

École polytechnique de Louvain

Smart firewall for IoT and Smart Home applications

Authors: **Dariusz MOHANDÉS, Justin OLIVA**

Supervisor: **Ramin SADRE**

Readers: **François DE KEERSMAEKER, Gorby KABASELE, Ramin SADRE, Igor ZAVALYSHYN**

Academic year 2021–2022

Master [120] in Computer Science

Acknowledgements

First of all, we would like to thank our supervisor, Ramin Sadre, and one of our readers, François De Keersmaeker, for their guidance and advice over the year, but also for the constructive remarks they provided during the revision of our thesis.

Then, we would also like to thank the rest of the readers, Gorby Kabasele and Igor Zavalyshyn, who agreed to read and evaluate our work.

Finally, we would like to thank our family, especially our parents, who have always supported us and given us the opportunity to get to this point.

Contents

1	Introduction	5
1.1	Objective	5
1.2	Overview	6
2	Background	7
2.1	Internet of Things	7
2.1.1	Smart Homes	7
2.1.2	IoT features	8
2.2	Network stack	9
2.2.1	Push and pull architectures	9
2.2.2	MQTT	10
2.2.3	CoAP	12
2.3	Security issues	14
2.3.1	Vulnerabilities	15
2.3.2	Attack examples	16
2.4	Countermeasures	17
2.4.1	Firewalls	17
2.4.2	Intrusion detection systems	19
2.5	Finite state machine	20
2.5.1	Deterministic finite automata	20
2.5.2	Non-deterministic finite automata	20
2.5.3	Probabilistic finite automata	21
2.6	Related work	21
3	Design and implementation	25
3.1	Theoretical considerations	25
3.1.1	Security goals	25
3.1.2	Assumptions	26
3.1.3	Threat model	27
3.2	Architecture	29
3.2.1	Centralized firewall	29
3.2.2	Components	31
3.2.3	Component interactions	34
3.2.4	Terminology	34
3.3	Protocol decoder	36
3.3.1	Content abstraction	37
3.3.2	Protocol behavior abstraction	37
3.4	Context	40
3.4.1	Context module	41
3.4.2	Triggers and network modules	41
3.4.3	FSM building	42
3.4.4	State inconsistency	44
3.4.5	Path likelihood	44

3.5	Parser and handler	45
3.5.1	Configuration file	45
3.5.2	Building Nftables	46
3.6	Workflow	47
3.6.1	Initialization	47
3.6.2	Filtering process	47
3.6.3	State update	49
4	Evaluation	50
4.1	Experimental setup	50
4.1.1	Simulation	50
4.1.2	Deployment	51
4.2	Latency	51
4.2.1	Burst test	52
4.2.2	Scaling test	55
4.3	Throughput	56
4.4	Context module assessment	57
4.5	Use cases	58
4.5.1	Home presence	59
4.5.2	Night time	59
4.5.3	Vacuum cleaner	60
5	Discussion and limitations	62
5.1	Security	62
5.1.1	Encryption	62
5.1.2	Authentication	64
5.1.3	Packet state storage	65
5.2	Implementation choices	66
5.2.1	Programming language	66
5.2.2	Low layers matching	66
5.2.3	Firewall layout	66
5.3	Context	67
5.3.1	State explosion	67
5.3.2	Inter-arrival time	69
6	Conclusion	71
A	Input file syntax	77
A.1	Home presence example	77
A.2	Night example	79
A.3	Vacuum cleaner example	80
B	Build time	86

1 Introduction

With the advent of the Internet of Things (IoT), smart objects have become increasingly ubiquitous. Today, everything is connected, from the small light bulb in the living room to the autonomous car. However, the last decade has also witnessed some of the most significant mass attacks. Indeed, while the industry has worked to improve the IoT, constantly bringing new possibilities, it has also introduced new vulnerabilities and ways to exploit connected devices with malicious intents. Smart homes are no exception to this phenomenon, bringing new IoT capabilities that open up new security breaches directly to the user's home. Unfortunately, traditional countermeasures such as firewalls are not always suitable for this type of scenario, as they are static and cannot adapt to changes due to real world conditions. This is why the research community has been working in recent years to improve security measures for the IoT. Therein lies our research, we want to bring an innovative way to secure smart homes by deploying a dynamic firewall that monitors traffic in a centralized fashion, but also adapts to the context created by the complex behaviors of interconnected devices.

1.1 Objective

This study is mainly conducted in smart homes. The latter are composed of a set of devices interacting with each other to improve the well-being of the home's occupants. In this scenario, we aim at providing a centralized smart firewall able to adapt its policies in real time based on observed interactions. To do this, we ask the user to define a set of policies limiting the interactions between devices to well-defined behaviors, thus preventing potential abuse by an adversary.

To achieve our main goal, a major challenge must be overcome, namely the heterogeneity of application layer protocols used in the IoT environment. Indeed, due to their very nature, IoT devices often rely on their own protocols on each layer of the OSI model [1]. Especially on the application layer where HTTP is not always the most suitable protocol for them. To handle this variety, our firewall provides an abstraction layer. This layer abstracts packets into a generic format before discovering which protocol they belong to by comparing them to a set of supported protocol behaviors.

Then, another challenge is to be able to represent the smart home and manage the interactions between the devices. In our study, the smart home is represented by the context, which can be viewed as the set of states of each device reflecting real world conditions. For example, whether a radio is currently on or off, the last value measured by a humidity sensor, etc. As long as a device is working as expected, its state represents a value in the physical world, such as the current temperature. In the firewall, it is modeled by a deterministic finite state machine (FSM) whose states correspond to the actual states of the smart home. Physical conditions are defined by fields attached to the devices and used to perform transitions on the FSM. To these transitions are assigned a set of triggers, responsible for enabling or disabling

specific interactions whenever the firewall observes network traffic. Thus, it is able to adapt to real-world events in real time, block abnormal behaviors and alert the user through alarms.

1.2 Overview

Our work is organized in four main chapters that we detail here. The firewall implementation can be found on this [repository](#).

Chapter 2: Background This chapter introduces the environment in which the research is carried out, namely the IoT and in particular smart homes. First, we show the specific capabilities brought by the IoT. Then, we provide a summary of the network stack used by smart devices and pay special attention to the application layer, presenting the protocols we support: MQTT and CoAP. Then, we focus on security issues in this environment, highlighting vulnerabilities and giving some examples of attacks. On the other hand, we present pre-existing countermeasures, mainly firewalls and intrusion detection systems. Finally, we briefly discuss finite state automata and their variants, since we use this concept to represent the smart home context.

Chapter 3: Design and implementation Here, we take a deeper dive into the working principle of the firewall. We first present important theoretical considerations, such as our assumptions and threat model. Then, we give a high-level overview of the architecture and define the different components that constitute it. Next, we describe in detail two important entities, namely the protocol decoder and the context. Finally, we describe the parsing process before concluding with a review of the overall system workflow.

Chapter 4: Evaluation This chapter summarizes the performance of the firewall, but also identifies its limitations. It begins by describing the complete configuration of the environment used in the evaluation. It then presents latency and throughput tests, followed by an evaluation of the context module. At the end, it includes some real-world use cases that demonstrate the usefulness of the system.

Chapter 5: Discussion and limitations Our solution is not flawless and this chapter discusses its main drawbacks but also provides insight into other alternative approaches that could have been explored. We first address security issues, then revisit some implementation choices before addressing some shortcomings of the context representation.

2 Background

This second section introduces the framework and helps the reader understand the setting in which the research is conducted and the problem we are trying to solve. To begin, we describe the Internet of Things, and in particular smart homes. Next, we explore the network stack used in this environment, and its security issues. Then, we introduce some solutions for these issues. Apart from that, we give a quick reminder about the finite state machine concept that we use in our work. Finally, we highlight some similar works and explain how they contributed to our research.

2.1 Internet of Things

The term "Internet of Things" (IoT) is now well-known. Despite the lack of consensus on its definition, we can understand it through the concept of "smart" objects also called "things". The capabilities of these objects are extended through network connectivity and computing power, allowing them to generate, exchange and consume data with minimal human intervention. They are everywhere, in security cameras, smart TVs and even in autonomous cars. The network created by the interconnection of these physical objects forms the IoT. If the latter is already omnipresent in our daily life with more than 12 billion connected devices in 2021, the trend should reach even more considerable figures with nearly 30 billion devices in 2025 [2].

IoT covers a wide range of applications and at different levels, from industrial to personal use cases, as outlined by Khanna *et al.* [3]. Smart mobility provides transportation assistance through the use of systems such as vehicle ad hoc networks [4] to improve the driving experience and make it safer. The smart grid improves the electrical grid by reacting to local changes in real time. Environmental monitoring observes weather conditions, studies the ecosystem and the effects of certain environmental phenomena such as global warming. The Internet of Medical Things [5] connects health services to a computer network and provides increased assistance to people with health problems. In the industry, modern connected equipment improves production rates. These use cases are just the tip of the iceberg, but already show us the many possibilities that IoT offers.

2.1.1 Smart Homes

The coverage of IoT is vast and the field of research is growing every day, but for our study, we decided to narrow it down to smart homes. A smart home can be defined as suggested by Harper [6] as "a residence equipped with computing and information technology, which anticipates and responds to the needs of the occupants, working to promote their comfort, convenience, security and entertainment through the management of technology within the home and connections to the world beyond". Many definitions have been proposed and this one fits our own vision of the concept. We find mainly three types of devices inside a smart home:

- The *sensors* are responsible for collecting data for further processing. This data is often related to the user's activity. For example, a user returning home will trigger the motion sensor or opening a window will change the carbon dioxide level. Information can be even more user-related, such as health measurements, for example heart rate, which can indicate a user's sleep activity.
- The *actuators* which, as their name suggests, act on the environment. In general, these devices only respond to commands and do not reason. A typical example is a light bulb that turns on or off or shutters that open or close.
- The controller, often referred to as a *hub*. The hub is like the brain of the network and manages the interactions between sensors and actuators. A hub can for example decide to turn off the lights at midnight. It is usually unique and can be physically present in the home like a Google Home [7] or in the cloud with services like IFTTT [8], but its presence is not mandatory. Some protocols allow devices to interact directly or through an intermediary called a *broker* to facilitate one-to-many relationships. The concept of a broker is developed in Section 2.2.2.

2.1.2 IoT features

In addition to covering a wide range of applications, the IoT brings its share of new features, as described by Zhou *et al.* [9]. The most interesting one for our research is definitely interdependence. Connected devices aim to improve the comfort and well-being of end users. To this end, they tend to require as little human interaction as possible, letting the network do the work autonomously. As a result, the devices not only communicate with each other, but also depend on each other and, in fact, control each other through various user-defined relationships. In addition, devices are subject to changes in their environment. Another important aspect is the constrained nature of IoT devices. Most of the time, devices are designed to be lightweight and small in size. This limits their computing and storage capacity. Sometimes, they also do not have access to a constant power supply, which leads to stringent power consumption requirements. In addition, some of them, such as the devices used in vehicle systems, must operate in real time with strict timing restrictions.

Since they have to accommodate many use cases, this increases diversity. The requirements are very heterogeneous, each device has its own specifications tailored to its purpose. A small humidity sensor is quite different from an industrial machine, but they may work together. What the authors of the paper [9] call "myriad" refers to the fact that IoT devices are proliferating rapidly and producing an astounding amount of data. They are pervasive in our daily lives, collecting our biological information continuously, even through mobile devices. These, such as wearables, move from one network environment to another and thus become suitable guinea pigs for the spread of malware. Finally, the authors emphasized the ubiquitous aspect of "things". It is very likely that in the near future we will not only use smart objects, but we will also depend on them just as we need air and water.

2.2 Network stack

This heterogeneity in terms of use cases is undoubtedly one of the greatest benefits of the IoT, but such diversity makes the task of standardizing communication really difficult. Most "things" are constrained, strongly coupled to their use cases and do not necessarily share the same characteristics with each other. Moreover, because IoT is such an emerging market, many manufacturers have jumped into it without agreeing on the technology to use. In this scenario, sharing a common consensus is almost utopian and working with different device families can quickly become challenging.

