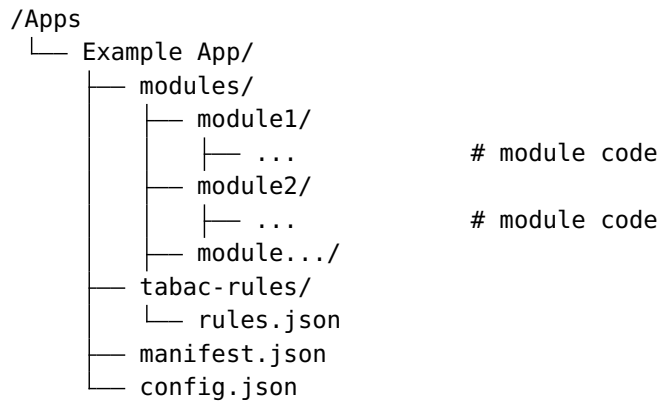


Figure 3.2: File structure of an App with New HubOS

The second verification step focuses on validating the internal composition of the **manifests.json**, **rules.json**, and **config.json** files. This verification serves two main purposes. First, it ensures that rules and other elements are properly defined in order to prevent any functional errors and to alert the users in case a required field is missing or incorrectly defined. Second, and more subtly, it aims to reduce the attack surface of New HubOS.

Indeed, New HubOS needs to read certain user-provided files to function correctly. However, because of these files include user-defined input, it is critical to ensure that they do not contain malicious content that could compromise the system or its environment.

One concrete example of vulnerability we aim to prevent is **path traversal** through the file system hierarchy. New HubOS identifies application modules by reading the **manifest.json** file, which lists the names of all modules located in the **<example_of_app>/modules/** directory, as shown in Listing 3.1. Then using this list, New HubOS iterates over each module name, constructs the expected file path (e.g. **hubos/apps/example_of_app/modules/module1/**), opens the corresponding folder, and executes the module's code inside a Docker container.

```

1 {
2   "name": "appName",
3   "description": "little description of the app",
4   "type": "app type",
5   "modules": [
6     {
7       "name": "module1Name",
8       "type": "service"
9     },
10    {
11      "name": "module2Name",
12      "type": "handler"
13    }
14  ]
15 }

```

Listing 3.1: Simple example of a manifest.json with New HubOS

This behavior could potentially be exploited by a malicious user who manipulates the path to execute unintended or hidden content. For instance, as demonstrated in Listing 3.2, an attacker could craft a module name that resolves to a deeply nested or obfuscated path (e.g., `hubos/apps/example_of_app/modules/module2/hidden/hidden2/maliciousDockerCode/`) leading to the execution of a Docker image Different from the one clearly visible and initially declared.

```

1 {
2   "name": "appName",
3   "description": "little description of the app",
4   "type": "app type",
5   "modules": [
6     {
7       "name": "module1",
8       "type": "service",
9       "description": "small description of module1"
10    },
11    {
12      "name": "module2/hidden/hidden2/maliciousDockerCode",
13      "type": "handler",
14      "description": "small description of module2, not malicious at all"
15    },
16    {
17      "name": "module3",
18      "type": "handler",
19      "description": "small description of module3"
20    }
21  ]
22 }

```

Listing 3.2: simple example of a malicious manifest.json

Consequently, to prevent such issues, the format of the fields within these JSON files is intentionally strict. Only safe and necessary characters are allowed, in order to limit the potential for malicious payloads and ensure the integrity of the system and execution process.

In addition, since certain data extracted from these JSON files are stored in the database, New HubOS also implements protection mechanisms against **SQL injection** attacks. All database operations are performed using prepared statements, which safely separate SQL commands from user-provided values. This approach ensures that input data are properly escaped and cannot be interpreted as executable SQL code, thereby reinforcing the security and reliability of database interactions.

3.1.2 User Agreement

A final verification performed by the App Manager involves obtaining the explicit agreement of the user importing the application. During the validation process, New HubOS prompts the user to carefully review the `config.json`, `manifest.json`, and `rules.json` files in order to understand which resources the application will require. This step ensures that the user is fully aware of the application's scope and behavior, and that they give their informed consent before the application and its modules are allowed to execute.

3.2 MQTT Broker with Access Control

One of the first improvements made to the system is the addition of an access control mechanism for MQTT communications. As previously identified, the original version of HubOS lacked such mechanism, resulting in all information exchanged over a topic being freely accessible. Consequently, it is essential to implement a system that restricts communication between two entities, ensuring that no third party can access the exchanged data without proper authorization. To prevent this, two possible solutions were considered.

The first option involves the use of cryptographic techniques to encrypt and sign messages transmitted over a topic. This would prevent unauthorized applications from accessing sensitive data, and would also provide message authenticity by ensuring that a malicious application cannot impersonate another. However, this solution is not fully effective on its own. It does not prevent an unauthorized application from publishing messages topics it should not have access to.

For this reason, we opted for the second solution, implementing access control directly within the MQTT broker. Fortunately, the Mosquitto broker already used in HubOS supports this through the **Dynamic Security Plugin** [51], which allows for flexible management of user access rights and enhances the security of our system by controlling connections and access permissions.

User creation and role assignment are performed through a dedicated MQTT topic that is only accessible to administrative entities. This ensures that no regular module or application can modify access control policies or impersonate other clients. During the module build process within Docker, New HubOS automatically generate fresh credentials for each new container instance. These credentials are securely provisioned and allow the module to authenticate with the broker.

By default, these identities are associated with restrictive roles that deny read and write to all MQTT topics. Permissions are then explicitly granted by New Hu-

bOS for specific topic relevant to the module’s operation. This design ensures that only the intended module, HubOS core services, and OpenHAB can access the associated MQTT channels, effectively enforcing the principle of least privilege and minimizing the attack surface.

3.3 Sandbox Manager

In the previous chapter, which discussed the depreciation of VM2 library and, consequently, its unsuitability for use in our new implementation. Therefore, it is necessary to find an alternative to this library. This section covers three possible alternatives to the sandboxing module and describes the implementation of the selected one within the **Sandbox Manager**.

In the context of HubOS, it is essential that the execution of modules and their scripts occurs within a secure and isolated environment. Indeed, a robust sandbox solution is required to ensure that any potentially malicious script remains confined and cannot escape its isolation.

At the same time, we aim to support a solution that allows application developers to import external modules, in order to enable the execution of more diverse and complex tasks. For instance, the ability to use AI-related libraries could facilitate advanced features such as fall detection or unknown face recognition, echoing the examples introduced earlier in this work.

Additionally, it would be highly beneficial to support a broader range of programming languages for module development. Indeed, VM2 is inherently limited to JavaScript and WebAssembly due to its reliance on the open-source V8 engine [52]. Overcoming this limitation would not only expand the range of supported use cases but also enable the execution of more complex code that may require greater computational resources.

3.3.1 Sandboxing Alternatives

In this section, we present three alternative solutions to replace VM2, each aiming to ensure the secure execution of user-developed applications within HubOS. These alternatives are evaluated based on their ability to provide strict isolation, prevent sandbox escapes, support programming languages beyond JavaScript and WebAssembly, and offer long-term maintenance and reliability.

Node.js | VM [53]

Despite the discontinuation of vm2, it would be possible to develop a library that utilizes and enhances the Node.js’s core vm module to securely execute sandboxes while maintaining the features of HubOS.

The vm core module in Node.js provides an API to execute Javascript code within a sandboxed environment in a separate V8 execution context. Enabling the creation of isolated execution contexts or preventing scripts from accessing the host. However, its security is not foolproof, as objects can still reference external resources unless properly controlled.

Vm2 aimed to offer better security and isolation than the built-in vm module, especially during the period when it was actively maintained and aligned with its original design goal. Therefore, considering the development of a more secure vm module, similar to what was done with vm2, would be a viable option. However, this approach would be complex and would require specialized knowledge and a significant amount of development time.

Isolated VM [54]

Isolated-vm is a Node.js module designed to securely execute untrusted JavaScript code in a completely isolated environment. It provides a high level of isolation compared to built-in vm and other alternatives such as vm2.

The main features are:

- Deep isolation: Each script runs in separate v8 instance preventing any access to the main Node.js context.
- Strict resource control: Allows setting memory and execution time limits.
- Flexible execution: Support of both synchronous and asynchronous execution.

However, due to its isolated nature, it will be impossible to execute applications relying on external modules. As noted by the project maintainer (*Laverdet*), each sandbox (due to its isolated nature) operates with its own environment and memory heap. This means that a script running inside an isolated VM cannot directly access external modules such as axios, mongoose, fetch, or ether.js. To overcome this limitation, one approach is to just pass the code for the function directly into the isolate and execute it there. This method works well for self-contained functions that do not rely on external resources like the file system or network. However, if we need to perform operations on files, network, native modules, etc., Then we will need to setup an intermediary (often called a "shim"). This shim handles the sensitive operations on behalf of the sandbox and returns the results. Although it offers strong isolation guarantees, its major limitation lies in the inability to allow the importation of external modules, which is a critical requirement for our application.

Gvisor [55]

In the previous sections, we reviewed various alternative modules to vm2. However, none of them fully meet exactly our needs. So why not explore an alternative outside the scope of the Node.js environment?

In this section, we introduce an alternative using gVisor, an application sandboxing solution developed by Google.

gVisor is an application kernel that provides strong isolation between running applications and the host OS. It includes an Open Container Initiative (OCI) runtime called **runsc** that makes it easy to work with existing container tooling such as Docker or Kubernetes.

According to Espe et al. [56], The Open Container Initiative (OCI) is a Linux Foundation project to design open standards for operating system level virtualization, most importantly Linux containers.

Compared to traditional virtual machines, it offers many of their security benefits while maintaining a lower resource footprint, faster startup, and greater flexibility. It limits the host kernel surface accessible to the application while still giving the application access to all the excepted features.

as shown in Figure 3.3, gVisor is composed of two keys element:

Sentry As described by Anjali et al. [57] and shown in Figure 3.3, Google gVisor is a sandbox container runtime that uses paravirtualization to isolate containerized applications from the host system without the heavy-weight resource allocation that comes with full virtual machines . It implements a user space kernel, Sentry that runs in a restricted seccomp container. All syscalls made by the application are redirected into the Sentry. This prevents the application from having any direct interaction with the host through syscalls. Two methods of syscall redirection are supported: **ptrace-mode** uses ptrace in the Linux kernel to forward syscalls to the sentry, and **KVM-mode** uses KVM to trap syscalls before they hit the Linux kernel so they can be forwarded to the sentry.

Gofer The Gofer is a standard host process which is started with each container and communicates with the Sentry via a socker or shared memory channel. The Sentry process is started in a restricted container without access to file system resources. The Gofer mediates all access to these resources, providing an additional level of isolation.

Additionally, gVisor has its own user-space networking stack called netstack. The Sentry uses netstack to handle almost all networking, including TCP connection state, control messages, and packet assembly.

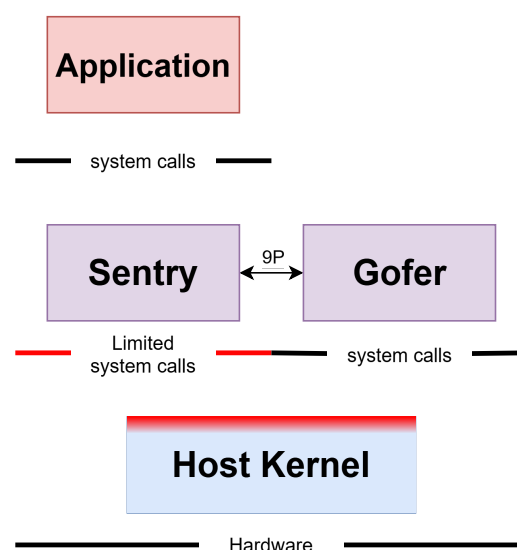


Figure 3.3: The architecture of gVisor as presented in the official gVisor documentation [58]

Choosing this solution would require rethinking part of the HubOS architecture. Currently, HubOS relies on a Node.js library, which greatly simplifies its sandbox implementation. However, since gVisor runs at the host level, certain modifications will be necessary. In particular, a new library would need to be developed to facilitate communication with gVisor and enable application execution. In other words, for each application module, a new instance of gVisor will be executed with its assigned module, and communication with HubOS will be handled via the MQTT protocol.

This approach offers a clear advantage: it removes the limitation of running only Javascript or WebAssembly applications, allowing execution of application modules written in a variety of programming languages (with the added benefit of supporting external library imports).

Advantages and Disadvantages

Throughout the three approaches presented in this section, we have identified both advantages and disadvantages for each proposed sandboxing solution.

Node.js vm This first approach will not be considered further. Developing a custom sandboxing module based on Node.js's core vm module, while adding isolation capabilities and support for external module imports, would require significant effort and complexity. In addition, the subsequent solutions we explored offer more practical and efficient ways to address our sandboxing needs.

Isolated-vm This option is much more feasible than the previous one. However, the fact that the sandboxing module does not support importing external modules significantly limits the development possibilities. As a result, this approach will also not be selected.

gVisor This option is the one that best meets our needs. It enables secure application isolation, allows the import of external modules (e.g., in the case of Node.js applications), and supports a wider range of programming languages. Although implementing a communication module to interact with the gVisor service on the host machine requires a lot of work, the workload and complexity involved remain significantly lower than what would be required for the first approach. Additionally, one of the core objectives of gVisor is to enable the safe execution of untrusted code. This makes its adoption particularly compelling compared to other sandboxing alternatives, as it provides an additional layer of isolation specifically designed to mitigate risks associated with potentially malicious or compromised applications.

Consequently, as illustrated in Table 3.1, we have outlined the advantages and limitations of each alternative. Among them, the solution that stands out is the one leveraging Docker combined with gVisor as the sandboxing mechanism. The next section presents the implementation details of this approach within the new HubOS architecture.

Option	Secure	Isolated	Module Support	Supported Languages	Complexity of Implementation
Self-made vm2 using Node.js vm	Yes	Yes	Yes	Javascript + WebAssembly	High
Isolated-vm	Yes	Yes	No	Javascript + WebAssembly	Mid-High
gVisor	Yes	Yes	Yes	No limits	Low-Mid

Table 3.1: Comparison of sandboxing and isolation modules

3.3.2 Docker with GVisor

As previously mentioned, gVisor can be used either as a standalone sandboxing tool or in conjunction with a container tool such as Kubernetes or Docker. In this project, we have chosen to integrate gVisor within the Docker environment for simplicity and compatibility reasons. Consequently, the **Sandbox Manager** will be responsible for managing and executing application modules within Docker containers, using gVisor as an additional isolation layer to enhance security.