A wide variety of protocols are present at each layer, as outlined by Triantafyllou *et al.* [10] and summarized in Figure 1. The physical and data link layers are designed for wireless communication. Protocols such as WiFi, 802.15.4 or Bluetooth are often used for short distances, for example in smart homes. For long range as in wireless sensor networks (WSNs), LoRa [11] or Sigfox [12] can be considered. A WSN [13] is a network composed of a set of spatially distributed sensors, often constrained, and one or more receiver nodes, typically to monitor a set of physical conditions in real time. The usual TCP/IP or UDP/IP is widely used at the network/transport layer, but considering a lightweight version of IPv6 such as 6LowPAN [14] is a good compromise for constrained devices. Finally, the application layer which contains valuable information to examine devices behaviors. Note that some specifications such as ZigBee [15] or Z-Wave [16] directly implement an entire suite of protocols on several layers. In our study, we paid a lot of attention to the application layer. We were particularly interested in two popular protocols in the IoT: MQTT and CoAP. These protocols are designed in two opposing message exchange patterns that we discuss hereafter.

Physical/Data Link	Network	Transport	Application	Multiple layers
WiFi, 802.15.4, Bluetooth, LoRa, Sigfox	IPv4, IPv6, 6LowPAN	TCP, UDP	MQTT, CoAP	ZigBee, Z-Wave

Figure 1: Network stack summary.

2.2.1 Push and pull architectures

Before presenting the application protocols supported by our firewall, it is interesting to focus on one main feature that divides the set of IoT application protocols: *push* and *pull* behavior. As mentioned earlier, there is a wide variety of IoT protocols and one of the causes of this heterogeneity is the fact that some protocols are push protocols and others are pull protocols.

In push protocols, according to Trecordi *et al.* [17], the client maintains a connection with a server to get the information. As shown in Figure 2, whenever an

event occurs or a resource changes, the client is automatically informed. For example, a cell phone is always connected to the mobile network so that when a caller tries to make a call, the mobile network transmits the call via the active connection it has with the phone: this is a push behavior. An important remark is that both parties of the communication may not maintain an active end-to-end connection, as in the previous example, they may maintain a connection with an intermediate server (the mobile network for example).

Pull protocols, on the other hand, do not require both parties to maintain an active connection. In this case, the client periodically connects to the server, checks for new events or resource changes, and finally disconnects, as depicted in Figure 2. Pull protocols result in a higher network load, as more messages are exchanged over the network, but in exchange, the connection is maintained only as long as it takes to transfer the data.

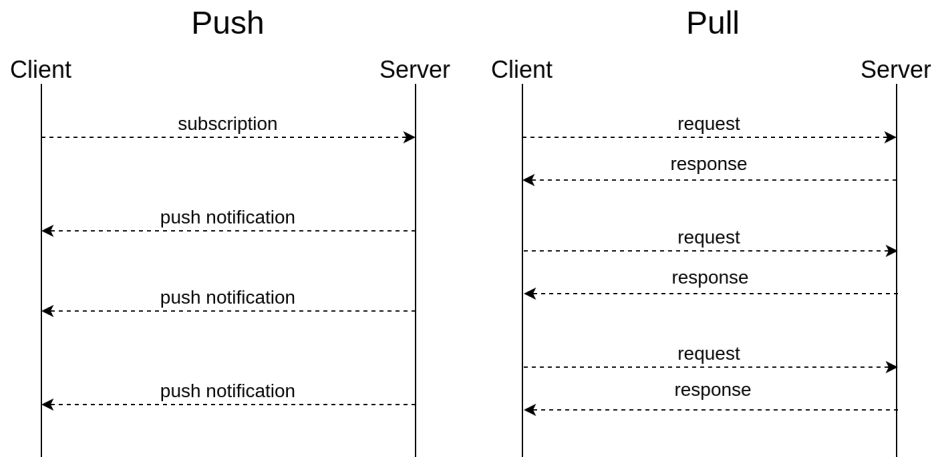


Figure 2: Push vs Pull architecture.

2.2.2 MQTT

The Message Queuing Telemetry Transport (MQTT) [18] protocol, developed since 1999 by the Organization for the Advancement of Structured Information Standards, is a lightweight application layer protocol suitable for IoT devices. It is a push protocol, based on the publish-subscribe model, and generally running on top of TCP/IP. Unlike the classical request-response model, the publish-subscribe model is based on an asynchronous scheme. Within this scheme, devices can act as publishers and/or subscribers and interact through an intermediate node called the broker. The broker manages subscriptions by registering subscribers to different topics. Topics are a way to identify the data streams produced by publishers, who typically send their data periodically. The relationships between publishers and subscribers are characterized as "one-to-many", a producer can have many subscribers and vice versa, a single device can also play both roles.

Let's consider the environment in Figure 3 with three devices, a smoke detector,

a fire alarm and a sprinkler. In this scenario, the detector publishes the smoke level to the broker with a topic such as "smoke_level". Then the fire alarm and sprinkler are subscribed to this topic through the broker. In case of a fire, the smoke alarm starts publishing a high smoke level which is then published to the subscribers. In this way, the fire alarm and the sprinkler can be activated.

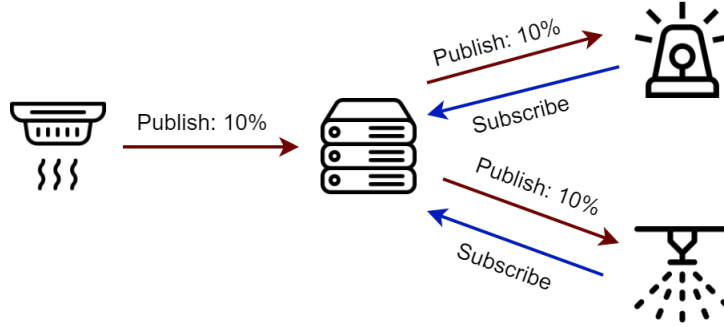


Figure 3: Publish and subscribe model.

Although we do not need to dive deep into the implementation, it is interesting to show the most important mechanisms of the protocol. Each MQTT control packet consists of a fixed header that can be followed by a variable header and finally the payload. To get an idea of what a MQTT packet looks like, Figure 4 shows the format of a publish/subscribe message. The only mandatory part is the fixed header, shown in blue, which consists of a type and some flags. A typical use of these flags is the implementation of Quality of Service (QoS) [19]. MQTT offers three levels of QoS, from level 0 to level 2. In the first level, messages should be delivered at most once, in the second level they should be delivered at least once, and in the last level exactly once. These QoS are verified by issuing various types of acknowledgements, but in our case there are none because we focus on the lowest QoS. Since the broker is the intermediary, it is always present during a message exchange. In fact, it behaves like a server while the other devices are clients with unique identifiers. The most important message types are:

- **CONNECT:** A connection request from a client to the server. As mentioned, MQTT is a connection-based protocol, a device must be connected before initiating any exchange.
- **PUBLISH:** A client publishes information. This can be a client publishing to the broker or the broker publishing to a client. Other messages are derived from this one depending on the quality of service
- **SUBSCRIBE:** A client subscribes to a topic.
- **PINGREQ:** A ping from the client to the server, useful to maintain an active TCP connection.

For security, MQTT supports payload encryption and credential authentication combined with the client ID. Since the network layer is TCP, it is also possible to configure TLS/SSL X.509 certificates [20] for encryption and authentication.

Bit	7	6	5	4	3	2	1	0
Byte 1	MQTT Control Packet Type				Flags specific to each MQTT Control Packet type			
Byte 2	Message length (M)							
Byte 3	Topic length (T)							
Byte 4								
Byte 5 to 5 + T	Topic							
Byte 5 + T + 1 to 5 + M - 1	Payload							

Figure 4: MQTT publish/subscribe message format.

MQTT broker

As explained earlier, MQTT uses a broker as an intermediate server to exchange messages between subscribers and publishers. All messages sent by the publishers and subscribers pass through the broker, as opposed to an end-to-end communication. This has some advantages: The first is to offload the complexity into a less constrained device that keeps track of the subscribers of each topic and forwards messages to them. The broker therefore allows for better scalability if the number of devices increases. The second benefit of the broker is the *last will* feature, a publisher can set up a last will in case it unexpectedly stops working. This usually happens if a device stop sending keep-alive messages for an extended period of time. In this scenario, the broker will send a message to all subscribers telling them what the publisher wanted to say. A third advantage of the broker is the *retain* feature, all published messages marked as retain will be stored in the broker and if some subscribers join later, they will receive those messages. Finally, another advantage is that since everything has to go through the broker, it is easier to monitor the traffic. However, there is a drawback if no authentication is used with the clients. In this case, anyone can send anything on the desired topic since no end-to-end TCP connection is maintained between the publisher and the subscriber.

2.2.3 CoAP

The Constrained Application Protocol (CoAP) [21] is a web transfer protocol designed for use with constrained (e.g., low power, lossy) nodes and networks such as WSNs. It is a pull protocol that has been implemented mostly by The Internet Engineering Task Force (IETF) since 2014. The CoAP protocol has many similarities with the HTTP protocol, as it is intended to be easily translated into the latter for web integration. CoAP is primarily built upon a request-response model where each message has been compressed to contain the minimum amount of information required in the fewest number of bytes. The general format of the messages is depicted in Figure 5 which comes from RFC 7252 [21], the most important fields are briefly described below to give the reader an insight of the protocol.

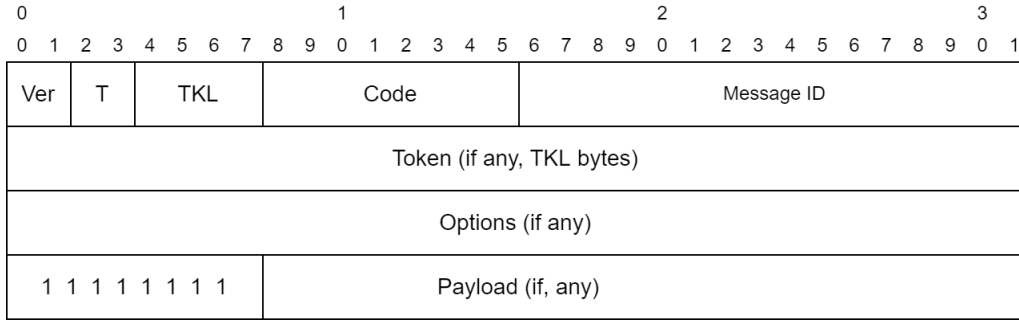


Figure 5: CoAP message format.

- T: Indicates the type of the message (confirmable, non-confirmable, ack or reset)
- Code: This is the request/response code, very similar to HTTP. If the message is a request, the code can be GET, POST, PUT, etc. and if it is a response, the code can be created, deleted, valid, etc.
- Message ID: The message ID must be unique for each message and is used to identify duplicate requests.
- Token: This is another identifier, but of a different kind than the message ID. Its length is identified by the TKL field. The token is used to bind a response to its request. Unlike the message ID, the token is the same for each message in case of duplication.
- Options: An option consists of the option delta, the option length and the option value. The message can contain as many options as desired, they are all consecutive. The option number represents the type of option, and the delta of an option is obtained by calculating the option number minus the delta of the previous option. For example, a packet containing two options, one of type Uri-Port (the server port, option number = 7) and the other of type Uri-Path (the resource path, option number = 11) will have a first delta of $7 - 0 = 7$ (no option before) and a second one of $11 - 7 = 4$. The option numbers are found by adding the delta of the current option and the previous option. In this case, $7 + 0 = 7$ and $7 + 4 = 11$.
- Delimiter (11111111): A byte set to 255 indicating the beginning of the payload.

CoAP Observer

CoAP is mainly a pull protocol but it can also behave like a push protocol thanks to its observe option [22]. By using this option with a GET message, a client can subscribe to a certain topic directly from the publisher, without going through a broker. In CoAP, the resource that is being observed is called the *subject*, the client that is interested in changes to the resource is the *observer*. The *registration* is the

mechanism by which a client becomes the observer of a subject. The *notification* is the message by which the editor notifies a change in the resource. It is important to specify at this point that the same token must be used for both the registration and the notification. An example of a CoAP observer is shown in Figure 6.

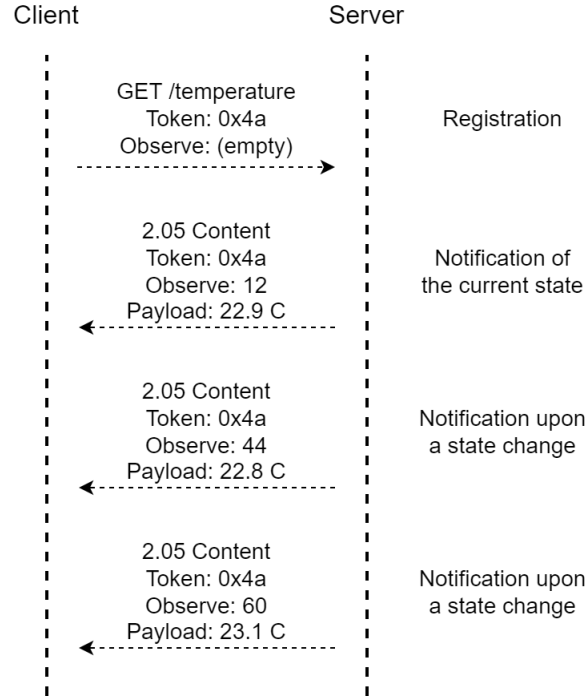


Figure 6: Observing a resource in CoAP.

2.3 Security issues

The IoT paradigm comes with innovative opportunities, but the introduction of new technologies also brings new vulnerabilities. This profitable business and the absence of regulation has led the industry to neglect security considerations. Taking advantage of customers' unawareness, manufacturers put production before security. This negligence allows adversaries to exploit vulnerable devices without much effort. Recent large-scale attacks highlight the growing threat behind the IoT revolution.

In October 2016, a distributed denial-of-service attack carried out by the Mirai botnet [23] on the service provider Dyn took down hundreds of websites for several hours. Among them, popular sites such as Twitter, Netflix, Reddit or Github. One might have expected an industry reaction after the public release of the Mirai code base. Instead, the code base has been leveraged and has inspired new botnet implementations like the BitTorrent DHT-based Hajime botnet. The number of infected machines more than doubled, as Kolias *et al.* highlighted [24]. Fortunately, this lack of security has raised many concerns and has drawn the attention of the research community.

2.3.1 Vulnerabilities

It is worth remembering the basic principles of computer security: Confidentiality, Integrity and Availability (CIA). First, confidentiality involves keeping data secret or private. Then, integrity involves ensuring that data is authentic and unaltered. Finally, availability involves keeping data accessible to authorized users. These three principles form the CIA triad. Vulnerabilities abound in the IoT domain, as shown by Butun *et al.* [25]. These flaws threaten the most critical security principles, violating the triad. An attacker is not only capable of accessing private data, but also of altering it and potentially disabling a resource for legitimate users. Neshenko *et al.* [26] classified the weaknesses into three layers:

- **Device-based:** The majority of IoT devices operate in an unguarded manner and are basically open to physical manipulation. This means that it is frequently possible for an attacker to gain unrestricted access to a device and the data it stores. This can serve as a starting point for further attacks. Another specific vulnerability of IoT devices is their constrained nature. Insufficient energy harvesting limits their capabilities and makes them targets for energy leaks. These considerations also increase the difficulty of adopting secure encryption and authentication schemes. To top it off, because these devices are so limited, they often omit critical security components such as memory management unit or address space randomization, making them more predictable in terms of program execution.
- **Network-based:** As we discussed earlier, the IoT networking stack is well supplied. Unfortunately, this provisioning comes with a handful of vulnerabilities. Through them, an attacker is able to gain remote access. These vulnerabilities include inadequate authentication due to low complexity, the risk of authentication key corruption or even no authentication at all. Weak or non-existent encryption endangers data confidentiality. Needlessly open ports also provide a loophole for malicious activity.
- **Software-based:** Security also relies on the software application that runs in a smart appliance. Even with a protected physical environment and a secure network stack, there is no guarantee regarding the implementation of a software itself. Careless programming practices usually offer many opportunities to adversaries. Insufficient access control results in brute forcing, which is one of the easiest and most effective ways to corrupt a device. This technique has been proven with botnets such as Mirai, which we showcased earlier. Another problem is improper update management that regularly leaves reported vulnerabilities unpatched for a long time. Softwares do not always support automated patch update mechanisms, or do not run them recurrently enough. And even if these mechanisms exist, lack of integrity can lead to corrupted updates. Finally, incomplete auditing mechanisms make it challenging to discover flawed systems.

In our study, we mainly focus on the network layer due to the very nature of firewalls. However, we have a strong interest in the application layer, which is usually

closely related to the software running on a device. For this reason, we could situate our work on both the network layer and the software layer.

2.3.2 Attack examples

In their paper [26], Neshenko *et al.* also classify different types of attacks, vectorizing them between the exploited layer and the security impact. In their case, they define security in terms of the CIA triad but also in terms of accountability. They highlight three categories of attacks.

Attacks against confidentiality and authentication

It includes all attacks aimed at gaining unauthorized access to IoT resources or data. Brute forcing attacks are very popular in this category because IoT devices often use default credentials. This is essentially how Mirai and its variants conduct their attacks. They first scan networks for open ports, then try a set of default login and password combinations on vulnerable devices. This attack is fairly simple to execute and yet very effective because most users are unaware of these issues and do not change the factory settings. These dictionary attacks do not require any prior configuration, they can be easily performed, which is why they represent a significant threat. Spoofing attacks also fall into this category, as they involve stealing the identity of a device to act on its behalf and deceive the network. Spoofing can occur at different layers, but it is most often performed at the physical or network layer. This category also involves a range of physical attacks (i.e., requiring physical access to a device) such as evil twin attacks [27], node cloning or even node tampering [25].

Attacks against data integrity

This encompasses any injection of false data that could compromise the state of a device. It also includes firmware modification, but we will not focus on that. False data injection is a very interesting approach in our case. It consists of fusing legitimate or corrupted inputs to IoT devices in order to deceive them. A device can react unexpectedly and cause an economic impact. Even worse, it is possible to disrupt an entire network just by corrupting a small fraction of devices. An example of such an attack is shown by Zhou et al. [9] and illustrated in Figure 7, it exploits the interdependence feature of IoT devices. In this scenario, a thermometer is assigned the role of temperature controller and monitors the state of the air conditioner to adjust the temperature to the user's expectations. A self-controlled window is also able to open or close according to the temperature. Consider that the air conditioner and the window are well secured while the thermometer has some flaws. By exploiting these loopholes, it is possible to control the air conditioner, in addition to that, too much heat opens the window. This means that a corrupted thermometer could cause the window to open, even though it is well secured. This example shows how this interdependence feature must be treated with great care.

Attacks against availability

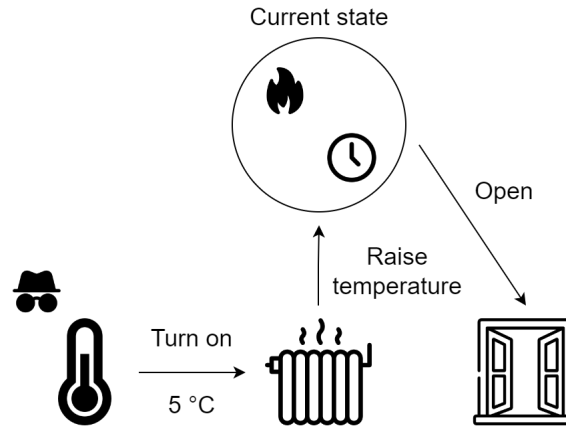


Figure 7: Interdependence scenario.

The main objective here is to induce a denial of service (DoS), in order to deny legitimate users access to their IoT resources. This category is very broad because there are countless ways to deplete a device’s resources. We can mention traditional DoS attacks, but IoT and in particular WSNs introduce a whole new set of flavors. These range from exhaustion attacks, such as sleep deprivation torture, to hole attacks or even routing attacks [25].

2.4 Countermeasures

As we just seen, vulnerabilities are not clustered at a single stage. They occur on different levels and this implies different solutions to be used to counter all of them. While physical shielding of a device safeguards it from being manipulated, it does not address the bugs in its software. It is not possible to deal with all issues in a single solution. Therefore, in our case, we concentrate on the networking side.

IoT devices are subject to various network attacks that endanger the well-being of customers. Fortunately, many pre-existing defensive measures can be applied in this area and the research community continues to work on establishing new defences better suited to the IoT. These solutions leverage existing technologies and do not reinvent the wheel. However, as we have demonstrated, new functionalities mean thinking a little outside the box and considering new possibilities.

2.4.1 Firewalls

Firewalls are one of the most effective lines of defence and have been very successful for many years. They consist of a software that acts as a barrier on a network, intercepting packet traffic. Today, they are available in several variants and can operate at different layers of the network stack. The main components present in any firewall are rules. A rule is a relationship between a packet signature and a decision. For network/transport layers firewalls, the packet signature typically consists of the source and destination IP, followed by a transport layer port. Upon receiving a packet, the firewall verifies whether the signature matches a rule, and

then applies the decision, which is either "accept" to admit the packet or "drop" to discard it. This type of firewall refers to packet filtering firewalls and does not keep track of any state.

While stateless firewalls are unable to identify two packets belonging to the same flow, stateful firewalls use efficient structures such as hash tables to map incoming packets to streams. This is a very useful feature for connection-based protocols such as TCP, as it allows a local network to access the Internet while protecting it. Thanks to connection tracking, stateful firewalls allow flows initiated from the internal network while blocking external connections. Nftables from the netfilter project [28] is an example of such a firewall.

At this point, we have only mentioned network layer firewalls, but modern solutions include other implementations that operate at a higher level. Application layer firewalls, such as web application firewalls [29], run at the last layer of the OSI model [1]. They typically act as a reverse proxy to protect a website from complex application layer attacks, such as cross-site scripting. They are commonly specific to HTTP/HTTPS protocol, but their scope can be applied to any application layer protocol. Lastly, there are next-generation firewalls. They exploit many sophisticated techniques, such as Deep Packet Inspection [30], which examines the entire contents of a packet, or sandboxing [31], which allows potentially harmful scripts to be run from an isolated environment.

Yet another characteristic of firewalls is the location of their deployment. Network-based firewalls act as gateway to other nodes in the same network. On the other hand, host-based firewalls are installed directly on a single host to protect it and only this host. A combination of the two is frequently used in practice. Multiple network-based firewalls can also be used simultaneously to create more sophisticated structures, such as demilitarized zones to separate public from private content.

Nftables

We have briefly recalled the concept of network firewall but it is interesting to dive into its most famous representative since we have used it in our system. Nftables is the successor of the well-known iptables [32]. In fact, Nftables is a more complete version of its precursor since it replaces and merges the popular ip, ip6, arp and eb tables into one command line tool. Nftables is very flexible and allows the user to define advanced rules. In addition, it offers many useful networking tools, including Network Address Translation (NAT), which allows multiple private IP addresses to be mapped to a single public IP address. As a side effect, NAT generally blocks attempted outbound connections, providing some sort of protection even though this is not its initial purpose.

In order to create a set of rules, the very first thing to define is a table. A table can be specified under any name and its purpose is to separate different rule sets. The most commonly defined table is the filter table, which, as the name suggests,

is used for packet filtering. To interact with the kernel, the tables use hooks which are links to the kernel's network stack, they catch packets on the physical interfaces. There are six types, but we will focus on only three of them: input, output and forward. The input hook is used to perform actions on packets directly addressed to the host running Nftables before NAT is applied. The output hook is the same with outgoing packets and finally the forward hook is for packets intended for another device. Inside the tables are the chains, which are sequential lists of rules. Each chain is attached to a hook, for example, a chain attached to the input hook will filter only packets destined for the host. Although the different chains may appear to be completely separated, this is not the case. In fact, packets can jump between multiple chains with the "jump" and "goto" statements. The jump is used to try to match in another chain and return to the original chain if there is no match, while goto does not go back.

The last and most interesting unit is the rule. Rules are what the signature of a packet is compared to in order to match inside a chain. A chain is traversed linearly and the first matching rule gives the verdict to accept or reject the packet. However, in our case, we do not make the decision at this level. Indeed, after matching, the packet is redirected to the user space where the decision is really taken according to the application layer content.

2.4.2 Intrusion detection systems

Intrusion detection systems (IDS) are another widely deployed security measure. IDSs are closely related to firewalls, but they act very differently. A first key difference is that IDSs are passive, as the name implies, they only detect anomalies and perform no further action. In contrast to intrusion prevention systems that detect and mitigate the intrusion.

Hung-Jen *et al.* [33] provide a comprehensive review of IDS. They describe three main detection methods, namely signature-based detection (SD), anomaly-based detection (AD) and stateful protocol analysis (SPA). A signature is a pattern that corresponds to a recorded threat, such as a combination of characters. In signature-based detection, each processed packet is compared to the threat signature database. If there is a match, the packet is considered malicious. This solution is straightforward to implement, but it is vulnerable to new attacks and expensive to maintain. In contrast, AD does not use a threat database but compares the packet to a number of profiles that represent expected behaviors. If the packet comes from irregular activity, it is considered malicious. In this case, new attacks are correctly identified, however, a legitimate use can also be considered an attack if it is unknown. SPA, also called specification-based detection, is a trade-off between the other two approaches. It relies on a set of properties to build a model used for detection [34]. Such a model can for example be expressed as a state machine. In this case, received packets correspond to transitions and states correspond to a valid situation or a violation. The latter method compensates for the high false negative rate of SD and the high false positive rate of AD.