Rad et al. [59] describe Docker as an open-source platform that allows developers to automate the deployment and execution of applications within lightweight, portable container. A Docker container provides an isolated environment running on top of the operating system’s kernel, without the need to virtualize an entire hardware stack. Compared to traditional virtual machines, this approach significantly reduces overhead, resulting in faster startup times and lower resource consumption.

Moreover, Docker gives developers the flexibility to implement modules that compose an application in the programming language of their choice. These modules can then be packaged as Docker images and made publicly available, allowing other users to import them seamlessly into their own New HubOS instances.

Currently, Docker has an HTTP API called the **Docker Engine API** [60]. This is an API service that the client can use to communicate with the Docker Engine. So, everything the Docker client can do can be done with the API.

In this context, New HubOS leverages a Node.js library called **Dockerode** [61], which offers a convenient interface to interact with the Docker Engine via its API. This integration allows the system to programmatically manage containers life cycle events such as image building, container instantiation, and runtime configuration.

By default, Docker uses **runc** as its container runtime. However, to take advantage of the additional security features provided by gVisor, it is necessary to explicitly set the runtime to **runc**, gVisor’s user-space kernel implementation. This configuration ensures that application modules launched within Docker containers benefit from the strong process isolation and syscall filtering offered by gVisor.

3.4 TABAC Manager

In the previous chapter, we discussed the relevance of delegating the management of triggers and their corresponding actions to the OpenHAB rule engine. This choice promotes the principle of separation of duties while leveraging an existing, well-tested rule management system that already incorporates its own security mechanisms and proven capabilities. In addition, it enables the implementation of rules that can directly interact with the state of devices connected to OpenHAB, thereby facilitating seamless integration and real-time responsiveness within the system. However, a constraint arises: the TABAC rule model is not directly compatible with the OpenHAB's rule model. Consequently, it is necessary to implement, within HubOS, a dedicated system responsible for converting TABAC rules into rules accepted by OpenHAB. This system has been named the **TABAC Manager**.

In this section, we present the logic behind the transformation process from TABAC rules to OpenHAB's TAR format, as well as the structure of the resulting TABAC rule files.

Before discussing the transformation process itself, it is important to first present the new structure of the associated configuration file and the TABAC rule file.

3.4.1 Configuration File

The configuration file is a simple **.json** file, in which the developer of the application specifies the variables that the user must complete to ensure the proper functioning of the application. As shown in Listing 3.3, the configuration file is composed of three main sections:

The first section specifies all the devices that the application needs and that the user must configure for it to work. On HubOs's side, only the device name and its OpenHAB UID are required. In this way, by referencing the device name, it will be possible to retrieve its associated UID.

The second section concerns external servers and domain names. In this section, the user can specify the IP addresses, port numbers, or domain names that the application will need to access during runtime.

The last section is designed to define any additional information the application may require that does not fall under the two first categories. For example, it may include empty fields intended to receive API tokens or other runtime parameters necessary for interacting with external services.

This design choice was made to centralize all the resources that the application will interact with during its execution. As a result, users can easily understand which physical devices and web services are involved simply by inspecting the configuration file. Furthermore, TABAC rules must explicitly reference these declared resources in order to gain access to them. This ensures that no rule can request access to undeclared or unauthorized devices or services, thereby reinforcing the security and transparency of the system.

```

1 {
2   "configuration":{
3     "devices" : {
4       "myCamera" : {
5         "UID": "xxxx",
6         "type": "camera",
7         "description": "Fall detection camera description - please insert
                        the camera OpenHAB UID"
8       },
9       "mySmarSpeaker" : {
10        "UID": "xxxx",
11        "type": "device",
12        "description" : "smart speaker device description - please insert
                        the smart speaker openhab UID"
13      }
14    },
15    "servers" : {
16      "myServer" : {
17        "host" : "0.0.0.0:1234",
18        "description" : "server description ..."
19      },
20      "myServer2" : {
21        "host" : "api.example.org:443",
22        "description" : "api server to contact for ..."
23      }
24    },
25    "others":{
26      "myServer2APIToken":{
27        "token": "xxxx:xx-xxxxxxx",
28        "chatId": "xxxxxxxx",
29        "description": "myServer2 API token"
30      }
31    }
32  }
33 }

```

Listing 3.3: Simple example of a configuration file with New HubOS

3.4.2 TABAC-Rule File

The **rules.json** file define triggers as well as **access authorizations** to certain resources.

In terms of structure, it remains quite similar to the one proposed by OpenHAB in order to maintain consistency with its original product. The only difference being that, when an event is triggered, the subsequent action is already defined by the execution of the handler module's script. Therefore, this **rules.json** will not determine the action to be taken, but rather the required access for the execution of this module for that specific event. As a result, when a specific event is triggered, the rule will determine the permissions to be granted to the corresponding module, which will then be able to perform the necessary actions.

The structure of New HubOS rule file consists of a list of TABAC rules. Each rule

is structured into three parts: the trigger (**when**), the conditions (**condition**), and the authorizations (**then**). The example shown in Listing 3.4 illustrates an example of a TABAC rule that will be used as reference throughout the next three sections.

```
1  [
2    {
3      "name": "rule_example_1",
4      "description": "rule example contains any",
5      "when": {
6        "event": "device.mySmarSpeaker",
7        "context": "updated",
8        "value": "event1"
9      },
10     "condition": [
11       {
12         "name": "if_in_the_morning",
13         "description": "if the event occur in the morning",
14         "if": {
15           "event": "system.time",
16           "context": "between",
17           "value": ["06:00", "10:00"]
18         }
19       }
20     ],
21     "then": [
22       {
23         "access": "NetworkClient",
24         "type": "service",
25         "context": {
26           "period": "3600",
27           "host": ["myServer", "myServer2"],
28           "concern": "module2"
29         }
30       },
31       {
32         "access": "pass_through",
33         "type": "flow",
34         "context": {
35           "concern": "module2",
36           "value": "event1"
37         }
38       }
39     ]
40   }
41 ]
```

Listing 3.4: Simple example of a TABAC rule.json with HubOS

Trigger

The **when** section defines the event that will activate the TABAC rule. It requires three variables:

Event It correspond to the event source that will be analyzed to determine if the trigger meets the rule . Currently, there are two categories of event sources:

Those coming from an MQTT topic, defined by the prefix **mqtt** followed by the relevant topic; Those coming from OpenHAB devices, defined by the prefix **device** followed by the device name; and those coming from the system, such as time or date, defined by the prefix **system**.

Context & Value The context refers to the condition that the event value must satisfy in order to trigger the rule. These contexts can take various forms depending on the desired behavior. The most relevant and commonly used contexts in the TABAC rule model are summarized in Table 3.2:

A detailed list of available contexts examples is provided in Appendix A.1.

Context Type	Description
CHANGED	Triggers when the event source transitions from on specific value to another (e.g., from OFF to ON).
UPDATED	Triggers when the event source changes from any state to a specific value, regardless of its previous state.
CONTAINS	Triggers when the updated value contains the specified sub-string or pattern (useful for string-based states).
EQUALS	Triggers when the updated value exactly matches the target value.
HIGHER / HIGHER THAN	Triggers when the updated value is (strictly) greater than the specified treshold.
LOWER / LOWER THAN	Triggers when the updated value is (stricly) lower than the specified treshold.

Table 3.2: Overview of the main TABAC rule Contexts

Conditions

The **condition** section maintains the same foundations as the previous section. However, it does not activate the rule itself; Instead, it specifies additional conditions to be verified after the trigger event has occurred.

Its main difference lies in the possibility to define multiple distinct conditions connected by a logical "**AND**" operator.

Consequently, all conditions must be fulfilled for the next section to be applied

Actions

The "then" section specifies the actions to be executed once a rule has been triggered and its conditions validated.

This section grants the application, and more specifically to its modules, time-limited access rights to certain resources.

Currently, New HubOS supports three specific types of actions, as defined in Table 3.3. The first two are designed to grant and authorize access to certain resources, such as web services or physical devices connected to OpenHAB. The third type is used to deliver an alert to another module, with the purpose of task delegation within a multi-module application architecture.

A detailed list of available access types can be found in Appendix A.2.

Access Type	Description
Network Client Access	Allows access to the network client, and consequently to the external network, for instance, to communicate with a remote server.
Device Access	Grants direct access to an OpenHAB device (e.g. to adjust a thermostat's temperature).
Pass Through	Enables the forwarding of an alert from one module to another.

Table 3.3: Overview of the main TABAC rule Actions

Notice that when granting access to a device, the user need to specify its name as defined in the **config.json** configuration file. In this way, HubOS can determine the UID of the target device (unique to the user).

The Pass_through action type aligns with the principle of separation of duties, which is particularly important in the local-first model. For instance, consider the Front Door Smart Security Camera application presented in section 1.2.2. When an unknown face is detected, the system must send a notification to the owner along with a video snapshot.

To preserve modularity and minimize privilege escalation, this application is divided into multiple cooperating modules. A first module is responsible for performing face recognition by analyzing the video stream from the doorbell camera. When an unknown face is detected, it sends an alert via its assigned MQTT topic. This alert triggers a rule, which in turn redirects the event to another MQTT topic accessible by a second module.

This second module receives the alert and is granted temporary permissions to access the doorbell camera and the necessary web resources required to send the snapshot and the notification to the user.

An example of such rule is defined in Listing 3.5

```

1  [
2    {
3      "name": "unknown_face_detection",
4      "description": "when front door camera detect unkown face, call AlertModule
                        in charge of sending the notification + video snapshot",
5      "when": {
6        "event": "mqtt.faceRecognitionModule",
7        "context": "updated",
8        "value": "Unkown Face Detected"
9      },
10     "condition": [
11     ],
12     "then": [
13       {
14         "access": "NetworkClient",
15         "type": "service",
16         "context": {
17           "period" : "1800",
18           "host" : ["webService", "FrontDoorCamera"],
19           "concern": "AlertModule"
20         }
21       },
22       {
23         "access": "pass_through",
24         "type": "flow",
25         "context": {
26           "concern": "AlertModule",
27           "value": "Unkown Face Detected"
28         }
29       }
30     ]
31   ]

```

Listing 3.5: Example of an unkown face recognition TABAC rule

3.4.3 TABAC to TAR rule

The TABAC manager is a service responsible for monitoring the management of TABAC rules. In other words, one of the first steps when initializing a new HubOS application is to execute the TABAC Manager in order to extract the rules and the configuration defined by the developer and configure the OpenHAB Rule Engine, allowing the triggering of events and the execution of scripts when validated.

Figure 3.4 illustrates the process behind the TABAC manager. As mentioned earlier, the first phase of this controller is to determine the application's rules and authorizations. These rules define when and for how long a specific module of a given application is authorized to access a particular resource. The next step consists in extracting the relevant elements from the TABAC rule in order to construct a corresponding TAR rule that is compatible with the OpenHAB rule engine. Finally, the last step is to submit this TAR rule to OpenHAB through its API, allowing the rule to be registered and executed within the OpenHAB environment.

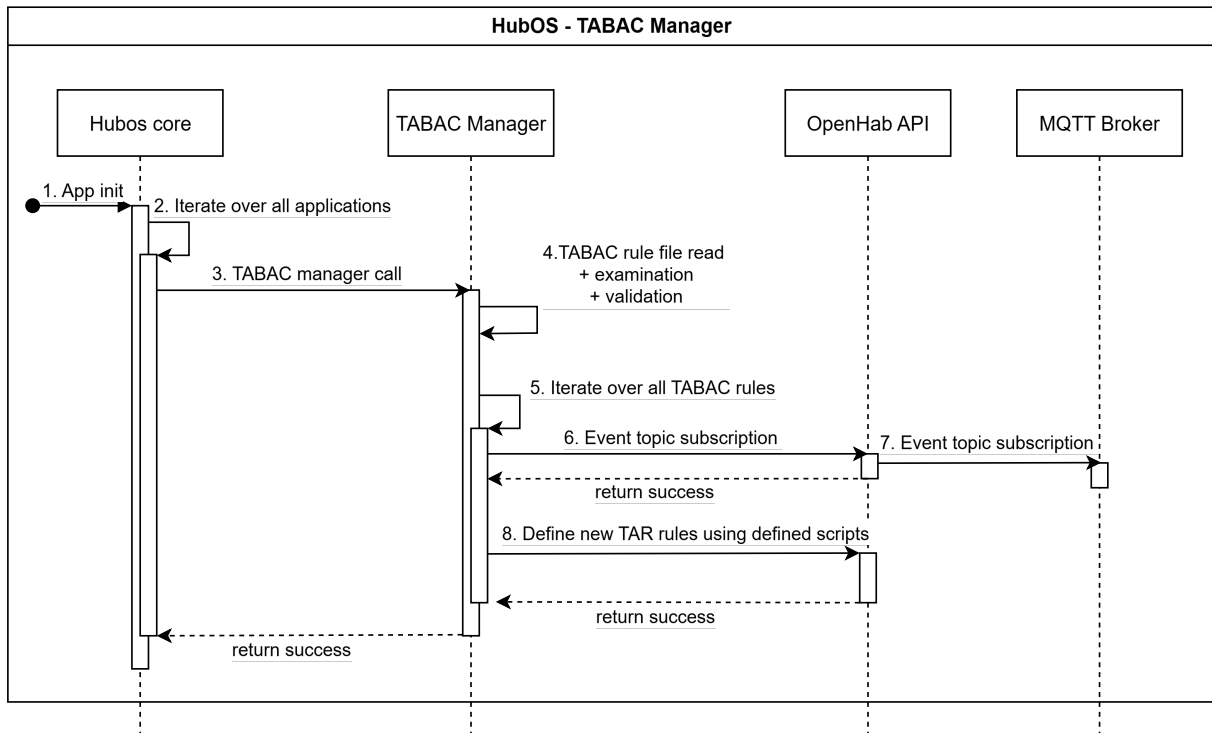


Figure 3.4: Simple example of the TABAC manager process

3.5 Resource Access Control

Throughout this chapter, we have presented how New HubOS executes applications using its Sandbox Manager, and how it applies TABAC rules to assign permissions to the relevant modules when a rule is triggered. This section will now describe how our system enforces these permissions and prevents modules running within Docker containers from accessing resources for which they are not authorized.

During the system's development, one of the first questions that arose was: "How can we restrict access to resources (OpenHAB/web) and only authorize it when explicitly allowed by the system?" As discussed in the previous chapter, the original version of HubOS exhibited a significant limitation in its handling of web access permissions, allowing unrestricted communication without consideration for the destination of outbound requests.

With New HubOS, we aim to implement a more comprehensive permission model. The new system is designed to deny all access to external resources by default and to require developers to explicitly define the resources a module is allowed to access, and when such access should be granted. In particular, any web resource that a module needs to communicate with must be clearly specified by the developer. This ensure that access is only granted to known destinations, thereby minimizing the risk of unauthorized interactions.

The first component of this new permission system is reflected in the definition of Network Client Access and Device Access, which are declared as part of the Actions section within TABAC rules. These declarations specify which modules are

granted access to which resources or hardware devices, and under what conditions.

In addition to this rule-based component, New HubOS also enforces several technical mechanisms to strengthen access control. These include a set of firewall rules applied to Docker containers, a proxy service responsible for mediating and filtering outbound web requests, and a permission management service that dynamically handles the granting and revocation of access rights. Each of these mechanisms will be explored in detail in the next three sections.

3.5.1 Proxy Service

One of the advantages of our predecessor, HubOS, was its use of VM2, a JavaScript-based sandboxing library that allowed for direct interaction between the NodeJS code (HubOS) and the VM2 instances it executed. This tight integration enabled seamless communication and control within the same runtime environment.

However, with New HubOS, we have deliberately opted for the use of Docker containers to isolate and manage application modules. This architectural change introduces a fundamental separation between New HubOS and its containers, making direct interaction more complex, as Docker operates as an independent runtime that communicates primarily through its Engine API.

As a result, unlike the original HubOS, we can no longer simply assign a communication interface to modules for direct interactions with the core system. In response to this challenge, two alternative solutions were considered:

REST API Middleware for Module Communication

The first option involves implementing a REST API service that acts as an intermediary between the Docker containers and external web resources. In this model, modules would send requests to the internal REST API, which would then process and forward the request to the appropriate external service, returning the response back to the module. More specifically, the REST API would be responsible for constructing the outgoing request itself based on the parameters received from the module (e.g., URL, headers, payload, method), waiting for the external response, and finally returning the result to the calling module.

This approach closely resembles the model used in the original HubOS and offers benefit of controlling and filtering requests based on their destination, which enhances security. However, it introduces several limitations. Most notably, since all traffic passes through the REST API, it can become a performance bottleneck, especially if multiple modules are communicating with external services simultaneously or if a malicious module attempts a DoS attack by flooding the service with requests. Furthermore, this solution would require considerable development effort to support various protocols and request types (e.g., GET, POST, RTPS, WebSockets, etc.).

A practical example of this configuration is illustrated in Figure 3.5.

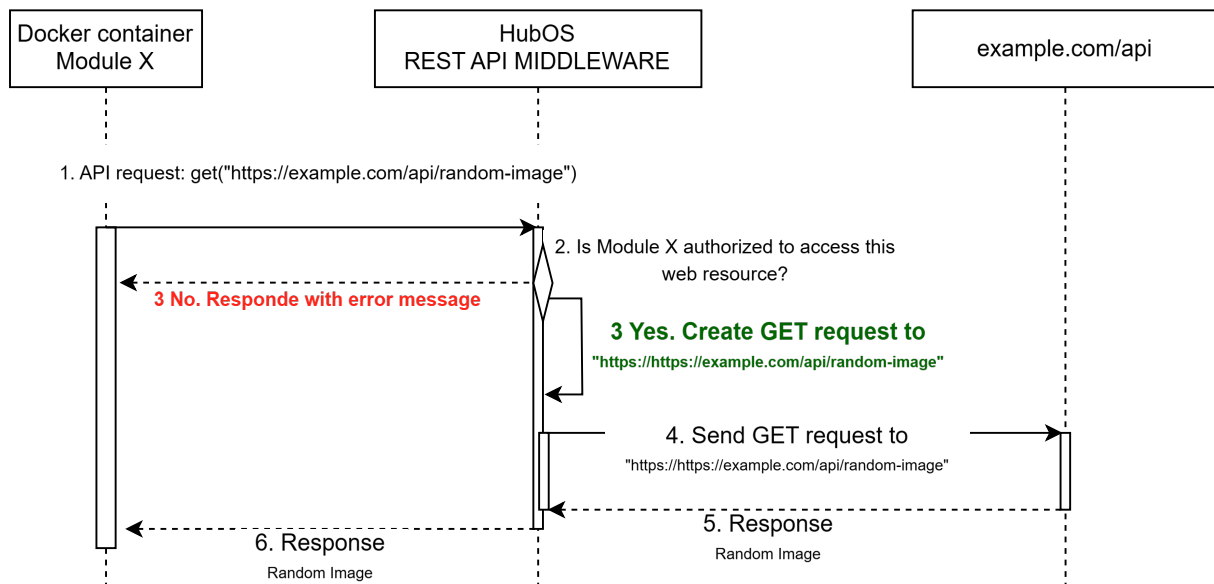


Figure 3.5: Example of a GET request sent by a module to an external web service through the REST API Middleware in order to retrieve a random image

Given these constraints, we opted for a simpler and more scalable solution: the use of a proxy service. Unlike the REST API approach, this proxy is designed to dynamically allow or deny requests based on both the sender (module) and the destination (external service). In this model, the computational load remains largely on the modules themselves, and the proxy acts as a filtering layer that enforces access control policies without being a central point of processing. This design preserves performance, reduces complexity, and aligns better with the modular and containerized architecture of New HubOS.

Forward Proxy

There are different types of proxies, each with its own specific use cases, advantages, and limitations. Among the most common are reverse proxies and forward proxies. A reverse proxy is typically used to manage and distribute incoming traffic to internal services, often providing benefits such as load balancing, and caching. In contrast, a forward proxy operates on the client side and is primarily used to control and filter outbound traffic from internal systems to external networks.

In the context of New HubOS, the forward proxy model is particularly relevant, as it enables control over which external web resources each application module is allowed to access. This section focuses on the design and integration of a forward proxy within New HubOS, and how it serves as a key components in enforcing the platform's permission model.

For the sake of simplicity, we opted to integrate an existing forward proxy solution named Straightforward Proxy [62]. This proxy is capable of intercepting HTTP, HTTPS, and Websocket request, which is more than sufficient for the needs of New HubOS. Its sole purpose is to intercept all outbound requests from the modules, identify the originating module, and decide whether the request should be forwarded based on the permissions granted at the moment of interception.

A practical example of this configuration is illustrated in Figure 3.6.

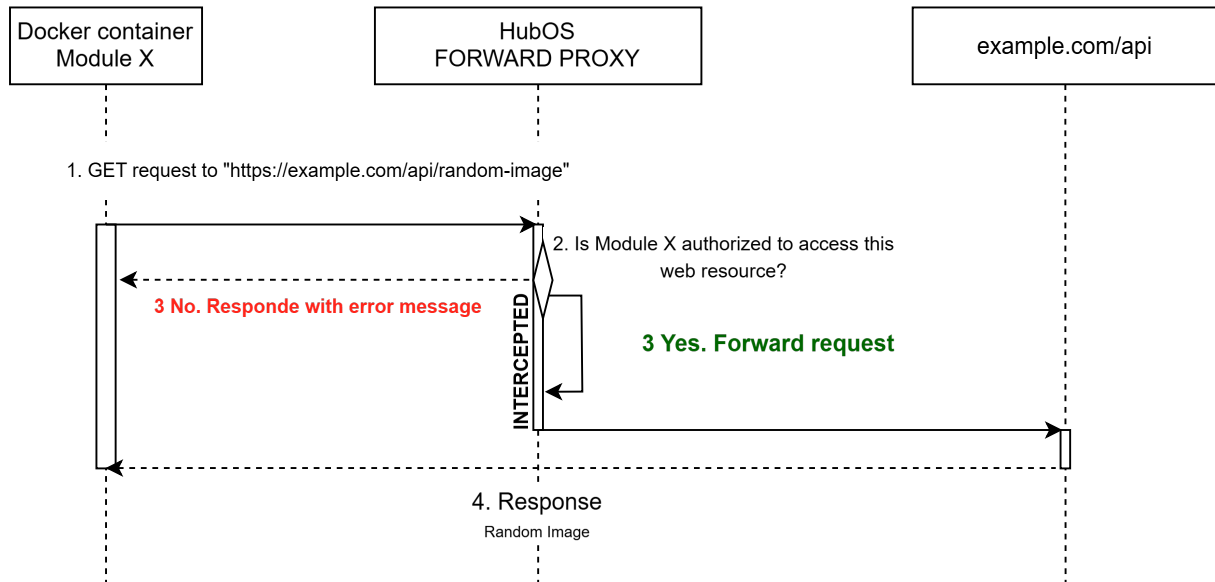


Figure 3.6: Example of a GET request sent by a module to an external web service in order to retrieve a random image, intercepted by the Forward Proxy

3.5.2 Proxy Authentication Header

To achieve the forward proxy, it is essential to reliably identify the source module of each outgoing request. One approach would be to associate each module with its container's internal IP address. In this case, when the proxy receives a request, it could determine the source module by inspecting the source IP, and then verify whether that module has the appropriate permissions to send the request.

An alternative approach involves adding an authentication mechanism directly within the proxy. In this model, each module is required to include a Proxy-Authorization header in its request. This header would contain the module's unique ID, along with a digital signature generated using its private key, following the format: `moduleId:signature(moduleId,privateKey)`. The corresponding public key, previously shared with New HubOS, would then be used to verify the identity of the module. This ensures that only authenticated modules are able to send outbound requests, and that these requests can be properly linked to their declared permissions.

We have chosen to implement this second solution, as it offers a method that is simpler to integrate within the current architecture of New HubOS. More importantly, it also prevents malicious modules to impersonating other modules in order to exploit their permissions, a critical aspect in preserving the integrity of the permission system.

3.5.3 Docker Firewall

The next component of our security system is the implementation of firewall rules designed to prevent Docker containers from bypassing the proxy service when making outbound requests. Conceptually, the goal is to build a protective wall around each container, allowing all external communication to flow through a single controlled gate: the forward proxy.

To enforce this policy, Docker provides a special iptables chain called `DOCKER-USER`, which acts as a placeholder for user-defined firewall rules and is processed before Docker's own default rules. By inserting our custom iptables rules into this chain, we can enforce strict network controls over the containers managed by New HubOS.

An example of the four essential firewall rules used for this setup is shown in Listing 3.6. To ensure that these rules do not interfere with other Docker containers unrelated to New HubOS, we created a dedicated Docker bridge network specifically for containers spawned by our system.

In summary, these firewall rules implement two key security principles:

1. **Only allow outbound traffic from containers to the proxy service** running on the same host, through the dedicated network interface.
2. **Reject all other outbound traffic** from containers, including any attempt to access localhost or to directly communicate with other containers on the same network.

```
1 Chain DOCKER-USER
2 -A DOCKER-USER -d 127.0.0.1/32 -i hubos_network_interface -j ACCEPT
3 -A DOCKER-USER -i hubos_network_interface -o lo -j REJECT --reject-with icmp-port-unreachable
4 -A DOCKER-USER -i hubos_network_interface ! -o hubos_network_interface -j REJECT --reject-with icmp-port-unreachable
5 -A DOCKER-USER -j RETURN
```

Listing 3.6: Custom iptables rules applied to the `DOCKER-USER` chain to restrict container network access

Since creating these firewall rules requires elevated privileges (i.e., sudo rights), we have chosen to delegate this task to the user during the installation or setup phase. This decision allows us to avoid running any part of the New HubOS codebase with administrative privileges, which could introduce significant security risks if the source code were to contain vulnerabilities. By isolating the privilege escalation to a controlled and manual step, we reduce the attack surface and increase the principle of least privilege within the system.

3.5.4 Permission Manager

The final component of the Resource Access Control is the Permission Manager. Its primary role is to maintain an up-to-date record of all access rights granted to modules, based on authorizations issued by OpenHAB. Whenever a TABAC rule is triggered and access permissions are granted to a specific module, OpenHAB communicates this information to New HubOS. The Permission Manager then updates the internal access records accordingly.

As a result, if a module attempts to access a resources after its permission has expired. The request will be denied by the Permission Manager. This enforcement mechanism ensures that modules can only interact with resources when explicitly authorized, thereby reinforcing the security and integrity of the system.

3.6 REST API Service

In the context of New HubOS, modules are expected to interact with two main categories of resources. The first type consists of web resources, which refer to external services accessed via standard web requests, often to retrieve or send data, and which were discussed in the previous section. The second type includes resources related to OpenHAB ecosystem, encompassing both the smart devices managed by OpenHAB and the OpenHAB system itself, accessed through its dedicated API.

Indeed, OpenHAB's API allows not only interaction with individual devices, but also with the core system itself. However, giving modules full access to this API would pose significant security risks. First, the API requires an API token to allow request, and exposing such tokens to modules would be an unsafe practice. Second, some endpoints in the OpenHAB API can alter sensitive system configurations. For example, by deleting connected devices, installing potentially malicious addons, or creating and removing automation rules. A malicious module could exploit these endpoints to compromise the entire HubOS ecosystem.

To mitigate this risk, we chose to implement an intermediate interface: a restricted REST API service that acts as a controlled gateway between the modules and the OpenHAB API. This service only exposes a subset of OpenHAB endpoints deemed necessary and compatible with the philosophy of HubOS. Its primary purpose is to allow modules to interact with specific devices through OpenHAB in a secure and controlled manner.

This configuration not only prevents modules from having direct access to sensitive API tokens (since the token is managed internally by the REST API Service itself) but also enables New HubOS to enforce strict access control. Using the Permission Manager described earlier, the service ensures that modules can only interact with devices they are allowed to access, in accordance with assigned TABAC rules.

3.7 New Communications Flows

In the previous chapter, we addressed the limitations of using the Stream Bus to transmit video and audio data over the MQTT protocol. As mentioned, this approach raises concerns regarding performance and the potential overload of the MQTT broker, especially considering that MQTT was not designed for high-bandwidth, continuous data streams.

This section introduces an alternative, more traditional solution for transmitting media streams. We will present two strategies, each positioned at different points in the HubOS scheme.

3.7.1 At the Source

The first approach aligns with the idea of delegating device interaction responsibilities to the module itself. By granting direct access, the module remains in charge of requesting and managing the stream according to its internal logic. In the case of camera streaming, OpenHAB's default API does not natively support media streaming, this limitation can be overcome by installing the IP Camera binding [63]. This binding enables the integration of most IP cameras into OpenHAB and exposes several HTTP endpoints for retrieving media streams.

For example, once a camera is integrated, modules can access the following resources via HTTP request:

- A snapshot image:
`http://openhabIP:8080/ipcamera/cameraUID/ipcamera.jpg`
- A MJPEG stream composed of continuous snapshots:
`http://openhabIP:8080/ipcamera/cameraUID/snapshots.mjpeg`
- An HLS (HTTP Live Streaming) stream:
`http://openhabIP:8080/ipcamera/cameraUID/ipcamera.m3u8`

This setup allows each module to independently request access to a video stream whenever needed, without requiring a dedicated streaming service in between.