Like firewalls, IDSes can exploit different sources of information, such as network traffic or host logs. In our case, we are limited to network information and we operate in a centralized manner. Although we tend to develop a firewall, our work also provides some intrusion detection capabilities and is relatively close to a specification-based IDS.

2.5 Finite state machine

Since we are using a finite state machine (FSM), it is necessary to give a clear definition of it and to explain how it works. This section gives an overview of the different types of FSM and their mechanisms.

2.5.1 Deterministic finite automata

According to Chen *et al.* [35], a finite state machine is a mathematical model of computation that takes inputs and responds to them by performing a set of actions, also called transitions. FSMs are widely used in many types of software, including network protocols, compilers, and event-driven systems, because they provide a succinct and expressive approach to explain a program logic.

Between the transitions we find nodes, each of which represents a state of the considered world. Thus, all the nodes of the FSM together form all possible occurrences of the system. An FSM can only be in one state at any given time and a transition occurs from one state to another when a certain condition is fulfilled. It is possible to completely represent a deterministic FSM by providing a tuple of five elements: the set of internal states, the initial state, the set of final states (also called accepting states), the input alphabet (a set of symbols) and the set of transition functions. The input alphabet can be viewed as a set of values that can be inserted into the FSM to perform transitions. From each state, there must be exactly one transition function for each input symbol in the alphabet. Deterministic finite automata (DFA) are often used to represent and control the execution flow of a computer system, but they can also be used to represent a simplified view of real-world state and interactions, as we do with the firewall.

2.5.2 Non-deterministic finite automata

Non-deterministic finite automata (NFA) are quite similar to deterministic automata, they take the same input tuple of five elements but they are less constrained. Not all states need to have a transition function for each symbol of the alphabet and moreover, they can have several transitions for one symbol, which makes them much less predictable. In addition, NFAs also have what we call "null" transitions, which are transitions that do not correspond to any input symbol.

It is interesting to note that DFAs are as powerful as NFAs. Indeed, it is still possible to represent a NFA by a DFA but the DFA will require many more states and transitions to match the same behavior. Figure 8 shows an example of DFA and

its conversion to NFA. As we can see, the DFA has three states which are composed of an initial state (left), an internal state and an accepting state (right). The input alphabet contains two symbols: "a" and "b". The arrows represent the transitions that are associated with an input symbol. Each state has exactly two transitions, one for each symbol because it is deterministic. On the other hand, the NFA has fewer transitions and even several transitions for the same symbol.

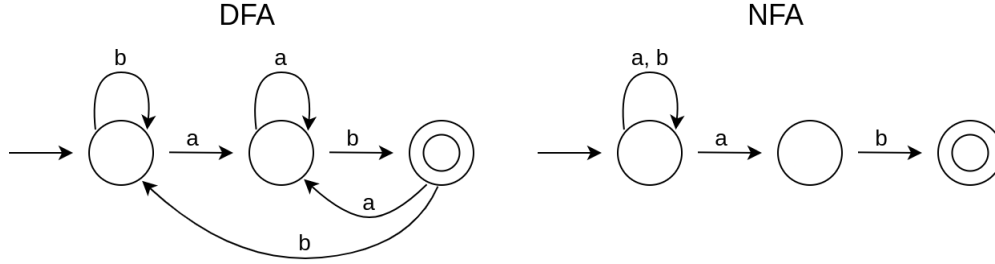


Figure 8: A DFA and its conversion to NFA.

2.5.3 Probabilistic finite automata

As described, given a specific state and an input symbol, the DFA can move along a transition to a single possible next state while the NFA has a set of possible destination states. In this case, no probability is associated with the transitions. However, it is possible to do so, as explained by Merve Nur Cakir *et al.* [36]. Given a current state, a probabilistic (or stochastic) finite automata (PFA) have a weighted set of next states where all weights are probabilities and must sum to one. Therefore, PFA can be considered an extension of NFA in the sense that an input symbol can lead to different states. However, PFA is closer to DFA because the probabilities of the transitions make it more predictable and therefore less non-deterministic. PFAs can also be considered as a generalization of the Markov chain, as it is a more powerful concept. We can represent PFAs by the same tuple of five elements as DFA and NFA but the initial state is replaced by a stochastic vector giving for each state its probability to be the initial state. A last element is necessary, it is the probability matrix giving the probability that a given state transition takes place. Figure 9 shows an example of PFA with the same alphabet as before, three states and six transitions. Note that the sum of the probabilities of all the transitions coming out of the same state is equal to one.

2.6 Related work

This section aims to summarize some of the most recent systems proposed to address security and privacy issues in the IoT. Some of them contributed greatly to the decision making process during our research.

ContextIoT is a context-based permission system for app-enabled IoT platforms proposed by Jia *et al.* [37]. It alleviates the over-privileging issues of IoT platforms

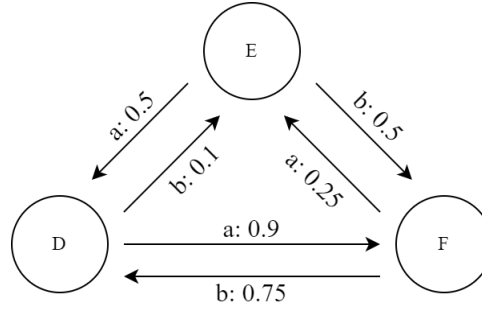


Figure 9: An example of PFA.

by providing the user with finer control over application behavior. It does this by emphasizing the preservation of "contextual integrity," that is, "the flow of information in accordance with contextual norms". They abstract the context of use of the application's functionality into program paths that represent execution flows at runtime. When an action is performed, the path is presented to the user as a sequence of steps, who can then decide to accept or reject the action. A cloud-based backend is used to remember previous decisions to limit the frequency of prompts. They implement a prototype for the Samsung SmartThings platform [38] and test the effectiveness of the system against several smartphone malware classes. While the proposed solution is effective, it tends to solve the problems introduced by the SmartThings framework and not necessarily the issues of any smart home. It also induces some latency.

HAWatcher proposed by Fu *et al.* [39] is another anomaly detection system for appified smart homes. Like ContextIoT, it is deployed on SmartThings. Their process begins with an extraction phase that uses a semantic-assisted extraction method exploiting various semantic information. After extracting, the next step is to generate hypothetical correlations between events within the smart network. For example, consider two events A and B , such that A means "the current is high" and B means "the switch is on". We can then create a correlation such that $A \implies B$, which means that if the current is high, the switch should be on. Subsequently, if we detect that the power is high and the switch is off, there is a mismatch and an anomaly can be detected. These correlations are then used to detect both device malfunctions and attacks. Correlations are extracted from three channels. On the smart application channel, because applications generate explicitly programmed correlations such as turning on lights if the user is detected. Second, on the physical channel, because devices can interact by changing physical properties, so a change in the physical environment caused by one device can induce an action on another device. Finally, the user activity channel, because the states of the devices reflect the activities of the user, and this activity can link different devices. For example, if the door opens, it means that the user is nearby and the motion sensor should also be triggered. Since SmartThings does not give access to internal states, they developed a shadow execution engine that keeps track of the state of each device and simulates the network based on the correlations. Their evaluation shows an excellent result with up to 97.83% accuracy and 94.12% recall. However,

they expose some limitations. Correlations related to user activity are subject to deviations that can cause false alarms and long-term correlations (with a large time interval) cannot be extracted. Furthermore, it is possible for an attacker who knows the correlations to build an attack that evades detection. Finally, sparsely deployed IoT devices with little interaction in the physical channel do not work well with the system.

PFirewall designed by Chi *et al.* [40] aims to increase the privacy of IoT platform users by providing a customizable data flow control system. Its main objective is to limit the data transferred to what is necessary for operation in order to comply with the "data minimization" principle of the European General Data Protection Regulation [41]. It does this by applying data minimization policies on observed traffic, which are either automatically generated or specified by the user. PFirewall addresses two challenges. First, it limits data transmission without harming the home automation system by using trigger-condition-action rules. These rules can be extracted through code analysis and natural language processing. Second, PFirewall performs filtering without applying any changes at the platform or device level, it only acts as an intermediary between the platform and the devices. They implemented the solution on SmartThings and openHAB and their results show that it can effectively reduce the data sent up to 97% without affecting home automation. One interesting point to mention is that they use a virtualization approach to handle encryption, PFirewall simulates a hub/cloud backend for the devices and also emulates the devices for the hub/cloud backend. However, PFirewall only provides privacy guarantees and filters the traffic generated by the devices, its purpose is not to provide security guarantees against an external adversary.

IOTsafe is a safety and security policy enforcement system, created by Ding *et al.* [42], that can collect and handle real-world physical interactions on smart home platforms while taking into account contextual factors. Physical interactions can be any interaction with an IoT device and a room, an object or another device. IOTsafe contains a runtime physical interaction discovery approach for identifying true physical interactions between IoT devices, which uses both static analysis and dynamic testing techniques to find runtime physical interactions across devices. IOTsafe also creates physical models for temporal physical interactions, allowing it to predict upcoming harmful scenarios and prohibit unsafe device states accordingly. In order to enforce the safety and security policies, they try to capture real physical interactions by using temporal and spatial context, implicit effects (such as increase of temperature causes increase of humidity) and join effect (multiple devices working together, like thermometer and heater). The authors consider an attacker that cannot have access to sensitive applications and devices but can compromise vulnerable applications and devices, the user safety and security policies are considered as trustworthy. Moreover, the prototype of IOTsafe is implemented on the smartThings platform. In the evaluation of the prototype, they are able to prevent risky temporal interactions with an accuracy of 96%.

These works allowed us to build the foundations of our research and we exploited

some of the concepts they introduced. In particular, we based our security objectives on the notion of contextual integrity highlighted by ContextIoT. The notion of channel from HAWatcher allowed us to identify the different modes of interaction between IoT devices. Moreover, we reused a prototype structure similar to that of PFirewall, with the idea of a centralized firewall filtering the traffic. Most of the proposed solutions rely on machine learning (ML) to analyze network traffic, devices behaviors and other data related to smart homes. We chose not to use ML because it requires heavy data sets that we did not have time to produce. The majority of them also run on proprietary IoT platforms such as Samsung SmartThings. We chose not to depend on these platforms but to provide a network protocol compliant solution, i.e. a solution that works with a range of network protocols and can be implemented independently of IoT platforms. We know that these works are of very good quality, our goal was not to compete with them but to propose a different approach.

3 Design and implementation

At this point, we have presented the framework we are in, the world of IoT and specifically smart homes. These environments suffer from many security issues. This is where our research lies, we strive to address these issues and provide an innovative smart firewall system to counter them. In this section, we dive into the design choices and implementation of this firewall. We start with some theoretical aspects, which are essential to define our goals. Next, a high-level overview of the architecture, the different components of the system and the terminology we use. Then, we delve a little deeper into the more sophisticated entities of the implementation, the protocol decoder and the context module. Next, we describe the parser and handler entities, specifically how the configuration file is entered and parsed. Finally, we will reconstruct the entire workflow, giving a sequential view of the system.

3.1 Theoretical considerations

Computer security is vast, and it is essential to narrow the field, which is the intent of this section. First, we define the specific security goals we want to achieve. Next, we provide the assumptions we made in order to target a specific problem. Finally, we describe the threat model and how we position an adversary against our system.

3.1.1 Security goals

Traditionally, security objectives are defined based on the CIA triad, which also applies to the IoT. Nevertheless, the IoT introduces a plethora of new threats that require more abstract definitions of security. Contextual integrity, as defined by Nissenbaum [43] is such a definition. It is based on three principles, and more specifically on the third one "Curtailling Intrusions into Spaces or Spheres Deemed Private or Personal" which represents the goal we are trying to achieve. Our work tends to respect this principle in its own way. In particular, it strives to ensure behavioral and network state integrity through the application of policies.

Our firewall gives the user the ability to define the network components, their state, and how they interact with each other. As mentioned, these interactions extend the attack surface of smart homes. Devices are expected to interact in a specific way, as defined by the user, but an adversary could attempt to deviate from these predefined relationships. This is where we introduce behavioral integrity, we strive to ensure that devices behave as they should. This is done by enforcing a set of policies ensuring that only authorized interactions can occur. They are dynamically enabled and disabled based on the actual network state and the firewall makes inferences about this state. Thus, inconsistent attempts that go against the firewall's expectations are constantly blocked. Even if a trusted device is corrupted, its capabilities will be strictly limited to what the user has allowed it to do.