However, this method has a major drawback: the video stream is exposed via HTTP without authentication, which means it is neither encrypted nor protected by tokens or credentials to access.

Nevertheless, since access to the stream is initiated through a standard web request, it is automatically intercepted by the forward Proxy described earlier. If the requesting module has the appropriate authorization (as defined by TABAC rules and enforced by the Permission Manager), the proxy will allow the request to proceed, ensuring that only authorized modules can access the stream.

Regarding audio streaming, such as from or to a voice assistant, we can reasonably assume that the device will be accessible via its local IP address and communicate using a well-defined protocol and API. As a result, this does not pose any particular technical difficulty from the perspective of access. The same logic of interception and authorization can be applied, ensuring secure and filtered access to audio streams through the Forward Proxy.

Figure 3.7 illustrates an example of a module requesting a JPG image through the IP Camera binding.

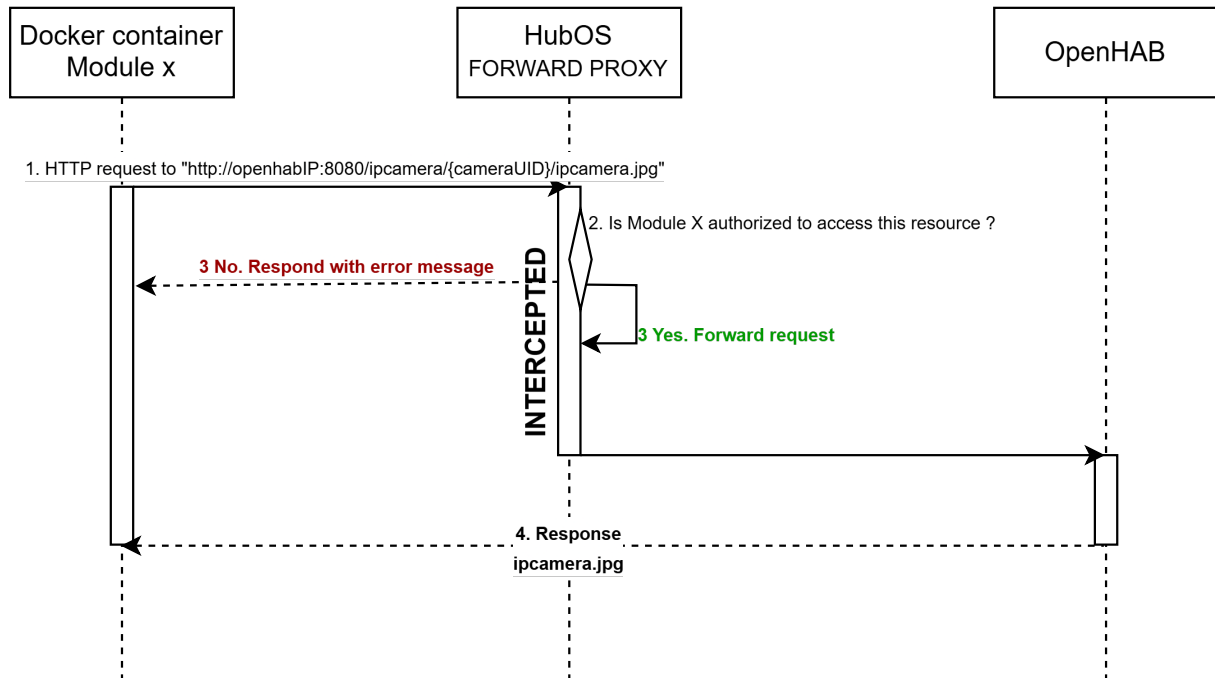


Figure 3.7: Example of a module requesting a JPG image via the IP Camera binding through the proxy

3.7.2 At the Stream Manager

Next, a second option is to place the data stream service at the HubOS level through the implementation of a dedicated controller the **Stream Manager**. Its role is to authorize stream access requests and relay the data stream to the intended recipient.

Architecturally, this design is very similar to the REST API Service presented earlier. The Stream Manager exposes a REST API interface that modules can call when they wish to access the stream of a specific connected device. Upon receiving the request, HubOS will first validate the module's permissions and, if authorized, it will initiate the retrieval of the stream from the device. This can be done either through OpenHAB's IP Camera binding or directly via the device's own API, depending on availability.

This centralized approach provides several benefits. Most importantly, it gives HubOS complete control over the stream flow, allowing it to implement rate limiting, access control, and unified handling of both audio and video streams. Additionally, the use of a common REST API simplifies integration for developers, while reducing the surface of exposure to external devices. The Stream Manager also enables future features such as data transformation. For instance, in scenarios where we might later wish to blur detected faces in a video stream or alter the pitch of the voice in an audio stream before delivering it to the requesting module.

Figure 3.8 shows an example of a module requesting a JPG image through the Stream Manager.

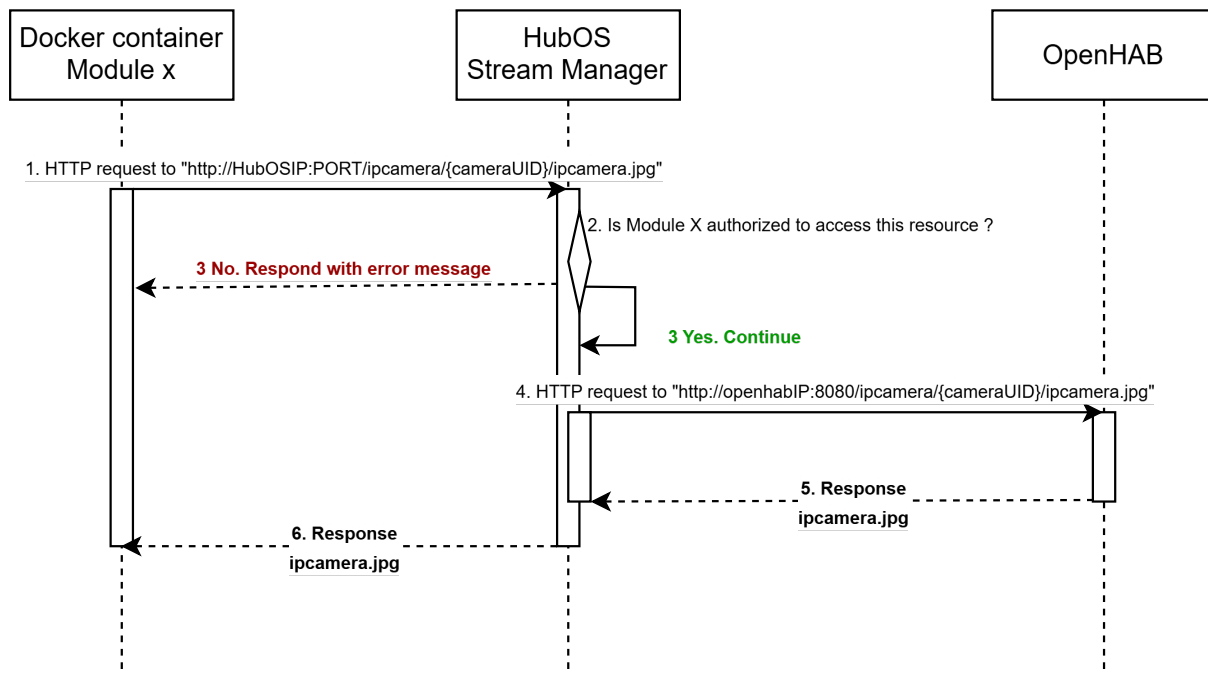


Figure 3.8: Example of a module requesting a JPG image through the Stream Manager

The implementation of such a service would not be particularly complex in itself. In fact, for the video streaming part, we are fortunate to benefit from the IP Camera binding, which abstracts interactions with cameras from various manufacturers such as ESP32 Camera, Hikvision, Amcrest, Dahua, and others. This significantly simplifies the integration of a wide range of video devices.

However, for audio streaming, the situation is more challenging. Currently, OpenHAB does not provide an addon that supports the integration of voice assistant listening devices in a generalized manner. Consequently, there is no unified interface for accessing audio streams from diverse hardware. As a result, the Stream Manager would need to communicate directly with each device using its native protocol, and ensure compatibility with a broad range of products.

For instance, one protocol that would require support is the Wyoming protocol [64], which is used by Home Assistant and Rhasspy [65], an open-source toolkit designed for building custom voice assistants. Supporting such protocols would be a prerequisite for enabling seamless two-way audio interactions, such as listening for commands and responding with synthesized speech.

Another important advantage of this approach lies in its ability to inspect and manage the content of the media streams, rather than only relying on the destination or source address, as is the case with Forward Proxy solution. This makes it possible to enforce deeper control, such as blocking inappropriate or malformed data, and verifying the nature of the transmitted content.

In conclusion, due to time constraints, we have chosen to proceed with the first solution, which delegates the responsibility of device interaction to the modules themselves. While this approach may not be the most sophisticated, it remains both functional and secure. Therefore, we consider the implementation of a centralized stream management service as a potential improvement for future version of HubOS.

3.8 Additional Security Measures

This section presents additional implementation aspects that, while not discussed in the previous chapters, are nonetheless essential to highlight as they reflect our commitment to build a secure and maintainable system with HubOS.

First, special attention was given to the creation of unit tests to ensure the correct functioning of the various internal components of HubOS. For example, all functions developed within the App Manager were tested to guarantee their reliability and robustness. In total, over 150 unit tests have been implemented for this purpose.

Second, although JavaScript is not inherently an object-oriented language, we adopted an object-inspired class structure. In addition, careful code documentation was maintained throughout the project, including class and function descriptions. This decision was made to enhance code readability, debugging, and facilitate porting the application to another language if needed.

Furthermore, the entire project adheres to recognized software engineering best practices, such as the use of the **Model-View-Controller** (MVC) architecture and the **Data Access Object** (DAO) pattern for database operations. These patterns contribute to improve maintainability, and scalability.

Moreover, since HubOS stores module-related information in a database and uses the database-generated identifier to uniquely reference each module, we opted for the use of UUIDs (Universally Unique Identifiers). A UUID is a 128-bit value designed to be globally unique, making it extremely difficult to guess or replicate. This adds an extra layer of protection, preventing malicious modules from spoofing other modules by predicting their IDs. A risk that would be more likely if a simple incremental ID system were used.

Then, a logging system was integrated into the application to facilitate debugging in case of unexpected behavior or system errors.

Finally, we made a deliberate effort to minimize external library dependencies implementing as many features as possible internally. For the libraries that were included, each was audited for known vulnerabilities to ensure they did not compromise the overall integrity of HubOS.

3.9 Overview of the Final Architecture

In this final section of the chapter, we review the architecture of New HubOS in its complete and integrated form. Figure 3.9 illustrates the overall system design, highlighting the interactions between the various components that make up the platform.

The diagram includes OpenHAB and its rule engine, the MQTT broker (which hosts the event bus), and the Docker environment, where containers are executed using the gVisor runtime for additional isolation. It also shows the Sandbox Manager, responsible for managing these containers, and the App Manager, which ensures that application file structures are valid and conform to expected standards.

The TABAC Manager is in charge of converting user-defined TABAC rules into TAR rules compatible with OpenHAB's rule engine. On the communication side, a REST API allows controlled interactions between modules and either HubOS itself or the OpenHAB API. For web resource access, the Forward Proxy intercepts outbound requests and applies access control policies.

Finally, the Permission Manager plays a central role by verifying whether a module is authorized to access a given resource, in accordance with the permissions dynamically assigned through the TABAC system.

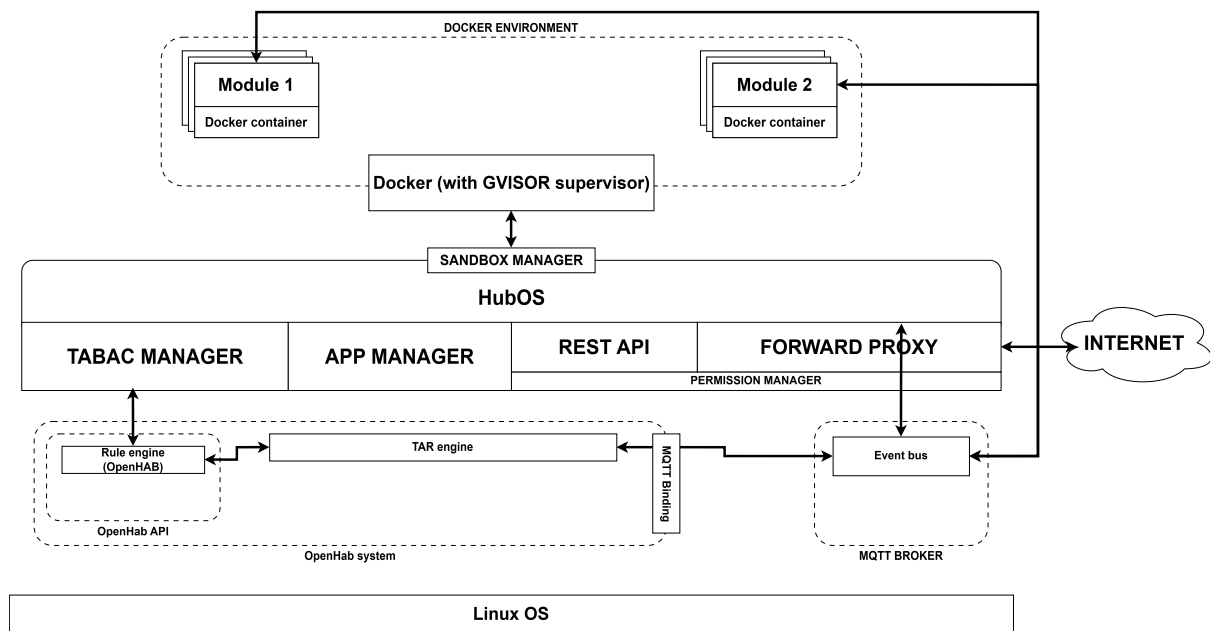


Figure 3.9: Final architecture of New HubOS

Chapter 4

Software Release Strategy and Functional Demonstration of HubOS

Since one of the long-term goals of this project is to make HubOS publicly available, it becomes essential to address aspects such as releases, packaging, and licensing. These elements play a key role in the distribution, maintenance, legal compliance, and potential adoption of the system by both developers and end-users. To support future users and contributors in understanding how to build and run applications on HubOS, this chapter will also include a practical use case demonstrating the system in action.