The user can also define different states for the devices, such as the current state of a sensor, for example an open or closed window or the temperature of a

thermometer. All these different states form the state of the network. It is essential to maintain the consistency of this state to avoid malicious actions, this is what we call network state integrity. Some of these states are inferred internally in the firewall and therefore cannot be corrupted unless the firewall is compromised. However, some data is extracted from the packet exchanges to update the states. In order to control these exchanges, strict constraints can be tested against the observed data, comparing with ranges of values for numbers and exact values for strings of characters. Thus, the thermometer can be prevented from publishing absurd values such as 1000 degrees or the shutter from being in a "semi-open" state.

Besides, we also aim to ensure protocol behavior integrity. This means that we strive to ensure compliance with application layer protocol standards we support. This is done through a rigorous comparison between the actual packet exchanges and the expected protocol behavior. Thus, any abnormal behavior is detected and rejected. Confidentiality and availability are outside the scope of our research, so we do not provide them. We really wished to focus on the smart side of the firewall, relaxing some of the security requirements to make it more feasible. However, we are aware that this is a bad practice in real-world systems, especially omitting confidentiality even though some real applications do not always provide it.

3.1.2 Assumptions

Our solution does not protect against all possible threats and is naturally based on a set of assumptions that we tried to reduce as much as possible. We made them mostly because the problem they address is out of reach and/or too costly to tackle at the moment, as we focused on the innovative aspects of the firewall.

The biggest assumption we had to make is certainly to omit encryption. From a security perspective, this is a very unwise decision, but since this is only a proof of concept, it is an acceptable assumption. The main reason for this choice is that an intermediate node (such as our firewall) between a source and a destination, is not able to understand encrypted traffic. However, our whole concept is based on the fact that we can understand the exchanged messages. A second reason for this choice is that this problem is not new and encryption proxies already exist, there was no real interest in recreating this type of technology. A third reason is that solving this problem often involves machine learning, which is used in most of the related work we have read. Indeed, instead of decrypting packets, it may be more efficient to have an algorithm learn their signatures as discussed in 5.1.1. In this way, the system can act based on the observed signatures. We found it more interesting to use a different technique to analyze the traffic, but the approach is also more feasible. The final reason is that IoT devices are sometimes limited and cannot perform encryption, but this is most often not the case in smart home installations. In these setups, devices have access to a constant power supply as emphasized by Stojkoska *et al.* [44] and are only limited by their computing capacity. Encryption is discussed in Section 5.1.1.

A second important assumption is that devices do not authenticate themselves. This means that the firewall is not able to determine whether a device is really what it claims to be. Omitting authentication is also a very unsafe practice, but we made this choice because authentication often depends on encryption, as discussed in Section 5.1.2.

Another assumption we made is that the firewall is able to intercept all traffic sent between any device. This assumption cannot be removed in our design since the firewall must observe all exchanges to update the network state. It also implies that the devices cannot interact directly with each other, but can only talk to the firewall. This is realistic in practice, as devices typically share a wireless connection with an access point that can filter traffic.

The devices must work with the protocols we support at each layer of the OSI model. This primarily involves the use of standard IPv4 or IPv6 protocols at the network layer, TCP or UDP at the transport layer and any protocol for which we have implemented support at the application layer. The main drawback is that our prototype does not work with lightweight protocols such as IEEE 802.15.4 and 6LoWPAN. On the application layer, we have currently added support for MQTT and CoAP but our implementation is extensible to other protocols. It is important to note that our work is not intended to work with existing IoT platforms such as Samsung SmartThings [38]. Instead, it is a standalone solution that is intended to work only with compliant devices, as discussed in Section 3.2.1.

3.1.3 Threat model

A threat model is used to list and identify potential threats, but also to define the most likely attack vectors on a system. To define ours, we separate the attack surface into two planes, the internal network and the external network. The internal network refers to everything that belongs to the smart home environment, such as the router or smart devices. The external network is the opposite, it is everything outside the internal network. It is most often symbolized by the Internet, but trusted devices beyond the internal network also belong to the external network. We consider an adversary acting in each plane and insist on this separation because of the nature of the IoT. Indeed, internal network communications are primarily wireless, which leaves the door open to many physical attacks that a firewall is ineffective against. These include attacks such as the evil twin attack [27], which consists of installing a fake WiFi access point in the hope of tricking devices into connecting to it instead of the legitimate access point. On the other side, firewalls shine when it comes to handling external attacks. This is where we are focusing our attention. Note also that attacks are not strictly clustered within these two planes and can span both. Figure 10 represents the threat model, we have labeled three attack positions that are interesting to consider. The separation between the external and internal network is characterized by a router, an access point or anything that can connect a local network to the outside world. This router often acts as a "natural" barrier due to its NAT capability, but this is not always the case.

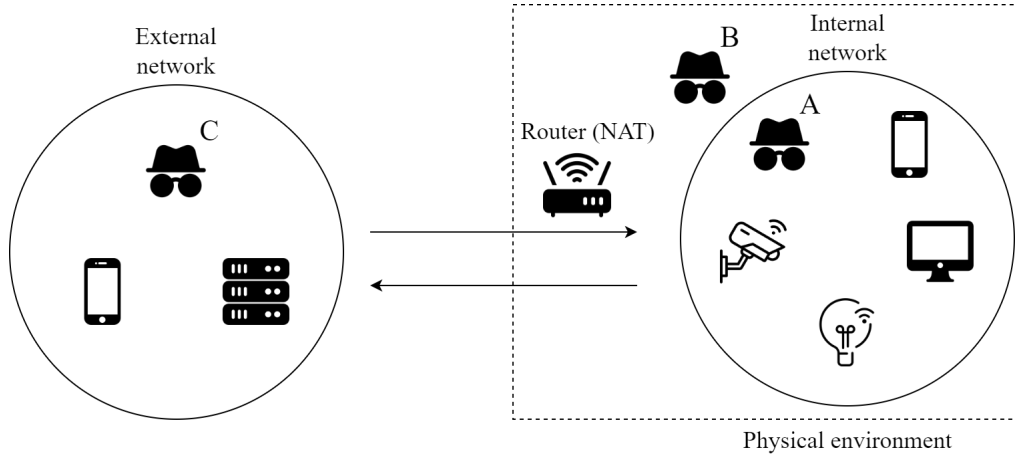


Figure 10: Threat model.

The internal network is the smart home environment. It is the set of trusted devices that interact with each other within the home’s local network. This is where we place our first type of adversary (A), which is a corrupted or malfunctioning device. With access to a trusted device, we consider scenarios where this type of attacker would attempt to violate contextual or network state integrity by performing false data injections. These are attacks where an expected protocol is used (e.g., MQTT), but with false information. For example, trying to turn off the security camera from the refrigerator. We also consider attacks that do not use the intended protocols, such as a dictionary attack from one device to access another device. Finally, we also consider faulty devices as adversaries. One can imagine a humidity sensor that constantly transmits low measurements when the humidity is high, which could quickly become annoying if that sensor triggers an automatic humidifier.

Looking at the diagram, we also notice another adversary outside the internal network (B). This corresponds to a malicious machine located outside the smart home but physically close to it, i.e. not part of the internal network at the origin. As opposed to a corrupted device that is part of the internal network. For example, an evil twin that tries to clone the trusted access point or a node that creates interference on wireless channels. This second adversary plays this role and has physical access to the local network without being part of it, it could be a neighbor for example. However, as we have already said, this type of attacks is not within the scope of a firewall and therefore we do not protect against them.

In general, internal devices have access to the Internet via a local router. The latter usually provide NAT, but this type of protection is not flawless and it is always possible to leave some open ports. For example, to access a cloud server acting as a hub. Or even if a user wants to access his internal hub from outside, he could connect to it directly without going through a remote server. Either way, we do not trust the external network and are envisaging attacks from this position (C). We are considering similar attacks as from the internal adversary, except this time no device is corrupted.

3.2 Architecture

In this section, we dive a little deeper into the architecture of the solution, without revealing the operating principle. First, we need to define the layout of the smart home and the structure needed to deploy the solution. Then, we give a high-level overview of the implementation components and their interactions. Finally, we define the terminology used to build the abstractions we employ. The intent is to provide the reader with a top-level grasp before delving into the implementation details.

3.2.1 Centralized firewall

There are two main approaches to firewalls: the distributed and the centralized approach. To secure a network, we can choose to route traffic through a single point and filter everything there, or we can have every node participate in the process, building the firewall in a distributed fashion. Ioannidis *et al.* [45] point out that a centralized firewall is best suited for small to medium-sized networks, which is the situation in a smart home. In addition, we were looking for a solution that is as self-contained as possible, as opposed to a distributed firewall that requires modifications to multiple devices. Instead, a centralized solution offers the possibility to leave the configuration burden on a single device, the firewall. The distributed approach is discussed in Section 5.2.3.

There are different categories of entities in our setup. The first category comprises sensors and actuators which we call IoT devices. These smart objects have limited computational capacity and, in general, serve only one purpose. We do not classify devices such as smartphones and desktops in this category because they have multiple uses and are more powerful. They form their own category of regular devices. However, these two categories are closely related because regular devices can interact with IoT devices. The last category includes all kinds of access points, such as provider routers, personal routers and the firewall itself.

Next, we need to define how these different network components are arranged. Fortunately, we are quite flexible on this point and many configurations are possible. Thus, we will only define the experimental configuration that we used during our research, as depicted in Figure 11. We can observe that we have chosen to segregate the regular home devices from the IoT devices. The latter are in a subnet that is accessed through the firewall. This decision was made for a simple reason: we did not want the firewall to interfere with non-IoT devices. Nevertheless, we leave it to the user to use the same access point for both types of devices, which reduces the number of access points to one (access point and router at the same time). This is not how the firewall was designed to be used, though. In this case, policies would have to be specified to allow, for example, a desktop computer to access websites.

On the smart object side, the firewall serves as a filtering bridge for end-to-end communication of IoT devices, access to the rest of the internal network and the Internet. This is where our firewall differs from the traditional centralized firewall:

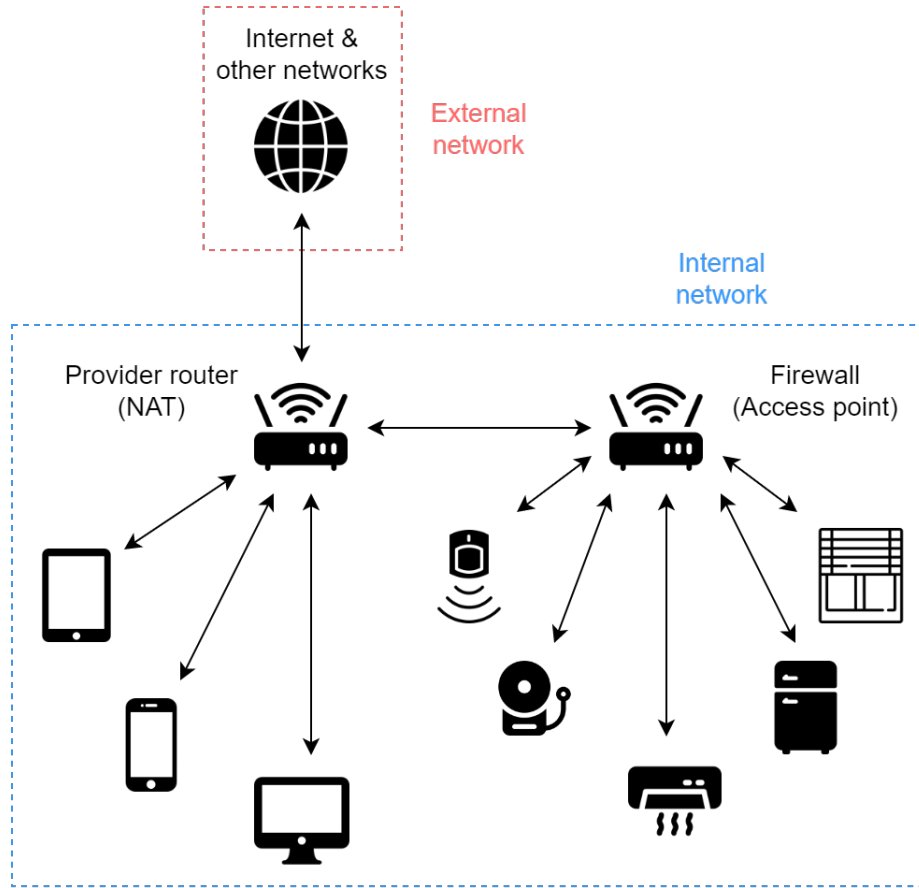


Figure 11: Network configuration.

it is not only the entry point to the subnet, but the intermediate point between all smart devices. This is realistic in practice, as the firewall is on an access point that is always the next hop for an exchange. We have pictured a provider router connecting both the subnet and the regular devices to the Internet. This is the point where the external and internal network are connected, and this is usually done through a NAT. Notice that the external network may contain known devices such as a hub/broker server, a smartphone or even a laptop trying to reach the smart home appliances.