4.1 Packaging

Just as discussed through this report, HubOS has been designed to integrate seamlessly with either a new or pre-existing OpenHAB setup. This allows the end-user to continue using their already-configured smart home environment without the need to transfer any backups, making the project more lightweight. However, if the user does not already have OpenHAB installed, they will need to set it up beforehand.

Since the project is built using Node.js, it remains accessible even to amateur users (the only prerequisites are installing Node.js and the project dependencies ¹. That said, a future enhancement could involve packaging HubOS as a standalone executable for easier deployment. However, this approach would increase the file size significantly, as the Node.js runtime would have to be bundled into the final binary.

The source code of HubOS, along with all files required for its deployment and production setup, is publicly available in the GitHub repository [66].

In addition, the user must perform a few installation and configuration steps to ensure the system functions correctly. These include:

- Installing OpenHAB (if not already present),

¹via npm install

- Installing Mosquitto and configuring the Dynamic Security Plugin,
- Setting up Docker and gVisor,
- Installing and configuring a PostgreSQL database.

4.2 Distribution

HubOS has been designed to align not only with the technical ecosystem of OpenHAB, but also with the open-source values it promotes. As stated by its founder, Kai Kreuzer: “OpenHAB is done by and for the community: It is positioned to be free and totally open.” [67] Given that HubOS is intrinsically connected to OpenHAB and that OpenHAB itself is open-source, we aim to follow the same philosophy by releasing HubOS as an open-source and community-driven project. Therefore, it is essential to consider, within this section, key aspects such as licensing, contribution frameworks, and security policy.

4.2.1 Open-Source Licensing

One of the prerequisites before distributing HubOS to the public is choosing an appropriate license. A license defines the rights and limitations associated with the product. These restrictions may apply to various aspects, such as its use, redistribution, or modification.

When releasing a project as open-source, it is crucial to carefully consider the licensing model, as it will shape how the community can contribute to, use, and distribute the software. The chosen license will ultimately influence the openness, legal framework, and long-term sustainability of the project [68, 69].

OpenHAB is Open-source and released under the Eclipse Public License v2.0 (EPL 2.0) [70]. This license allows developers to copy, modify, and distribute the covered software, provided they comply with the obligations outlined in the license. Among the key requirements are the need to provide the source code of the distributed software, and to clearly state that it is licensed under the EPL if any modifications have been made.

Given HubOS’s close integration with OpenHAB and its adoption of similar values, it might seem logical to assume that it should also inherit the same license. However, since HubOS does not modify or redistribute the OpenHAB source code, it is not legally required to adopt the same license.

That said, HubOS itself includes several dependencies, each governed by its own license. It is therefore important to examine these components. In total, HubOS imports 17 Node.js dependencies, covered by three different open-source licenses:

- MIT License [71]
- Apache 2.0 License [72]
- BSD-2-Clause License (also known as the FreeBSD License) [73]

(A complete list of all dependencies, their versions, and licenses can be found in Appendix B.1.

These are all considered permissive licenses, as they comply with the Open Source Definition. In general terms, they allow software to be freely used, modified, and redistributed, often with minimal obligations such as attribution and inclusion of the license text. Importantly, these licenses do not impose restrictions on how the software can be used. This legal flexibility ensures that HubOS remains unconstrained by its dependencies.

Furthermore, since HubOS not yet provide a compiled or executable version of the system, users are required to install the dependencies themselves using the `npm install` command. As a result, HubOS does not technically fall under the definition of redistribution of these dependencies, which simplifies our obligations regarding license compliance.

To further respect and acknowledge the license of these dependencies, a NOTICE file will be included with the project. This file will explicitly list all third-party packages used in HubOS, along with their respective licenses and copyright statements.

We found that HubOS is not legally bound to adopt specific license. This gives us full freedom to choose the most suitable open-source license for our project.

Given the vast array of open-source licenses available (each with its own legal implications and scope) this choice can be complex. A straightforward approach would be to adopt the popular MIT License, which is widely used in Node.js ecosystem. The MIT License is highly permissive, allowing anyone to freely use, copy, modify, and distribute the software, while also disclaiming liability and warranty.

However, upon deeper consideration, aligning our choice with the license used by OpenHAB would be more meaningful. Since HubOS heavily relies on OpenHAB in both functionality and philosophy, using the same license fosters compatibility and shows respect for the ecosystem in which HubOS operates.

Moreover, EPL-2.0 strikes a strong balance between openness, contributor traceability, and legal safeguards. It requires that source code be made available upon distribution and that any modifications be clearly identified. This promotes transparency, encourages community involvement, and ensures a more structured collaborative development process.

Therefore, we have decided to license HubOS under the Eclipse Public License v2.0. This choice reinforces our commitment to open-source values and ensures that HubOS remains both accessible and responsibly maintained.

4.2.2 Open-Source philosophy

In line with the open-source philosophy of the project, it is important to consider how the community can actively contribute to HubOS, not only by submitting new features or improvements, but also by reporting issues or potential vulnerabilities. Establishing clear mechanisms for contribution and feedback fosters transparency and enables collective problem-solving, ultimately strengthening the robustness and evolution of the system.

Community Contributions

Just like its counterpart OpenHAB, HubOS aims to encourage community contributions, especially through pull requests, which we will do our best to process them promptly while providing constructive feedback.

To ensure that all contributions follow a consistent and high-quality structure, the following contribution guidelines will be established:

Conventions

1. All contributions must begin by forking the repo and applying changes on a dedicated feature branch.
2. The name of the feature branch should follow the format xxx-something, where xxx refers to the issue number.
3. All changes must include relevant unit tests, and the entire test suite must pass before submitting a pull request.
4. If applicable , contributors must update the documentation accordingly, ensuring clarity, conciseness, and correctness.
5. Code must be clean, well-commented, and properly formatted, in order to promote ease of reading, writing, and maintenance.
6. Pull request descriptions must be clear, structured, and include references to the issues they address.
7. Commits that fix or close an issue must include a reference such as Closes #xxx or Fixes #xxx allowing the issues to be automatically closed upon merge.

These guidelines will help us maintain the integrity and sustainability of the project, ensuring that the codebase remains up-to-date, clean, and easy to maintain.

Additionally, to provide an extra layer of legal protection, we will require contributors who submit code changes to sign a Developer's Certificate of Origin [74]. This certificate affirms that the contributor is indeed the author of the submitted changes and has the legal right to contribute them under the project's open-source license. This process ensures transparency, and legal clarity regarding the ownership and origin of contributions.

Security Policy

To ensure proper issue tracking within the project, the community will be encouraged to submit issues directly through the project's GitHub repository. In more critical cases, contributors will also have the possibility to report vulnerabilities. This openness allows the community to support one another and proposes fixes or improvements in response to identified problems.

In addition, the project maintainers will remain actively involved to monitor submissions, provide guidance, and ensure that the project continues to evolve in a secure manner.

Documentation

In order to facilitate collaboration and improve the understanding of the project for community members wishing to contribute either to the development of HubOS itself or the creation of new applications, particular emphasis was placed on documentation.

Significant effort was invested in providing detailed documentation describing the internal functioning of HubOS, including its architecture, core components, and system logic. In parallel, the source code is also documented to offer a more technical and implementation-oriented perspective.

This approach aims to lower the entry barrier for new contributors, support collaborative development, and ensure that HubOS remains understandable, extensible, and maintainable for the community over time.

4.3 Example of an Application

In this chapter, we expressed our intention to align with the open-source philosophy promoted by OpenHAB. To do so, we examined how to structure community contributions to the project, and we discussed the choice of the most appropriate license for HubOS.

As a final point in this chapter, we will present an example application integrated into the project, which demonstrates:

1. The capabilities of HubOS
2. Its interaction with OpenHAB, and
3. How the community is expected to develop applications and their associated modules.

This example builds on the concept of the Front Door Smart Security Camera previously introduced in section 1.2.2, in which a camera placed at the entrance of a house performs facial recognition and notifies the homeowner if an unrecognized face is detected. We will refer to this application as FaceCamera

To highlight HubOS's ability to handle multiple modules developed in different programming languages, this application is composed of two distinct modules:

- The **first module**, written in Python, receives the video stream from the camera and performs facial recognition to identify whether the detected face matches that of the homeowner. Python was chosen here due to its strong support for computer vision and facial recognition libraries.
- The **second module**, developed in Node.js, is responsible for alerting the homeowner when an unknown face is detected. To demonstrate the power of TABAC rules, the usefulness of permission assignment, and the possibility of interacting with external services, this second module is granted two permissions when triggered:
 1. Access to the video stream of the camera, and

2. Access to the Telegram API - This API allows the module to send message and snapshots via a Telegram bot to the homeowner through a dedicated chat.

Although we will not detail the code implementation of these modules in this report, we will present the associated manifest.json, config.json and TABAC rule files, as they are essential to understanding how applications are structured and deployed within HubOS.

4.3.1 Manifest.json

The manifest.json file shown in Listing 4.1 identifies the names of the two modules in the application: face_recognition and notifier, corresponding respectively to the module responsible for facial recognition and the one responsible for notifying the homeowners. The folders containing the source code for these modules must use the same names, as illustrated in Figure 4.1

```
1 {
2   "name": "FaceCamera",
3   "description": "This application uses a camera positioned at the front door to
4     detect faces. If a recognized (pre-registered) face is detected, no action
5     is taken. However, if an unknown face is detected, the owners are
6     immediately notified with an alert and a snapshot of the visitor.",
7   "type": "faceCamera",
8   "modules": [
9     {
10      "name": "face_recognition",
11      "type": "handler",
12      "description": "This module continuously intercepts the video stream
13        from the front door camera and performs real-time face recognition
14        to identify whether the detected faces match those previously
15        registered in the system."
16    },
17    {
18      "name": "notifier",
19      "type": "notifier",
20      "description": "This module sends notifications to the owners when an
21        unknown face is detected by the face recognition module."
22    }
23  ]
24 }
```

Listing 4.1: Manifest.json file of the FaceCamera app

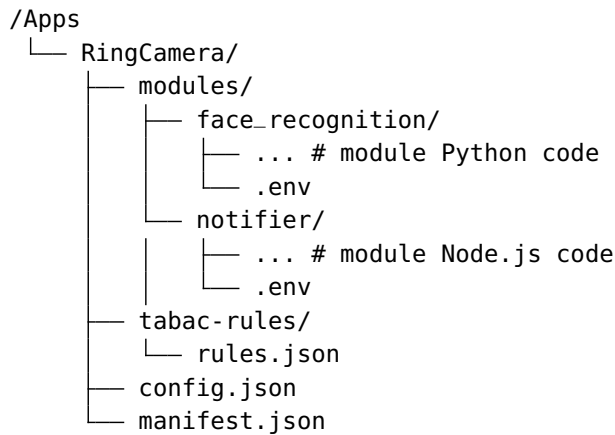


Figure 4.1: File Structure of the RingCamera App

4.3.2 Config.json

The `config.json` file shown in Listing 4.2 defines four key elements required for the application's execution.

First, the `devices` field is left empty because this application does not need access to any OpenHAB items during its runtime.

In contrast, the `servers` field requires the configuration of two entries: The host of the Telegram API and the host of the user's OpenHAB server. The Telegram API host is already prefilled, as it is likely to be the same for all the users who wish to use this application. However, the OpenHAB server host must be defined by the user, as it refers to the IP address of their own OpenHAB instance (a value that will vary depending on their local setup).

Finally, this application also requires additional third-party configuration values that are necessary for the application's logic, but not for HubOS itself. This include:

- The Telegram API token;
- The Chat ID to which notifications and snapshots should be sent;
- And the `IP_Camera` URLs, in order to access the desired video stream.

```

1 {
2   "configuration":{
3     "devices":{
4     },
5     "servers":{
6       "telegramServer":{
7         "host": "api.telegram.org:443",
8         "description": "telegram server to send messages"
9       },
10      "OpenHABServer":{
11        "host": "",
12        "description": "Openhab server <IP>:8080"
13      }
14    },
15    "others":{
16      "TelegramBot":{
17        "token": "",
18        "chatId": "",
19        "description": "Telegram API token and chat ID to send notifications
20                        messages"
21      },
22      "DoorbellCamera":{
23        "mjpegUrl": "",
24        "mjpegUrlDescription": "IPCamera url to mjpeg stream (http://<
25                               openhab_ip>:<openhab_port>/ipcamera/<camera_uid>/ipcamera.mjpeg)
26        ",
27        "snapshotUrl": "",
28        "snapshotDescription": "IPCamera url to snapshot stream (http://<
29                               openhab_ip>:<openhab_port>/ipcamera/<camera_uid>/ipcamera.jpg)"
30      }
31    }
32  }
33 }

```

Listing 4.2: Config.json file of the FaceCamera app

4.3.3 Rules.json

The final file required for the application's execution contains the TABAC rules, as shown in Listing 4.3. This file defines two important rules:

- The first rule is triggered every hour between 9:00 AM and 6:00 PM. It updates the access permissions for the face_recognition module, allowing it to access the camera's video stream during that time window.
- The second rule is triggered when the module's topic is updated with the value "notify_owners". This event grants the notifier module two temporary permissions: Access to the Telegram API service and access to the camera stream, both valid for 5 minutes. Additionally, the event is forwarded to the notifier module to inform it of the event.

```

1  [
2    {
3      "name": "face_recognition_work_time_access_to_camera_stream",
4      "description": "Specify activation time access to camera stream for face
5        recognition module",
6      "when": {
7        "event": "system.time",
8        "context": "GenericCronTrigger",
9        "value": "0 0 9-18 * * ? *"
10     },
11     "condition": [],
12     "then": [
13       {
14         "access": "NetworkClient",
15         "type": "service",
16         "context": {
17           "period": "3600", // 1 hour
18           "host": ["OpenHABServer"],
19           "concern": "face_recognition"
20         }
21       }
22     ],
23   },
24   {
25     "name": "ring_button_call_ring_handler_module",
26     "description": "when ring button triggered call ring handler with auths",
27     "when": {
28       "event": "mqtt.face_recognition",
29       "context": "updated",
30       "value": "notify_owners"
31     },
32     "condition": [],
33     "then": [
34       {
35         "access": "NetworkClient",
36         "type": "service",
37         "context": {
38           "period": "300", // 5 minutes
39           "host": ["telegramServer", "OpenHABServer"],
40           "concern": "notifier"
41         }
42       },
43       {
44         "access": "pass_through",
45         "type": "flow",
46         "context": {
47           "concern": "notifier",
48           "value": "notify_owners"
49         }
50       }
51     ]
52   }

```

Listing 4.3: rules.json file of the FaceCamera app

Chapter 5

Conclusion

The outcome of this project is a nearly complete implementation of the new HubOS system, designed to integrate seamlessly with either a fresh or existing OpenHAB home automation setup. Since communication between the two systems is handled through the OpenHAB API, this architecture offers flexible deployment: HubOS can run on the same host machine as OpenHAB or on a dedicated server.

This new version of HubOS distinguishes itself from its first version not only by the completeness of its implementation but also by its strong adherence to the original philosophy: extending OpenHAB with the ability to run private-by-design applications. However, our system sets itself apart through the care invested in building a nearly complete, well-thought-out and secure solution that minimizes potential risks.

New HubOS features container execution isolated through Docker’s gVisor runtime, a permission management system governing resource access via both the Forward Proxy and REST API (flow bus), and an event handling mechanism through a secured MQTT-based Event Bus with controlled access.

In this conclusion, we will review the key insights gained throughout the project, discuss the main challenges encountered, and highlight areas of HubOS that warrant further exploration. Finally, the role of AI-based tools in the development process will be addressed, before closing with acknowledgments.

5.1 Reflection

This work has undoubtedly been a major source of learning for me. From researching alternatives to VM2, to working with Docker containers and implementing a forward proxy, it has allowed me to deepen my knowledge in these areas and many others. Already interested in home automation prior to this project, this experience confirmed my enthusiasm for the field. The entire process (understanding my predecessor’s work, designing a new architecture, revisiting certain decisions, and ultimately implementing them in code) has provided valuable insights, especially regarding how to carry a project from initial design to final implementation.

5.2 Challenges Faced

Several aspects of the project development posed significant challenges in terms of both design and implementation. First, since the previous version of the project was a proof of concept, there was little reusable code. Apart from the core logic, which was retained and reworked, the entire system had to be rebuilt from scratch. Another major challenge was the implementation of the forward proxy. This component demanded extensive research, especially since the project was developed in Node.js (JavaScript runtime environment designed for building servers and web applications). Most available proxy libraries in the Node.js ecosystem are designed for reverse proxying, making them incompatible with our need for a true forward proxy. A final constraint was the use of JavaScript as the main language. Being dynamically typed, JavaScript can lead to subtle programming errors that are not caught at compile time. This requires additional effort in reviewing and testing the code to ensure correctness and stability. JavaScript had initially been chosen because the previous version of the project was also written in this language. Since the original goal of this work was to continue and extend that earlier prototype, it made sense to follow the same direction. However, with the major architectural shift from VM2-based sandboxing to containerization using Docker, one of the main reasons for keeping JavaScript became obsolete. Indeed, VM2 relied heavily on Node.js's internal vm module, which was specific to the JavaScript ecosystem. Once this component was removed, the choice of JavaScript lost some of its original justification. At that point, we briefly considered switching to a different language better suited to systems programming, such as Python, Java, or Go. However, the development had already progressed significantly, and rewriting the codebase would have added considerable overhead. As a result, we decided to continue with JavaScript while applying strict development practices.

5.3 Next Steps for HubOS

This section reviews several components of New HubOS that would require further development or reevaluation in the event of a future version of the project.

5.3.1 Rethinking Stream Access

One of the first areas worth revisiting is the improvement of communication flows, particularly the Flow Bus mechanism discussed in the previous chapter. In which, we explored two approaches for enabling access to streaming resources such as video from cameras and audio from smart speakers. We concluded that the most robust and secure solution would involve implementing a centralized Stream Manager, essentially a REST API service dedicated to stream handling.

While this approach offers clear advantages such as the ability to inspect and filter request content (something not feasible with the existing forward proxy setup). It was not adopted in the current version due to complexity of ensuring compatibility with the wide range of communication protocols used by smart devices.

Designing such a service would require extensive work to support heterogeneous standards across various brands and device types. However, given its potential benefits in terms of access control, security, and content monitoring, the implementation of a centralized Stream Manager should be considered a priority for future iterations of HubOS.

5.3.2 HubOS Web-Based Graphical User Interface

A second area worth developing in a future version of HubOS is the addition of a graphical user interface (GUI) in the form of a web dashboard. This interface would provide users with a clear and user-friendly overview of the system's status, including running applications, active modules, and their associated rules.

Currently, New HubOS interacts with the user exclusively through the command-line interface (CLI), which, although powerful, may not be intuitive or accessible for all users. A web-based GUI would significantly improve usability, system monitoring, and ease of configure, especially in environments with multiple applications and dynamic permissions rules. Moreover, such an interface would also strengthen the user-agreement process for newly installed applications. In the current CLI-based approach, users are required to manually open and read configuration files to understand the implications of an application's permissions. This design does not fully address the issue highlighted during the initial presentation of HubOS: many users tend not to read or understand permission prompts. With the introduction of a graphical interface, it becomes possible to clearly display configuration files content, visually highlight the most sensitive or critical elements, and guide users through the validation process in a more accessible and informative way.

Due to time constraints, it was not possible to finalize the implementation of this graphical interface before the publication of this report. However, to showcase the preliminary design work that has been undertaken, we present a series of mockups illustrating what the future web-based GUI could look like, along with the benefits it would bring to the user experience. These mockups are provided in Appendix C. It is important to note that this design is not final and may differ from the eventual implementation, as further development and user feedback will likely influence the final version.

Initial Access and Credential Configuration

First and foremost, since this is a web-based GUI, access to the control interface must be restricted to the authorized administrator. For this reason, during the first connection to the interface, the user will be prompted to set a username and password (see Figure C.2). These credentials will be securely stored in the HubOS database. On each visit to the management interface, the user will be required to enter these login credentials (see Figure C.1). In the event that the user forgets their credentials, it will be necessary to reset the HubOS instance via the command-line interface (CLI).

Application Status Dashboard

The homepage of the interface (see Figure C.3) is designed to provide the user with a clear overview of the status of each application running within the HubOS environment at a glance. This page is structured as a three-column grid of tiles, each displaying the name of the application along with its current status.

Each application can have one of four distinct statuses, visually represented by color codes for quick identification:

- **Not Installed** (grey): The user has not yet configured or launched the initial execution of the application, and thus the Docker containers have not been created.
- **Running** (green): The Docker containers for all the modules have been successfully created and are currently running.
- **Paused** (blue): The containers have been created but were manually paused by the administrator.
- **Crashed** (red): One or more modules have failed, or the Docker containers could not be created.

The use of color coding enables the administrator to quickly assess the health and operational state of each application. When the administrator clicks on a tile, they are redirected to the application detail page, which provides more in-depth information.

Reviewing and Validating Application Details

The application details are organized into three main sections: the manifest file, the configuration file and the TABAC rules (see Figures C.4, C.6, and C.7 respectively).

The Manifest section simply displays the content of the application's manifest file. It does not require any action from the administrator, aside from reviewing the information and validating it. If the file is incomplete, a warning will be displayed to alert the administrator that this may indicate an abnormal or potentially unsafe application, and that further verification is advised.

The Configuration section is where the administrator is expected to interact more actively. This section allows the administrator to fill in any required fields needed to configure the application properly. Fields that need input are marked with a special symbol, allowing for quick identification of incomplete configurations.

The Rules section lists all TABAC rules associated with the application, along with additional metadata such as the last time each rule was triggered. If the administrator clicks on the name of a rule, they are redirected to a detailed view of that rule (see Figures C.8 and C.9).

At the bottom of each section is a checkbox labeled "Validate the [section name]". In order to authorize the first execution of the application, the administrator is required to visit each of the three sections, scroll to the bottom, and check this box to confirm validation. This step ensures that the administrator has reviewed each part of the application before activation.

Application Control Buttons

Finally, at the top of the application details page, three specific buttons are available to the administrator:

- **Initialize Application:** Launches the application for the first time and creates the necessary Docker containers to run its modules.
- **Resume Application:** Resumes the execution of an application that was previously paused.
- **Pause Application:** Suspends the execution of the Docker containers running the application modules.

During the initialization process, a confirmation modal appears (see Figure C.10), informing the administrator that all sections (manifest, configuration, and rules) must be validated, and that a final explicit authorization is required. Without this step, the application will not be executed.

Viewing Module and Application Logs

Additionally, it would be useful to provide the administrator with a dedicated page to view the logs of each modules individually, or grouped by application.

5.3.3 Remote Docker Interaction

And finally, a third area worth exploring is the ability to decouple the Docker environment from the main HubOS host. In the current architecture, as described in the previous chapter, HubOS interacts with the Docker Engine through the `dockerode` library. However, `dockerode` is essentially a client interface designed to simplify communication with the Docker Engine API.

As such, it is not limited to local deployments; it also supports remote connections. By specifying the appropriate IP address and port, HubOS can seamlessly interact with a Docker environment hosted on a different machine. This enhancement would offer greater deployment flexibility, allowing resource-intensive modules to run on separate, dedicated servers while keeping the core HubOS instance lightweight and focused.

5.4 Final Remarks

It is important to clarify that the HubOS project is not yet fully complete. While significant portion of the development has been successfully finalized and the system could technically be deployed in a production environment starting tomorrow, there remains room for improvement. It will also be essential to monitor the evolution of HubOS over time to ensure its continued functionality and relevance.

Moreover, in this report, we deliberately chose not to detail every internal mechanism and implementation aspect of the new HubOS system. As a result, readers seeking a more in-depth, code-oriented, and up-to-date documentation are

strongly encouraged to consult the repository documentation [66] as well as the source code itself, which together provide a more exhaustive and technical perspective on the system’s behavior and design choices.

Although we cannot predict with certainty whether HubOS will resonate with the smart home communities, we believe that if interest does arise, it will be crucial to meet that demand. This includes not only continuing to improve the platform but also fostering a strong communities. For example, through the creation of a dedicated forum to facilitate communication among users and contributors.

In addition, a dedicated website could be developed to centralize the list of applications created and shared by the community. This platform could include a voting system to highlight not only the most popular applications but also those considered safe and reliable by the users.

Additionally, it is worth mentioning that a new team has recently begun working on potential improvements to the project, with an initial focus on the implementation of a more complete REST API service, as well as on the integration of the stream access model presented earlier. Their efforts have already helped validate the operational stability of HubOS across various devices and network configuration.

5.5 Usage of Artificial Intelligence

Artificial intelligence tools were used at various stages throughout the development of this thesis project. These tools were primarily employed to streamline the development process and save time. However, all design decisions and core logic originated from me. AI was mainly used as a coding assistant, helping to speed up certain tasks such as searching for function names or recalling syntax more efficiently.

Additionally, AI played a significant role in the writing phase. It greatly assisted in improving the clarity and style of this report, particularly in English, by helping formulate more formal and fluent sentences. This allowed me to save time and express my idea more clearly and effectively.

5.6 Acknowledgments

First and foremost, I would like to express my gratitude to Professor Riviere, whose original idea inspired the creation of this thesis. His guidance, support, and expertise throughout the project have been invaluable and greatly deepened my understanding of the subject.

I also wish to thank François De Keersmaecker and Lionel Metongnon for being part of my thesis jury and for dedicating their time to review and assess my work.

Special thanks also go to Sacha Defrère and Henri Pihet for help and for taking over the project future continuation.

Finally, I extend my thanks to my friends and family who supported me throughout this work: Kayrah H., Dode K., Elvis K., Arnaud M., Salma, M., Ayoub T., and Yi Z.

Appendix A

TABAC Rules

A.1 Trigger Conditions

At this moment there are 11 different keywords to verify an event:

1. CHANGED

- (a) The rule will be triggered if the value of the event changes from one specific state to another.

```
1 "when":{  
2   "event":"mqtt.fallDetection"  
3   "context":"changed",  
4   "value": [False, True]  
5 }
```

Listing A.1: simple example of a TABAC 'changed' trigger

- (b) A variant of this rule exists when the user want to check if the event changes from one ANY state to another ANY another state.

```
1 "when":{  
2   "event":"mqtt.fallDetection"  
3   "context":"changed",  
4   "value": ["ANY", "ANY"]  
5 }
```

Listing A.2: simple example of a TABAC 'changed any' trigger

2. UPDATED

- (a) The rule will be triggered if the value of the event is updated (in other words, it receives the same state).

```
1 "when":{  
2   "event":"mqtt.fallDetection"  
3   "context":"updated",  
4   "value": "ANY"  
5 }
```

Listing A.3: simple example of a TABAC 'updated' trigger

3. BETWEEN

- (a) The rule will be activated if the value of the event is between two thresholds
- (b) $a \leq event.value \leq b$.

```
1 "when":{  
2   "event":"system.time"  
3   "context":"between",  
4   "value": ["00:01", "10:00"]  
5 }
```

Listing A.4: simple example of a TABAC 'between' trigger

4. CONTAINS

- (a) The rule will be activated if the defined element is a subtext of the event value.
- (b) $subText \subset event.value$.

```
1 "when":{  
2   "event":"mqtt.vocalCommand"  
3   "context":"contains",  
4   "value": "Turn on the light"  
5 }
```

Listing A.5: simple example of a TABAC 'contains' trigger

- (c) Only text values are accepted.

5. CONTAINS ANY

- (a) The rule will be triggered if one of the defined elements is a subtext of the event value.
- (b) $event.value \supset [for\ subText\ in\ [value1, value2, \dots, valueN]]$.

```
1 "when":{  
2   "event":"mqtt.vocalCommand"  
3   "context":"contains any",  
4   "value": ["Turn on the light", "Open the garage", ... ]  
5 }
```

Listing A.6: simple example of a TABAC 'contains any' trigger

- (c) Only text values are accepted.

6. EQUALS

- (a) The rule will be triggered if the value of the event equals a specific value.
- (b) $event.value = value$

```

1 "when":{
2   "event":"mqtt.fallDetection"
3   "context":"equals",
4   "value": True
5 }

```

Listing A.7: simple example of a TABAC 'equals' trigger

7. EQUALS ANY

- (a) The rule will be triggered if the value of the event equals any of the given values.
- (b) $event.value \subset [value1, value2, \dots, valueN]$

```

1 "when":{
2   "event":"mqtt.FacialDetection"
3   "context":"equals any",
4   "value": ['Person1', 'Person2', 'Person3', ...]
5 }

```

Listing A.8: simple example of a TABAC 'equals' trigger

8. HIGHER THAN

- (a) The rule will be triggered if the value of the event is higher than a given value.
- (b) $event.value > value$

```

1 "when":{
2   "event":"mqtt.TotalPeople"
3   "context":"higher",
4   "value": 5
5 }

```

Listing A.9: simple example of a TABAC 'higher than' trigger

9. HIGHER OR EQUALS THAN

- (a) The rule will be triggered if the value of the event is higher or equals than a given value.
- (b) $event.value \geq value$

```

1 "when":{
2   "event":"mqtt.TotalPeople"
3   "context":"higher or equals",
4   "value": 10
5 }

```

Listing A.10: simple example of a TABAC 'higher or equals than' trigger

10. LOWER THAN

- (a) The rule will be triggered if the value of the event is lower than a given value.