Device compliance

One question that may arise now is "how do we make it all work together?". Because of the diversity of the IoT, this is one of the most challenging parts of the work. As mentioned in the related work and assumptions, we had to limit ourselves to certain protocols. This means that we assume that the devices use IPv4/IPv6, TCP/UDP, but also a supported application layer protocol. We also encourage the use of WiFi to connect devices to the firewall, but any wireless protocol that supports the above layer protocols will do.

One of our goals was to create an abstraction layer, merging different application

layer protocols into one abstract format. For this reason, devices must also adhere to a specific payload format, otherwise the firewall will not be able to comprehend it. This means that if a user has a smart Philips light bulb [46] that he wants to connect inside the network, it will not work. However, if the user has a configurable light bulb that can run an instance of CoAP (or any other supported protocol) and transmit standards-compliant payloads, it should work. We realize that this restriction is not really workable in practice due to the configuration burden. However, it was necessary for the purpose of our research. Also note that we have provided standard support for MQTT and CoAP but have not sought to support all the specific features of these protocols, we have limited ourselves to the most common uses (as in the evaluation) to provide support for at least two protocols. For example, we do not provide support for MQTT QoS [19] greater than zero.

3.2.2 Components

We can distinguish four main parts constituting the firewall. We present them in the following order: the *client* component, the *nft* component, the *nfqueue* component and the *context* component. First, the client is used to configure the firewall. Then, the nft module is responsible for the matching with the network/transport layer while nfqueue is dedicated to the application layer. Finally, the context keeps track and update the state of the system.

Client

The naming of this part comes from the original idea of building a command line interface (CLI) that interacts with a firewall daemon. Although this idea is attractive, it was not very practical to implement. For this reason, we decided to move on to configuration file parsing. We kept the name client because it is still the only point where the user is involved in the operation process, which in our sense is similar to a client interacting with a server. Thus, this part is responsible for the analysis of a configuration file provided by the user. This file contains the set of smart policies that govern the interactions in the smart home environment. After parsing, the entire configuration is given to a handler, which is responsible for applying the configuration. Again, this comes from the original idea of CLI interacting with the command handler to modify the running daemon.

Nft

The point of our firewall is to be smart, and for that we do not need to reinvent the wheel. Indeed, the very first step of a firewall is network/transport packet matching, which might seem like a trivial part of the implementation but is actually a huge burden. Initially, we designed our own matching mechanism, but we quickly realized that it was out of reach, and that it was much more interesting to rely on a complete and reliable solution such as Nftables. Besides being a firewall, it provides an efficient network/transport layer matching. Moreover, some libraries provide an interface to interact with Nftables directly from a script through JSON

commands instead of the usual CLI. However, the library we used is only responsible for sending the command and not building them. Therefore, we had to implement a small command constructor. This command constructor is then employed in an API exposing an interface to communicate with the Nftables daemon from other components.

Nftables plays its own role in the matching procedure but leaves the complex application layer handling to another component. This is made possible by a feature called "user space handling". This feature offers the possibility to queue packets to a user space process, which is responsible for delivering the final verdict. To do this, one must specify an "nfqueue" target as well as the index of the queue on the rules. This does the trick, but manipulating Nftables is still less efficient than a custom firewall-specific solution, as discussed in Section 5.2.2.

Nfqueue

This component implements the user space process delegated to the application layer matching, it is the largest and its name comes from the Nftable target "nfqueue". Its function is to ensure that the application payloads respect the restrictions imposed by the user's configuration. It maintains an active *handling queue* to receive packets, compare them to strict constraints before accepting or dropping them. These constraints are kept inside a *relation mapping*, which allows to save the constraints to be applied to the different possible interactions.

However, before comparing the payload of a packet to the constraints, we need to extract the contents of the packet and ensure that the packet is protocol compliant. This is done by what we call the *abstraction layer*. At the receiving end, packets are abstracted into a uniform format called *abstract packet*, useful for representing any type of protocol. At this stage, it is the mission of the *protocol decoder* to fill in this abstract packet correctly but also to understand its protocol. It automatically detects the application protocol and delegates the decoding work to a specialized decoder. The latter extracts the content from the application layer. In addition to decoding application layer content, the protocol decoder is also capable of understanding protocol behavior and all interactions between devices using a supported protocol. Since we are dealing with abstract packets in the handling queue, we would also like to ask high-level questions, to avoid being too protocol specific. These are questions such as "is this a pull packet?" or "is this a request?" to identify the behavior of the protocol. This is the task of the *packet state* component, which will process all packets to verify that they follow the principles of a supported protocol by asking the corresponding protocol decoder the right questions.

Context

Finally, we come to the heart of the firewall, the context. The context can be thought of as the representation of the set of states of all devices. The reader may think of this as the state of the network, but the context goes beyond that because device

states are often deeply related to physical conditions. This means that this state is not limited to internal device states, such as values in memory, but is also related to real world conditions, such as the current temperature. The context represents changes in this state by applying transitions based on observed data emitted by different interactions. These interactions are generated by the devices, but are also triggered by time. Keeping track of the current state of the network is a non-trivial task since the context will be populated by the user's specifications. If these are not complete or contain errors, then the state can quickly deviate from the real world situation. In addition, unexpected real-world events may trigger changes that are not reflected by the firewall.

We keep track of such a context in order to make the firewall *dynamic*. Indeed, in traditional firewalls, rules are *static*, meaning the user defines them and they never change (unless done manually). But in the IoT environment, many devices can communicate with each other and the interactions can have physical consequences in the real world. Thus, the context is constantly changing and it is interesting to adapt the ruleset. This is why we use what we call *triggers* which are functions that dynamically activate or deactivate interactions depending on the current state of the network. Furthermore, the context is not limited to the states of the devices and, for example, the time of the day can also trigger the activation/deactivation of interactions.

Take the example of someone who wants to prevent the heater from being turned on if the window is open. If the firewall sees a packet from the switch to open the window, then it knows that the heater should be turned off. As shown in Figure 12, the interaction between the heater switch and the heater is disabled until the window is closed. This is a simple example to explain the principle, but it is actually possible to express much more complex conditions for the triggering of these functions.

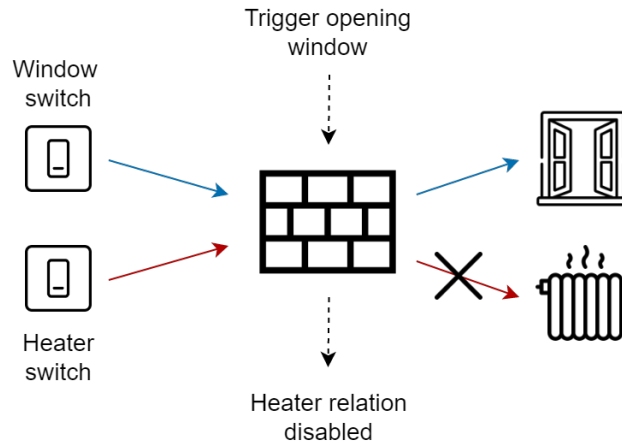


Figure 12: Example of trigger in the context.

3.2.3 Component interactions

Figure 13 gives an overview of the high-level interactions between components. As can be noticed, the entry point is the user input configuration. It establishes the initial set of policies and its syntax is explained in Section 3.5.1. Then, the packet flow drives the system forward and from one step to the next. The flow starts at the lower layers (network/transport) matching, a first filtering step. This matching sends a request for a verdict to the handling queue process. The latter informs the context of the changes observed inside the network, in an abstract way such that the context is able to make a transition to the next state. This transition can trigger the modification of the current policies. The whole process repeats itself as long as the firewall is active on the host. As we can see, our firewall goes far beyond traditional firewalls in many ways. First, in addition to working on the network layer, it works on the application layer at the same time. Moreover, permitted interactions are defined by relationships that allow strict constraints to be applied to packet payloads. Finally, the firewall is dynamic and is able to adapt in real time to state changes based on packet information.

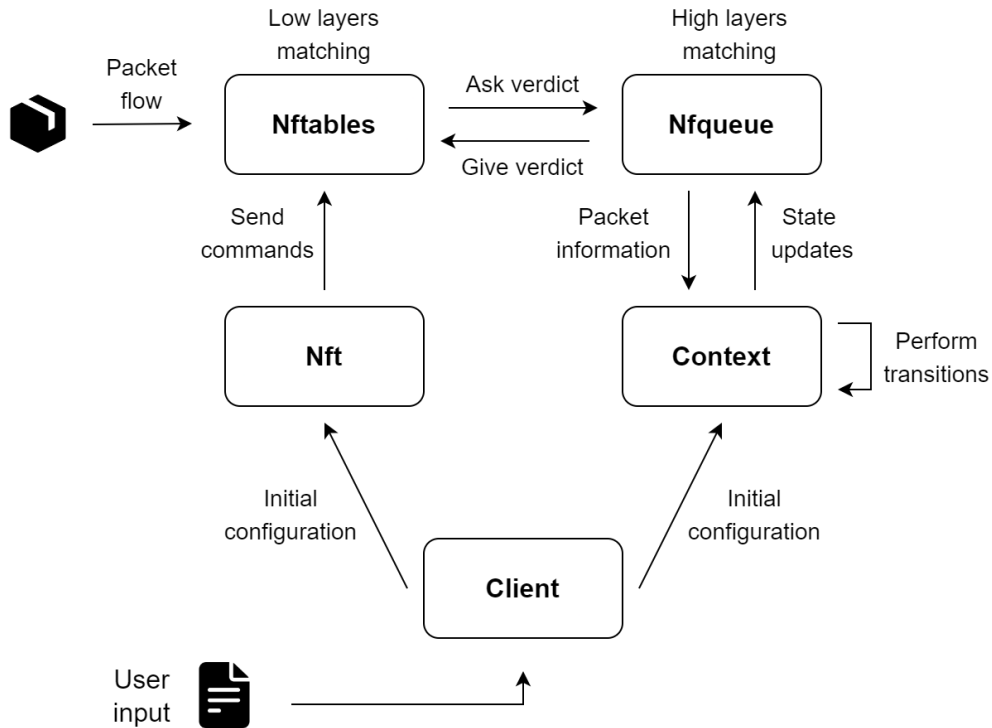


Figure 13: Component interactions.

3.2.4 Terminology

This section provides the definition of the notations we used in the design of the prototype to construct different abstractions.

Members

We have seen that IoT devices are based on different protocol architectures. However, to have a unified abstraction layer, we had to represent different types of devices independently of their protocol. A publisher is an entity that sends data about a specific subject, as opposed to the subscriber who receives that data. A broker is an intermediate node between a publisher and a subscriber. These three entities form the concept of *member*. In fact, it goes even further because a member also corresponds to a specific network/transport layer signature. This essentially means an IP or range of IPs with a port or range of ports like "192.168.1.1/24:1883". This notation is very convenient because it allows complex interaction patterns to be configured efficiently. According to this definition, the set of members consists of three subsets, but these sets may overlap each other. This means that it is not excluded that a member fulfills different roles, such as publisher and subscriber at the same time.

Fields and constraints

We have briefly mentioned the concept of network state, the fields are used to represent this state. They are data attributes that can be interpreted in string or decimal format. They are attached to a member, and more precisely to a publisher. A publisher who publishes a value for a field is considered the owner of that field. Fortunately, fields are not uniquely identified, e.g. a status field can be defined on each device. This gives the user the possibility to create very complex schemes with a wide range of fields on different devices. Moreover, we allow the user to define field *constraints*. These constraints are evaluated against the fields when the packet is intercepted. If the values are allowed, then the packet can pass, otherwise not. This ensures that only specific values are allowed and avoids incorrect scenarios. Again, these parameters can be set by the user and give him full control over the network traffic.

Rules and relations

We interact with the Nftables daemon in the implementation and to do so, we have exploited the concept of *rules*. In the prototype, the rules are similar to Nftables rules. In fact, they have a direct correspondence with them except that they do not bind two signatures but two members. The rules are used to create a larger structure, the relations. A *relation* is an entity representing an interaction between at least two members, a publisher, a subscriber and possibly a broker. They are defined generically, i.e., they are not tied to a specific application layer protocol. Depending on the presence of the broker, a relation is bound to two or four rules, one for each possible exchange path. Without a broker, this means publisher to subscriber and vice versa. A broker can be added between the two, resulting in four rules: bidirectional between the publisher and the broker, and the same between the broker and the subscriber. These rules are associated with a mark, useful for identifying and mapping them with relations. Relations are uniquely identified by a name and are also linked to a subject. One or more constraints can be applied to the fields of the publisher linked to the relation.

Figure 14 gives an overview of the way everything works together. Three members are defined with and without fields, and then two relations are established. The first one does not involve a broker, so only two rules (in orange) need to be initialized. On the other hand, the second relation considers B as the broker. Therefore, four rules are defined this time. This relation also defines two constraints (in red), one on each publisher field.

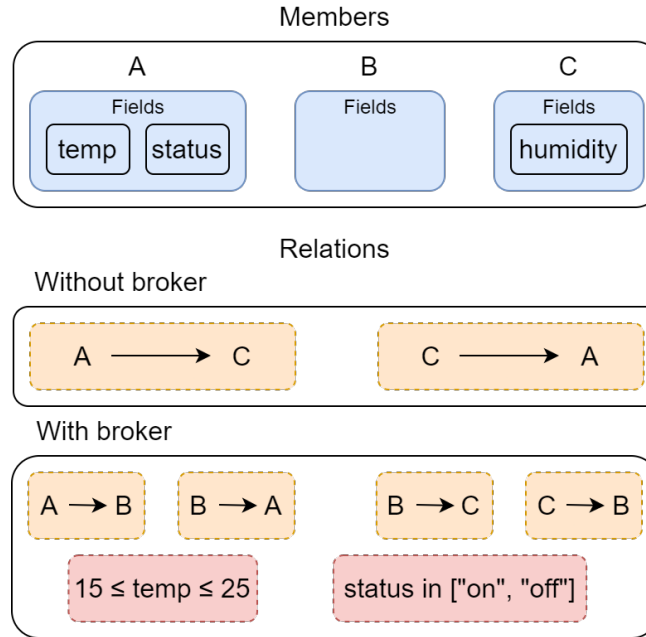


Figure 14: Example of relations.

3.3 Protocol decoder

The protocol decoder sets the system apart from more traditional firewalls and allows it to adapt to various circumstances. Moreover, it enables the firewall to support the heterogeneous world of IoT protocols and helps to enhance network security by forcing all packets to comply with their corresponding protocol.

To give a quick overview of how the protocol decoder works, it is important to start with the handling queue where all packets arrive. At some point, all layers up to the transport layer have been decoded and only the application layer remains non-decoded. The protocol decoder has two roles: to decode the application layer and to help determine the protocol behavior of the packet. In fact, it is the job of the `nfqueue.abstract_paquet.parse_application` function of the abstract packet object to get the contents of this layer. Then, the `nfqueue.paquet_state.handle_packet` function of the packet state uses this freshly decoded data to decode the behavior. In other words, the packet state class is responsible for understanding what type of abstract packet it is and whether it conforms to a protocol to decide whether the packet should be dropped or not. The abstract packet and the packet

state form what we have called the abstraction layer, an abstract way of representing both the contents of a packet and the behavior of its protocol.

3.3.1 Content abstraction

The application layer of the packet is decoded via the abstract packet class. This class calls the main decoding function of the protocol decoder class to obtain a tuple of four elements for each packet: the protocol, the header, the subject and the content of the payload. In the protocol decoder class, the decoder first automatically detects the protocol used by the packet in order to call the correct specialized decoder. The source and destination port numbers are considered to be reliable enough to correctly detect the protocol. It is important to note that some protocols may use different ports, for example, MQTT uses ports 1883 and 8883. A second remark is the fact that polymorphism is used, all specialized decoders implement the same function `nfqueue.any_protocol_decoder.decode`, this allows us not to specify any protocol in the protocol decoder class itself. Each specialized decoder will group and organize its decoded data in a well organized tuple object common to all decoders. In the next step, the application layer payload is decoded. This part of the decoding is protocol independent, we use a precompiled regex to extract the information from the payload. What we get at the end is a (field, value) tuple list where the field is optional. If the field for a value is not specified in the payload, the subject of the packet is used as the field for that value. The regex that is used is hardcoded at the moment but this can easily be modified.

3.3.2 Protocol behavior abstraction

At this point we have successfully decoded the content of the application layer, we have a complete abstract packet that contains all the information needed for the rest of the operations. Now an important question arises: what should we do with this abstract packet, how can we use it to make a decision about the concrete packet? We have a problem since we are dealing with abstract data, it is difficult to know if the packet is legitimate. We need to use low-level information specific to each protocol, but the latter differ in the way they exchange messages. For example, some protocols like MQTT have the subject and payload in the same publish message while in the CoAP observer, only the registration contains the subject (the uri-path) and the notification contains the payload. Some protocols are push (MQTT) while others are pull (HTTP) or even hybrid (CoAP). Some push protocols use a broker but others directly establish an end-to-end connection between the publisher and the subscriber (CoAP Observer). In order to handle such a variety, we abstracted the behavior of all these protocols and implemented functions in the specialized decoders to handle their own behavior.

Behavior in packet state

The initial step in behavior management is to capture the possible behaviors of all protocols in a class that must be abstracted to cover a wide range of protocols. This

is done in the packet state class which is in charge of abstracting the behavior of all protocols, this class also returns for each packet a verdict which provides three pieces of information. The first indicates whether the packet should be allowed, this depends on whether the packet appears to be protocol compliant, e.g. if a broker starts publishing messages with a subject to which the destination devices are not subscribed, then the packet should be blocked. The second and third information tells the handling queue whether the contents of the packet should try to match the constraints and finally whether it should be sent to the context. For example, it is not useful to match constraints with signaling packets or to update the context with these packets since they do not carry data.

In the packet state class, all packets are treated differently depending on the architecture to which their protocol belongs (pull or push). In fact, all requests for pull protocols are stored in a dictionary until their responses arrive back, and similarly for push protocols, all subscription messages are stored until an unsubscribe message or other type of stop event occurs. In this way, when a reply or publish message arrives, it can be mapped to a previously made request or subscription and thus ensure that it is a compliant, non-malicious packet. But storing these packets can be problematic in the event of high network load or a Denial of Service attack, as discussed in Section 5.1.3.

In order to remain abstract and never refer to a specific protocol, this class uses a function called `nfqueue.protocol_decoder.ask_protocol` which is able to retrieve information from the corresponding protocol decoder of the abstracted packet. This means that the packet state module is constantly communicating with the protocol decoder class. As explained earlier, the latter class is also responsible for decoding the application layer content, but it can actually do more. Indeed, it is capable of recovering any information that any specialized protocol decoder can recover. The motivated reader who has gone through this class will notice that the method `nfqueue.packet_state.handle_packet` contains only calls to this `ask_protocol` function, so this part of the implementation is not aware of the content or protocol of the packet.

Behavior in protocol decoder

As with content decoding, the protocol decoder class is responsible for delegating the work to the corresponding protocol decoder using its `ask_protocol` function. This class also does not know the names of the specialized decoder functions. This section is not intended to be an exhaustive list of these functions. However, it is always useful to mention and explain a simple example to give an idea of what it looks like.

Two interesting functions are the `any_protocol_decoder.is_push_packet` and the `any_protocol_decoder.is_pull_packet` which tell the packet state whether the packet belongs to a push or pull communication. As developed in the subsection 2.2.1, all protocols use either pull, push or even a hybrid approach. This structure

of a specialized question-answering decoder capable of dynamically adapting to protocol changes is therefore very suitable in this case. For example, as far as MQTT is concerned, its response to the `is_push_packet` function will always be true whereas for the `is_pull_packet` function it will always be false. For CoAP, however, it is not as trivial as for MQTT. A CoAP client can become an observer by using the observe option in the header. Therefore, it is clear that the firewall will look for this option in a CoAP packet to determine if it is a push packet or not. These are just two of the many functions implemented in decoders, which use all the subtleties of each protocol to recover any information.

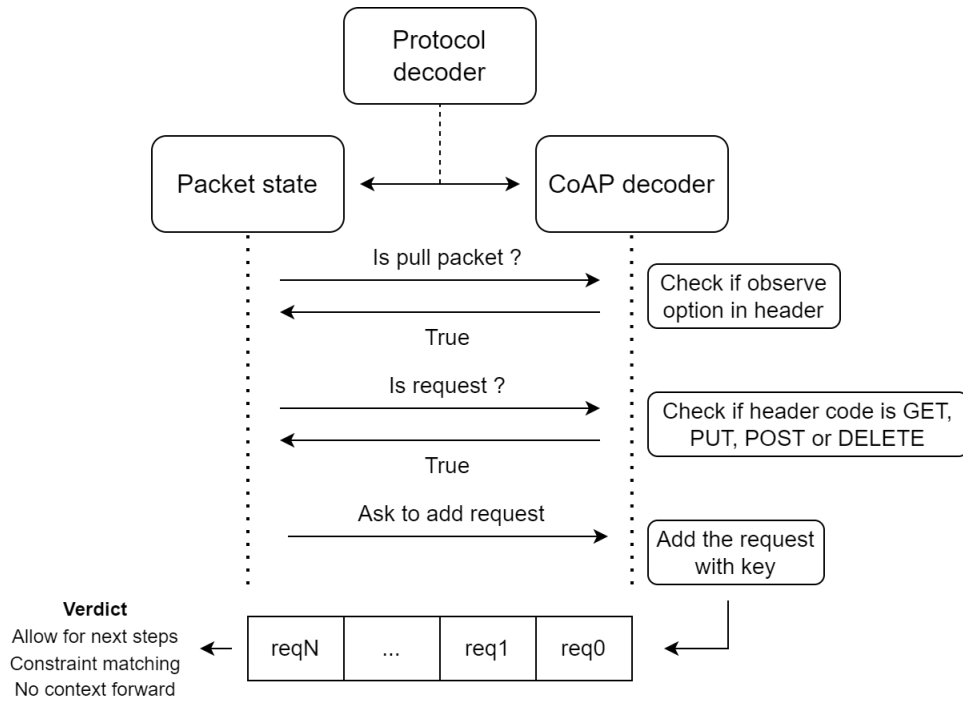


Figure 15: An example of execution of the protocol decoder processing a CoAP request.

Because a picture is worth a thousand words, Figure 15 shows an example of the protocol decoder processing a CoAP GET request. In this example, we can see that the packet state sequentially asks the CoAP decoder the questions through the protocol decoder. First it asks if it is a pull packet and the decoder answers by checking the observe option in the header. Then it checks if it is a request by examining the header code. Finally, it tells the decoder to add the request to the queue of pending requests used to initiate the verdict for the next steps. In this case, the packet is authorized, must try to match the constraints, and should not be sent to the context because it is a request and not a response.

By taking a closer look at the MQTT and CoAP decoder, we can see that they modify the packet state data structures by calling its functions. Specifically, they modify the request and subscription dictionaries under certain conditions specific to their protocol. For example, upon receiving a subscription, the MQTT decoder

adds a subscription only if the devices have successfully completed the connection exchange with the broker. Furthermore, in order to add an item to these dictionaries, a key is required. This key is a tuple composed of the source and destination IP of the relation, the protocol used between these devices and a last item that changes depending on the protocol decoder. For example, in MQTT, the last item of the key is the subject of the subscription and in CoAP, it is the message identification field of the CoAP header. Having a key that uniquely identifies the packets in a relation allows us to quickly find the subscription of a publish message or the response request.

To conclude this section, it seems important to summarize the mechanism of the protocol decoder. Figure 16 shows how this module essentially works and the interactions between its components. As we can see, the protocol decoder helps both the abstract packet class to decode the packet contents but also the packet state class to understand the protocol behavior. All information is retrieved through the specialization of the protocol decoder. This part of the code is extensible, and we have built it so that new decoders can easily be added to support a wider variety of standards.

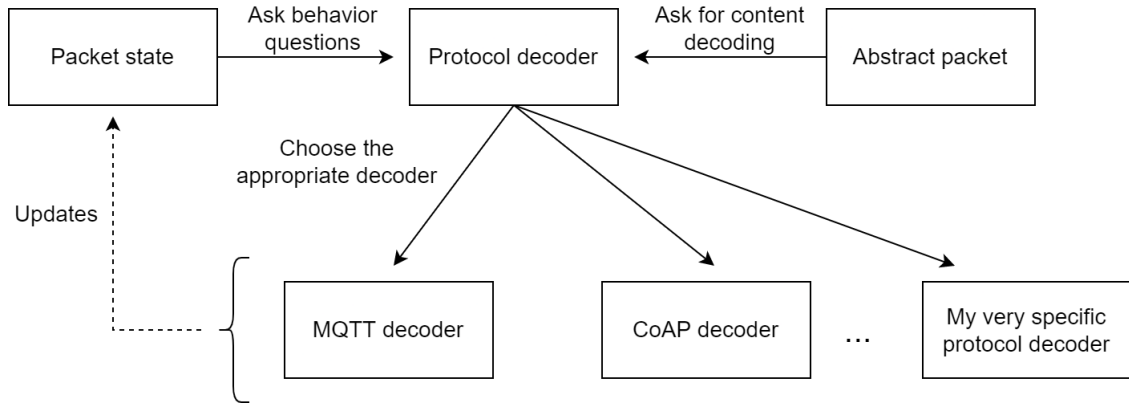


Figure 16: Overview of the protocol decoder module.

3.4 Context

This section develops and analyzes the context implementation, how it allows the firewall to adapt and react to real-world events by observing the packets exchanged by IoT devices. In this module, we maintain the complete state of the network as we want to disable or enable relations dynamically based on the packets passing through the firewall. Unlike other filtering entities such as the protocol decoder, this module does not explicitly drop or allow packets. Instead, the context modifies the relation mapping dictionary by setting the relation, and in particular the subject, of the corresponding mark as *enabled* or *disabled*. As a reminder, the mark identifies an interaction between two members, but several interactions can be established between these two members, so the subject allows to differentiate them. This process

of dynamically changing the status of a relation when the context is refreshed is the key feature of this module and the following subsections describe how it is achieved.

3.4.1 Context module

Some packets that have passed all filtering steps (IP filtering, protocol compliance and constraint matching) are sent to the context. In fact, these packets are placed in a queue that is shared between the handling queue and the context module. The context waits for abstract packets with a blocking call in a separate thread, which means that if many packets are waiting to be processed, it does not affect the handling queue. Once a packet is out of the queue, each tuple (field, value) in its payload is considered one by one and undergoes some operations before being sent to the real context data structure.

The main operation on the tuples is called *categorization* and is only necessary if the value of a field is a number. In this case, the value is labeled from a float category to a string category through a user-supplied mapping. The mapping is stored as a dictionary whose keys are the names of the members concatenated to the fields. When we travel the (field, value) tuple list, we can retrieve the member name with packet information and concatenate this string with the field to build the key. An illustrative example of mapping is the temperature of a thermometer. If the context receives a tuple (`temp`, 27) and the user labeled the values below 10 as cold, between 10 and 20 as average and above 20 as hot, then the value is classified as hot. The first array represents the range and the second the corresponding labels in order. The usefulness of such categorization is discussed in Section 5.3.1.

3.4.2 Triggers and network modules

At this point, we have converted the numbers to strings. The firewall now processes tuples (field, value) where both are strings. In order to process this input and update the context, two modules are needed: the *triggers* and the *network*.

Triggers module

Triggers can be considered as the list of high-level rules that define under which condition one or more actions should be executed. This list of triggers is part of the user input specification, the user must specify the condition with its type and the resulting action. A condition can be of type field or time, the former specifies only what the current value of a list of fields in the context should be and the latter is a time interval that indicates at what time of the day an action should be launched and at what time it should be cancelled. For example, the user can specify the deactivation action on the "trigger_window" relation between 8pm and 9pm. If, on the other hand, a field is used as a condition, then the user must specify a value on this field to perform the action. For example, if the temperature is "hot", the relation "trigger_window" can be activated.

The main purpose of the triggers module is therefore the management of high-level rules, it keeps these rules in memory and applies them to the network data structure. This data structure has not yet been revealed and explained because the implementation has been made in such a way that it is possible to easily change it for another one. For example, if the data structure later turns out to be no longer suitable and a better solution has been found. Looking at the triggers class, we can observe that in the functions `context.triggers.add_trigger` and `context.triggers.del_trigger`, triggers are first added/removed in the local dictionary, and then a call to the `context.network.add_triggers` and `context.network.del_triggers` methods of the network is made. The network is thus considered as a switchable black box that must implement these two functions.

Network module

We will now dive into the main course, the network itself which is responsible for managing the context FSM. The data structure we have chosen is a DFA because it is well adapted to our use case. The first main advantage is its simplicity and the fact that the correspondence with the real world is quite direct. A state in the FSM represents the current state of the network and when an event occurs (in our case, a packet field value passed to the context), a transition happens from a source state to a destination state. The second advantage is the performance of such a data structure: given an input, the correct transition is traversed almost instantaneously and any trigger attached to that transition is executed. One last benefit of FSMs is that they are flexible [47], meaning its easy to modify them and add new features.

The current state of the FSM is represented as a dictionary of fields, which are set to a possible value. These values are specified in the *state combinations* which defines for each field what are the possible values, it is reconstructed from the input file. When a tuple (field, value) of a packet updates the context, the value of one or more fields can be affected and a transition occurs. Indeed, two states linked by a transition can have more than one different field thanks to the *state inference*. It is a dictionary that indicates the inferences that can be made from certain observed field values. For example, if the firewall receives a packet from the thermometer indicating that the temperature is high, it could guess that the heater is going to be turned off, if the user has specified it. Thus, the destination state has two fields with a different value, the temperature becomes high and the heater is now off. While it is not mandatory to specify state inference, its use can really help to maintain consistency of the network context in its mapping to real-world device states.

3.4.3 FSM building

The first step in explaining the network was to describe how the context is stored and updated, but such a data structure must also be built and maintained. First, two inputs are needed to build the FSM, the first is the combination of states that has already been explained and the second is the initial state. When the firewall

is restarted, it is also possible to save the current state and reuse it to rebuild the data structure. With these inputs, the first step is to build the list containing all states, considering all possible combinations of values for all fields. Once all the states are created, the transitions are generated by considering the transitions from all states to all others. The default is that only states with a single difference field are connected, but because of the state inference discussed earlier, it is possible for two connected states to have multiple differences. A second thing done during the construction is to define as inconsistent all states that meet one of the conditions of the input *inconsistent state*, provided by the user. It is also important to make sure that we do not bind a state to itself, thus creating a self-loop. Self-loops are actually implemented via a hook function that detects them before attempting to make a transition and a custom exception is thrown, preventing the rest of the code from being executed. This small optimization is useful because self-loops can happen often if, for example, some published package is retransmitted several times.

The attentive reader will have noticed that the number of states contained in the FSM is exponential because all combinations are considered. Indeed, each member can be in a certain number of states depending on its number of fields and their number of values. For a single member, this is calculated by computing the product of the number of values for each field. For example, a member with one field of two values and one field of three values will have a total of $2 * 3 = 6$ states. In order to get an FSM state, we need to take into account all the members. Thus, we perform the last calculation for each member and multiply the results between them as shown by the following formula:

$$\prod_{i=1}^M \prod_{j=1}^{F_i} v_{ij},$$

where M is the number of members, F_i is the number of fields for the i th member and v_{ij} is the number of values for the j th field of the i th member. This problem of exponential number of states in finite state machines is well-known and is discussed in section 5.3.1. Now the finite automaton is missing one last element: the triggers. To manipulate the triggers in the data structure, the function `network.add_triggers` is necessary. This function iterates over all the transitions, ignoring the one that starts from an inconsistent state or that leads to it. For each transition, it checks if a field in the condition of any trigger is modified, if it is then it means that this trigger must be attached. Now remember that a transition has a source state and a destination state. If the source state does not fulfill the trigger condition but the destination state does, then it simply attaches the trigger action (enable or disable relation) to the transition. On the contrary, if the source state fulfills the condition but not the destination state, it attaches the opposite action to the transition. This means that enable becomes disable and vice versa.

3.4.4 State inconsistency

The state inconsistency entry is a list of conditions defining the set of inconsistencies. For example, if the temperature is high and the heater is on, this can be defined as an inconsistency. The goal of the contextual FSM is to avoid entering these states, and the triggers attached to the transitions should prevent this, because in specific states they disable relations that could lead to an inconsistent state. However, if the firewall fails to block the inconsistency, a hook function is triggered on entering and exiting the state to raise a stealth alarm. The occurrence of such an inconsistent state is expected to be mainly due to a misconfiguration by the user. It can also be caused by an unexpected order of packets or packets close in time, this topic is discussed in Section 5.3.2.

3.4.5 Path likelihood

As mentioned earlier, FSMs are adaptable because it is simple to add new functionality to them. In this subsection, we add a counter to each state and transition that is incremented each time they are visited. In doing so, this allows us to obtain the probability of a transition being applied by simply dividing the counter of a transition by the counter of its source state. In this scenario, an important invariant must be respected: the sum of the probabilities of all outgoing transitions of a state must be equal to one. Then, we introduce the concept of *infrequent state* which is a state that has been traversed less than n times. This type of state will not be considered at runtime until it becomes frequent. Finally, if we consider a sequence of k following transitions as a path, then the *likelihood* of the path is the probability that this path can be taken.

The purpose of using probabilities is to be able to trigger an alarm when a transition with a too low probability is triggered because it is unusual. However, these events may appear from time to time without any malicious behavior causing them. This is why we consider paths, the idea being that if several surprising events occur close together in time, it may mean that something really unusual is happening. This path probability is recalculated each time a new packet is introduced into the network context. The value of k is important because if it is too low, the firewall might raise an alarm too often and the false positive rate would increase. Conversely, if k is too high, the path likelihood will capture many events in a large time interval and the result might not be relevant either. Another important parameter to play with is the *threshold* for triggering an alarm, defining the minimum value of the path likelihood that is considered normal. The formula for calculating the path likelihood is strongly inspired by the course of Jurafsky *et al.* [48] and given by:

$$\prod_{i=1}^k P_{t_i} \text{ if } C_{s_i} \geq n,$$

with $P_{t_i} = C_{t_i}/C_{s_i}$ where C_{t_i} is the number of transitions i , C_{s_i} is the number of states i , and P_{t_i} is the probability of transition i in the sequence. The computation

of this formula is illustrated in Figure 17. We assume that the transitions are independent, so the likelihood of the sequence is just the product of the probabilities. A first remark is that all states are not always taken into account in the product. In fact, if the state is infrequent, it is ignored because the probability of its transition may not be relevant. On Figure 17 we can see that t_2 is not taken into account because S_2 is not frequent. If the current path does not contain more than x frequent states, then the whole sequence is ignored and not compared to the threshold. A second remark is that this formula is not implemented as just stated. The reason is that the multiplication of probabilities for a long path will tend to zero. To solve this, the formula is modified a little by taking the logarithm of the product, which gives [48]:

$$\sum_{i=1}^k \log_2 P_{t_i} \text{ if } C_{s_i} \geq n$$

An advantage of this new formula is that for large k , there can be an improvement in performance since it is no longer a floating point multiplication but a sum which is faster to compute.

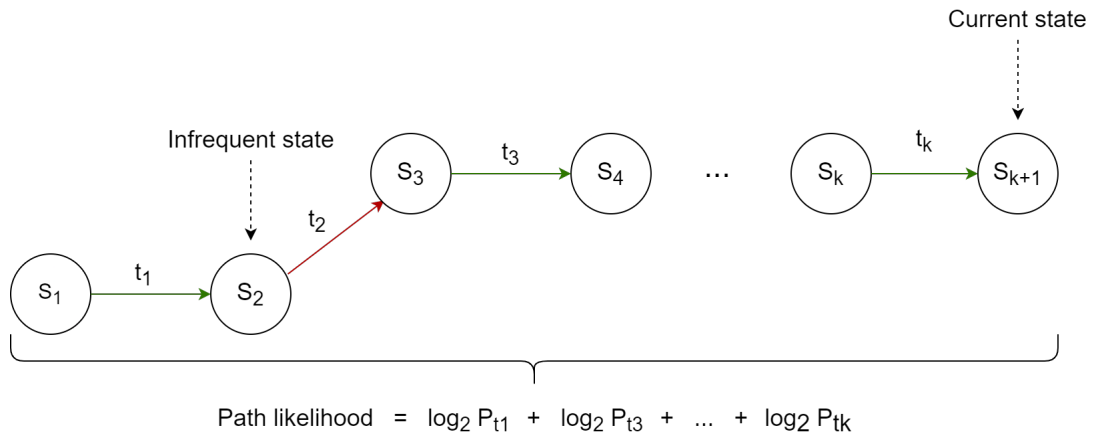


Figure 17: Path likelihood