- (b) $event.value < value$

```
1 "when":{
2   "event":"mqtt.TotalPeople"
3   "context":"lower",
4   "value": 17
5 }
```

Listing A.11: simple example of a TABAC 'lower than' trigger

11. LOWER OR EQUALS THAN

- (a) The rule will be triggered if the value of the event is lower or equals than a given value.

- (b) $event.value \leq value$

```
1 "when":{
2   "event":"mqtt.TotalPeople"
3   "context":"lower or equals",
4   "value": 123
5 }
```

Listing A.12: simple example of a TABAC 'lower or equals than' trigger

12. prefix NOT

- (a) By adding the **NOT** prefix to the context, it is possible to perform the inverse check. For example, **NOT EQUALS** will be triggered if and only if the value of the event is not equal to the defined value.

- (b) $event.value \neq value$

```
1 "when":{
2   "event":"mqtt.fallDetection"
3   "context":"not equals",
4   "value": False
5 }
```

Listing A.13: simple example of a TABAC 'NOT EQUALS' trigger

A.2 Action Types

1. Network client access

- (a) Allows access to the network client, and consequently to the external network, for instance, to communicate with a remote server.

```
1 "then": [
2   {
3     "access": "NetworkClient",
4     "type": "service",
5     "context": {
```

```

6         "period": "3600",
7         "host": ["myServer", "myServer2"],
8         "concern": "module2"
9     }
10 },
11 {
12     "access": "NetworkClient",
13     "type": "service",
14     "context": {
15         "period": "3600",
16         "host": ["all"],
17         "concern": "module3"
18     }
19 }
20 ]

```

Listing A.14: simple example of a TABAC 'network client' access

2. Device access

- (a) Grants direct access to an OpenHAB device, for instance, to adjust a thermostat's temperature.

```

1 "access": "myThermostat", // represent the device name as defined in conf.
  env
2 "type": "device",
3 "context": {
4     "period": "120", // Time-limited access in seconds
5     "concern": "module1"
6 }

```

Listing A.15: simple example of a TABAC 'direct device' access

3. Pass through

- (a) Forward an alert to one module.

```

1 "access": "pass_through",
2 "type": "flow",
3 "context": {
4     "concern": "module2",
5     "value": "event1"
6 }

```

Listing A.16: simple example of a TABAC 'pass through' action

Appendix B

Software Packaging, Licensing, and Distribution Model of New HubOS

B.1 List of imported dependencies with their licenses, versions, and utilities

License	Dependency Name	Version	Utility
MIT	Axios [75]	1.9.0	HTTP client for REST requests
	base-64 [76]	1.0.0	Base64 encoding/decoding utilities
	bcrypt [77]	5.1.1	Password hashing library
	commander [78]	14.0.0	CLI argument parser and command manager
	cors [79]	2.8.5	Express middleware for handling CORS
	crypto-js (crypto) [80]	1.0.1	Cryptographic algorithms in JavaScript
	express [81]	5.1.0	Web framework for building APIs and servers
	http-proxy [82]	1.18.1	Programmable HTTP proxy library
	jsonwebtoken [83]	9.0.2	JWT creation and verification
	mqtt (mqtt.js) [84]	4.3.8	MQTT protocol client for pub/sub messaging
	pg (node-postgres) [85]	8.14.1	PostgreSQL client for Node.js
	straightforward [62]	4.2.2	Forward-proxy library for HTTP traffic
	tar-fs [86]	3.0.8	Tar archive creation/extraction using FS streams
	uuid [87]	11.1.0	Generation of RFC-compliant UUIDs
	vitest [88]	3.1.1	Fast unit testing framework
Apache-2.0	dockerode [61]	4.0.6	Docker Remote API client for Node.js
BSD-2-Clause	dotenv [89]	16.0.3	Loads environment variables from a .env file

Table B.1: List of imported dependencies with their licenses, versions, and utilities

Appendix C

HubOS Web-Based Graphical User Interface

HubOS

Sign in

Username

Password

Sign in

[Can't access your account ?](#)

Figure C.1: Mockup: Viewing the admin Sign in page

HubOS

Create your admin account

Sign Up

Figure C.2: Mockup: Viewing the admin Sign up page

HubOS



FaceCamera
Not Installed

ExampleApp2
Paused

ExampleApp3
Running

ExampleApp4
Crashed

+

Figure C.3: Mockup: Viewing the Home screen listing all the apps

HubOS

FaceCamera

Not Installed

⬇️ ▶️ ⏸️

Manifest

Config

Rules

⚠️ Warning: Manifest not validated.

Description

This application uses a camera positioned at the front door to detect faces. If a recognized (pre-registered) face is detected, no action is taken. However, if an unknown face is detected, the owners are immediately notified with an alert and a snapshot of the visitor.

Type: faceCamera

Modules

Module 1

Name: face_recognition

Type: handler

Description:

This module continuously intercepts the video stream from the front door camera and performs real-time face recognition to identify whether the detected faces match those previously registered in the system.

Module 2

Name: notifier

Type: notifier

Description:

This module sends notifications to the owners when an unknown face is detected by the face recognition module.

☐ Validate the manifest

Figure C.4: Mockup: Viewing application manifest details

HubOS

FaceCamera

Not Installed

⬇️ ▶️ ⏸️

Manifest ?

Config

Rules

? Warning: Manifest not validated.

Description

This application uses a camera positioned at the front door to detect faces. If a recognized (pre-registered) face is detected, no action is taken. However, if an unknown face is detected, the owners are immediately notified with an alert and a snapshot of the visitor.

Type: faceCamera

Modules

Module 1

Name: face_recognition

Type: handler

Description:

This module continuously intercepts the video stream from the front door camera and performs real-time face recognition to identify whether the detected faces match those previously registered in the system.

Module 2

Name: notifier

Type: notifier

Description:

? WARNING: This field is empty and does not follow HubOS best practices. Make sure this app is safe

☐ Validate the manifest

Figure C.5: Mockup: Viewing application manifest details not complete

HubOS

FaceCamera

↓ ▶ ||

Not Installed

Manifest

Config

Rules

Configuration

⚠ Warning: Some fields are empty and need to be properly configured.

Devices

Servers

TelegramServer

Host:

Description: telegram server to send messages

OpenHABServer

Host:

Description: Openhab server <IP>:8080

Others

TelegramBot

Token:

ChatID:

Description: Telegram API token and chat ID to send notifications messages

TelegramBot

mjpegUrl:

mjpegUrlDescription: IPCamera url to mjpeg stream (http://<openhab_ip>:<openhab_port>/ipcamera/<camera_uid>/ipcamera.mjpeg)

snapshotUrl:

snapshotDescription: IPCamera url to snapshot stream (http://<openhab_ip>:<openhab_port>/ipcamera/<camera_uid>/ipcamera.jpg)

☐ Validate the configuration

Figure C.6: Mockup: Viewing application configuration details not complete

HubOS

FaceCamera

Not Installed

↓

▶

||

Manifest

Config

Rules

?

Warning: all the rules are not validated.

List of Rules

Rule name	Last triggered	Validated
ring_button_call_ring_handler...	None	☒
face_recognition_work_time_...	None	☐

Figure C.7: Mockup: Viewing the list of rules

HubOS

FaceCamera

Not Installed

Download

Run

Pause

Manifest

Config

Rules

face_recognition_work_time_access_to_camera_stream

Description:

Specify activation time access to camera stream for face recognition module

When:

Event:

system.time

Context:

GenericCronTrigger

Value:

0 0 9-18 ** ? *

Condition:

Then:

Action 1

Access:

NetworkClient

Type:

Service

Context:

Period:

1 hour

Concern module:

face_recognition

Host:

OpenHABServer →

☒

Validate the rule

Figure C.8: Mockup: Viewing the details of the first rule

HubOS

FaceCamera

↓▶||

Not Installed

Manifest

Config

Rules

ring_button_call_ring_handler_module

Description:

when ring button triggered call ring handler with auths

When:

Event:

mqtt.face_recognition

Context:

updated

Value:

notify_owners

Condition:

Then:

Action 1

Access:

NetworkClient

Type:

Service

Context:

Period:

5 minutes

Concern module:

notifier

Host:

OpenHABServer → telegramServer →

Action 2

Access:

pass_through

Type:

flow

Context:

Concern:

notifier

Value:

Validate the rule

Figure C.9: Mockup: Viewing the details of the second rule

90

HubOS

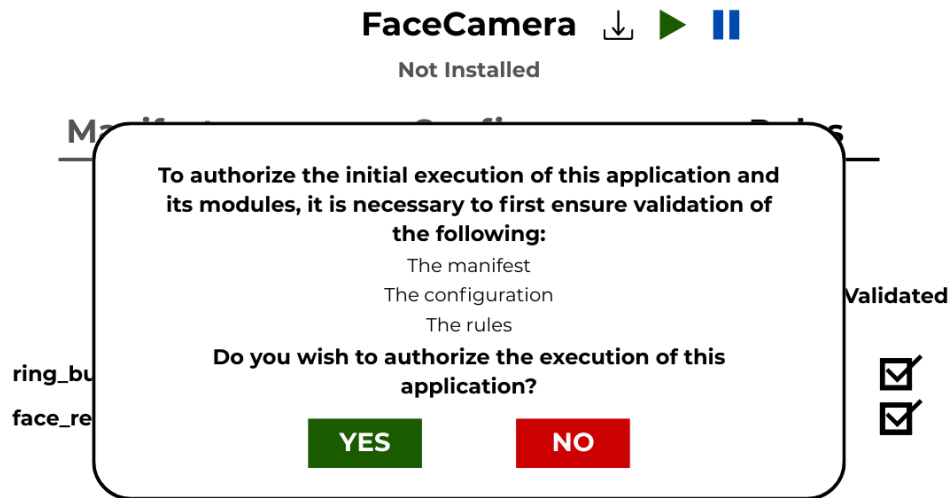


Figure C.10: Mockup: Application execution authorization modal

Bibliography

- [1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". In: *Computer Networks* 54.15 (2010), pp. 2787–2805. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2010.05.010>. URL: <https://www.sciencedirect.com/science/article/pii/S1389128610001568>.
- [2] S. Madakam, R. Ramaswamy, and S. Tripathi. "Internet of Things (IoT): A Literature Review". In: *Journal of Computer and Communications* 3 (2015), pp. 164–173. DOI: 10.4236/jcc.2015.35021. URL: <https://www.scirp.org/journal/paperinformation?paperid=56616>.
- [3] Sagar Bhat Pradyumna Gokhale Omkar Bhat. "Introduction to IOT". In: *International Advanced Research Journal in Science, Engineering and Technology* 5.1 (Jan. 2018). ISSN: 2393-8021. URL: https://www.researchgate.net/profile/Omkar-Bhat/publication/330114646_Introduction_to_IOT/links/5c2e31cf299bf12be3ab21eb/Introduction-to-IOT.pdf.
- [4] Joseph Bugeja, Andreas Jacobsson, and Paul Davidsson. "On Privacy and Security Challenges in Smart Connected Homes". In: *2016 European Intelligence and Security Informatics Conference (EISIC)*. 2016, pp. 172–175. DOI: 10.1109/EISIC.2016.044.
- [5] Igor Zavalyshyn et al. "Local-First Smart Home Applications with HubOS". In: *Proceedings of the 20th European Dependable Computing Conference (EDCC 2025)*. 11 April 2025. Lisbon, Portugal, Apr. 2025. URL: <http://hdl.handle.net/2078.1/297227>.
- [6] Google. *Google Home*. Google Home webpage. 2025. URL: <https://home.google.com/welcome/> Consulted on 15/8/2025.
- [7] Apple. *Apple Home app*. Apple Home-app web page. 2025. URL: <https://www.apple.com/home-app/> Consulted on 15/8/2025.
- [8] Wikipedia contributors. *Amazon Alexa* — *Wikipedia, The Free Encyclopedia*. 2025. URL: https://en.wikipedia.org/w/index.php?title=Amazon_Alexa&oldid=1289534257 Consulted on 16/5/2025.
- [9] Alex Hern. *Amazon staff listen to customers' Alexa recordings, report says*. 2019. URL: <https://www.theguardian.com/technology/2019/apr/11/> Consulted on 18/7/2025.
- [10] Jon Brodtkin. *Google workers listen to your "OK Google" queries—one of them leaked recordings*. 2019. URL: <https://arstechnica.com/information-technology/2019/07/google-defends-listening-to-ok-google-queries-after-voice-recordings-leak/> Consulted on 18/7/2025.

- [11] OpenHAB. *OpenHAB*. OpenHAB documentation. 2025. URL: <https://www.openhab.org/docs> Consulted on 12/8/2025.
- [12] Wikipedia contributors. *Home Assistant — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Home_Assistant&oldid=1300768313. 2025. Consulted on 17/7/2025.
- [13] Domoticz contributors. *Domoticz wiki*. https://wiki.domoticz.com/Main_Page. 2025. Consulted on 17/7/2025.
- [14] Wikipedia contributors. *Zigbee — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Zigbee&oldid=1298483913>. 2025. Consulted on 18/7/2025.
- [15] Florian Heimgaertner et al. "Scaling home automation to public buildings: A distributed multiuser setup for OpenHAB 2". In: *2017 Global Internet of Things Summit (GloTS)*. 2017, pp. 1–6. DOI: 10.1109/GIoTTS.2017.8016235.
- [16] OpenHAB. *OpenHAB Event Bus*. OpenHAB documentation. 2025. URL: <https://www.openhab.org/docs/developer/utils/events.html> Consulted on 12/8/2025.
- [17] Viam.com. *Set up people detection notifications in Home Assistant*. URL: <https://codelabs.viam.com/guide/home-assistant/index.html> Consulted on 20/7/2025.
- [18] Viam.com. *Plant watering with moisture sensor and pump*. URL: <https://codelabs.viam.com/guide/plant-watering-device-workshop/index.html> Consulted on 20/7/2025.
- [19] Paulus Schoutsen. *Disclosure: security vulnerabilities in custom integrations HACS, Dwains Dashboard, Font Awesome and others*. 2021. URL: <https://www.home-assistant.io/blog/2021/01/22/security-disclosure/> Consulted on 20/7/2025.
- [20] Robert Simmons. *Threat actor Banana Squad exploits GitHub repos in new campaign*. 2025. URL: <https://www.reversinglabs.com/blog/threat-actor-banana-squad-exploits-github-repos-in-new-campaign> Consulted on 20/7/2025.
- [21] Ravie Lakshmanan. *200+ Trojanized GitHub Repositories Found in Campaign Targeting Gamers and Developers*. 2025. URL: <https://thehackernews.com/2025/06/67-trojanized-github-repositories-found.html> Consulted on 20/7/2025.
- [22] Bako Ali and Ali Ismail Awad. "Cyber and Physical Security Vulnerability Assessment for IoT-Based Smart Homes". In: *Sensors* 18.3 (2018). ISSN: 1424-8220. DOI: 10.3390/s18030817. URL: <https://www.mdpi.com/1424-8220/18/3/817>.

- [23] Martin Kleppmann et al. “Local-first software: you own your data, in spite of the cloud”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: <https://doi.org/10.1145/3359591.3359737>.
- [24] Wikipedia contributors. *Google Nest (smart speakers)* — *Wikipedia, The Free Encyclopedia*. 2025. URL: [https://en.wikipedia.org/w/index.php?title=Google_Nest_\(smart_speakers\)&oldid=1278962150](https://en.wikipedia.org/w/index.php?title=Google_Nest_(smart_speakers)&oldid=1278962150) Consulted on 16/5/2025.
- [25] Wikipedia contributors. *Siri* — *Wikipedia, The Free Encyclopedia*. 2025. URL: <https://en.wikipedia.org/w/index.php?title=Siri&oldid=1290531410> Consulted on 16/8/2025.
- [26] Selen Hande Kabil, Hannah Muckenhirn, and Mathew Magimai-Doss. “On Learning to Identify Genders from Raw Speech Signal Using CNNs”. In: *Interspeech*. 2018, pp. 287–291.
- [27] Mousmita Sarma et al. “Emotion Identification from Raw Speech Signals Using DNNs”. In: *Interspeech*. 2018, pp. 3097–3101.
- [28] Nicholas Cummins, Alice Baird, and Bjoern W Schuller. “Speech analysis for health: Current state-of-the-art and the increasing impact of deep learning”. In: *Methods* 151 (2018), pp. 41–54.
- [29] Serkan Arik et al. “Neural Voice Cloning with a Few Samples”. In: *Advances in Neural Information Processing Systems*. Vol. 31. NIPS. 2018.
- [30] Jianwei Qian et al. “Speech sanitizer: Speech content desensitization and voice anonymization”. In: *IEEE Transactions on Dependable and Secure Computing* 18.6 (2019), pp. 2631–2642.
- [31] Yu Hu et al. “SpeechHide: A Hybrid Privacy-preserving Mechanism for Speech Content and Voiceprint in Speech Data Sharing”. In: *Seventh IEEE International Conference on Data Science in Cyberspace*. DSC. IEEE. 2022, pp. 345–352.
- [32] Sarina Meyer et al. “Speaker Anonymization with Phonetic Intermediate Representations”. In: *Proc. Interspeech 2022*. 2022, pp. 4925–4929.
- [33] Andreas Haas et al. “Bringing the web up to speed with WebAssembly”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: <https://doi.org/10.1145/3140587.3062363>.
- [34] Patrick Kelley, Lorrie Cranor, and Norman Sadeh. “Privacy as part of the app decision-making process”. In: *Apr.* 2013, pp. 3393–3402. DOI: 10.1145/2470654.2466466.

- [35] Adrienne Porter Felt et al. "Android permissions: user attention, comprehension, and behavior". In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. SOUPS '12. Washington, D.C.: Association for Computing Machinery, 2012. ISBN: 9781450315326. DOI: 10.1145/2335356.2335360. URL: <https://doi.org/10.1145/2335356.2335360>.
- [36] Serena Zheng et al. "User Perceptions of Smart Home IoT Privacy". In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (Nov. 2018). DOI: 10.1145/3274469. URL: <https://doi.org/10.1145/3274469>.
- [37] Eric Zeng, Shrirang Mare, and Franziska Roesner. "End user security & privacy concerns with smart homes". In: *Proceedings of the Thirteenth USENIX Conference on Usable Privacy and Security*. SOUPS '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 65–80. ISBN: 9781931971393.
- [38] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. "A role-based access control model and reference implementation within a corporate intranet". In: *ACM Trans. Inf. Syst. Secur.* 2.1 (Feb. 1999), pp. 34–64. ISSN: 1094-9224. DOI: 10.1145/300830.300834. URL: <https://doi.org/10.1145/300830.300834>.
- [39] Vincent Hu, D. Kuhn, and David Ferraiolo. "Attribute-Based Access Control". In: *Computer* 48 (Feb. 2015), pp. 85–88. DOI: 10.1109/MC.2015.33.
- [40] patriksimek. *vm2*. github. 2025. URL: <https://github.com/patriksimek/vm2> Consulted on 12/5/2025.
- [41] The MITRE Corporation. *CVE - Common Vulnerabilities and Exposures*. <https://www.cve.org/>. Accessed: 2025-07-18. 2025.
- [42] Manuj Aggarwal. "A Study of CVSS v4.0: A CVE Scoring System". In: *2023 6th International Conference on Contemporary Computing and Informatics (IC3I)*. Vol. 6. 2023, pp. 1180–1186. DOI: 10.1109/IC3I59117.2023.10397701.
- [43] leesh3288 and SeungHyun Lee. *Sandbox Escape in vm2 - CVE-2023-37466*. <https://github.com/patriksimek/vm2/security/advisories/GHSA-cchq-frgv-rjh5>. 2023. Consulted on 18/3/2025.
- [44] leesh3288 and SeungHyun Lee. *Sandbox Escape in vm2 - CVE-2023-37903*. <https://github.com/patriksimek/vm2/security/advisories/GHSA-g644-9gfx-q4q4>. 2023. Consulted on 18/3/2025.
- [45] leesh3288. *Sandbox Escape in vm2@3.9.19 via custom inspect function*. <https://gist.github.com/leesh3288/e4aa7b90417b0b0ac7bcd5b09ac7d3bd>. 2023. Consulted on 18/3/2025.
- [46] leesh3288. *Sandbox Escape in vm2@3.9.19 via Promise[[@@species]]*. <https://gist.github.com/leesh3288/f693061e6523c97274ad5298eb2c74e9>. 2023. Consulted on 18/3/2025.
- [47] Ecma International. *ECMAScript 2022 Language Specification – 27.2.5.4 Promise.prototype*. <https://tc39.es/ecma262/2022/multipage/control-abstraction-objects.html>. 2022. Consulted on 18/3/2025.

- [48] Node.js Contributors. *Node.js source code — internal/errors.js*. <https://github.com/nodejs/node/blob/92a938b4dd318a8732110da03f1f666db2e96f97/lib/internal/errors.js>. 2023. Consulted on 18/7/2025.
- [49] Node.js Contributors. *Node.js source code — wasm web api.js*. https://github.com/nodejs/node/blob/92a938b4dd318a8732110da03f1f666db2e96f97/lib/internal/wasm_web_api.js. 2023. Consulted on 18/7/2025.
- [50] Mosquitto. *Mosquitto*. Mosquitto documentation. 2025. URL: <https://mosquitto.org/documentation/> Consulted on 12/8/2025.
- [51] Mosquitto. *Mosquitto - Dynamic Security Plugin*. Mosquitto documentation. 2025. URL: <https://mosquitto.org/documentation/dynamic-security> Consulted on 12/8/2025.
- [52] The V8 Team. *V8 JavaScript Engine*. <https://v8.dev/>. 2024. Consulted on 18/7/2025.
- [53] node.js. *Node.js vm*. Documentation/api. 2025. URL: <https://nodejs.org/api/vm.html> Consulted on 12/5/2025.
- [54] laverdet. *Isolated-vm*. github page. 2025. URL: <https://github.com/laverdet/isolated-vm> Consulted on 12/5/2025.
- [55] Google. *gVisor*. Github page. 2025. URL: <https://github.com/google/gvisor> Consulted on 12/5/2025.
- [56] Lennart Espe et al. "Performance Evaluation of Container Runtimes". In: *Performance Evaluation of Container Runtimes*. Jan. 2020, pp. 273–281. DOI: 10.5220/0009340402730281.
- [57] Anjali, Tyler Caraza-Harter, and Michael M. Swift. "Blending containers and virtual machines: a study of firecracker and gVisor". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 101–113. ISBN: 9781450375542. DOI: 10.1145/3381052.3381315. URL: <https://doi.org/10.1145/3381052.3381315>.
- [58] Google. *gVisor*. documentation. 2025. URL: <https://gvisor.dev/docs/> Consulted on 12/5/2025.
- [59] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. "An introduction to docker and analysis of its performance". In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), p. 228.
- [60] Docker. *Docker Engine API*. Docker documentation. 2025. URL: <https://docs.docker.com/reference/api/engine/version/v1.48/> Consulted on 12/8/2025.
- [61] Apocas. *Dockerode*. github page. 2025. URL: <https://github.com/apocas/dockerode/> Consulted on 12/8/2025.
- [62] berstend. *straightforward Proxy*. URL: <https://github.com/berstend/straightforward> Consulted on 5/6/2025.
- [63] OpenHab. *IP Camera*. Openhab doc. 2025. URL: <https://www.openhab.org/addons/bindings/ipcamera/> Consulted on 26/4/2025.

- [64] OHF-Voice. *Wyoming*. URL: <https://github.com/OHF-Voice/wyoming>.
- [65] rhasspy. *rhasspy3*. URL: <https://github.com/rhasspy/rhasspy3/>.
- [66] Valentin Kola. *HubOS: Improving and Redesigning a Local-First, Privacy-by-Design Operating System for Smart Home Applications*. https://github.com/valk-ulb/hubos_2. Accessed: 2025-11-29. 2025. Consulted on 16/11/2025.
- [67] Kai Kreuzer. *openHAB for business purposes*. <https://community.openhab.org/t/openhab-for-business-purposes/1460/2>. Forum post on the openHAB Community. 2015. Consulted on 16/11/2025.
- [68] Jean Tirole and Josh Lerner. *The Scope of Open Source Licensing*. Tech. rep. w9363. NBER Working Paper No. w9363. National Bureau of Economic Research, Dec. 2002. URL: <https://ssrn.com/abstract=359303> Consulted on 16/11/2025.
- [69] Arnoud Engelfriet. "Choosing an Open Source License". In: *IEEE Software* 27.1 (2010), pp. 48–49. DOI: 10.1109/MS.2010.5. Consulted on 16/11/2025.
- [70] Open Source Initiative. *Eclipse Public License version 2.0*. <https://opensource.org/license/epl-2-0>. Version 2.0; SPDX short identifier: EPL-2.0. 2025. Consulted on 12/11/2025.
- [71] Open Source Initiative. *The MIT License*. <https://opensource.org/license/mit>. Version N/A; SPDX short identifier: MIT. 2025. Consulted on 12/11/2025.
- [72] Open Source Initiative. *Apache License, Version 2.0*. <https://opensource.org/license/apache-2-0>. Version 2.0; SPDX short identifier: Apache-2.0. 2025. Consulted on 12/11/2025.
- [73] Open Source Initiative. *The BSD-2-Clause License*. <https://opensource.org/license/bsd-2-clause>. Version N/A; SPDX short identifier: bsd-2-clause. 2025. Consulted on 12/11/2025.
- [74] The Linux Foundation. *Developer Certificate of Origin — Version 1.1*. <https://developercertificate.org/>. Consulted 9 december 2025. 2006. Consulted on 16/11/2025.
- [75] Axios contributors. *Axios: Promise based HTTP client for the browser and node.js*. Version 1.9.0. 2025. URL: <https://github.com/axios/axios> Consulted on 16/11/2025.
- [76] base64 contributors. *base-64: Base64 encoding / decoding*. Version 1.0.0. 2025. URL: <https://github.com/mathiasbynens/base64> Consulted on 16/11/2025.
- [77] bcrypt.js contributors. *bcrypt.js: Optimized bcrypt in JavaScript*. Version 5.1.1. 2025. URL: <https://github.com/kelektiv/node.bcrypt.js> Consulted on 16/11/2025.
- [78] Commander.js contributors. *Commander.js: The complete solution for node.js command-line interfaces*. Version 14.0.0. 2025. URL: <https://github.com/tj/commander.js> Consulted on 16/11/2025.
- [79] cors contributors. *cors: Express middleware to enable CORS*. Version 2.8.5. 2025. URL: <https://github.com/expressjs/cors> Consulted on 16/11/2025.

- [80] CryptoJS contributors. *CryptoJS: Standard and secure cryptographic algorithms in JavaScript*. Version 1.0.1. 2025. URL: <https://github.com/brix/crypto-js> Consulted on 16/11/2025.
- [81] Express contributors. *Express: Fast, unopinionated, minimalist web framework for node.js*. Version 5.1.0. 2025. URL: <https://github.com/expressjs/express> Consulted on 16/11/2025.
- [82] node-http-proxy contributors. *node-http-proxy: A full-featured HTTP proxy library for node.js*. Version 1.18.1. 2025. URL: <https://github.com/http-party/node-http-proxy> Consulted on 16/11/2025.
- [83] node-jsonwebtoken contributors. *jsonwebtoken: JSON Web Token implementation for node.js*. Version 9.0.2. 2025. URL: <https://github.com/auth0/node-jwebtoken> Consulted on 16/11/2025.
- [84] MQTT.js contributors. *MQTT.js: Client library for the MQTT protocol*. Version 4.3.8. 2025. URL: <https://github.com/mqttjs/MQTT.js> Consulted on 16/11/2025.
- [85] node-postgres contributors. *node-postgres: PostgreSQL client for node.js*. Version 8.14.1. 2025. URL: <https://github.com/brianc/node-postgres> Consulted on 16/11/2025.
- [86] tar-fs contributors. *tar-fs: Tar archive creation and extraction using file system streams*. Version 3.0.8. 2025. URL: <https://github.com/mafintosh/tar-fs> Consulted on 16/11/2025.
- [87] uuid contributors. *uuid: RFC-compliant UUID generation library for JavaScript*. Version 11.1.0. 2025. URL: <https://github.com/uuidjs/uuid> Consulted on 16/11/2025.
- [88] Vitest contributors. *Vitest: Fast unit testing framework for modern JavaScript/TypeScript*. Version 3.1.1. 2025. URL: <https://github.com/vitest-dev/vitest> Consulted on 16/11/2025.
- [89] dotenv contributors. *dotenv: Loads environment variables from a .env file*. Version 16.0.3. 2025. URL: <https://github.com/motdotla/dotenv> Consulted on 16/11/2025.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl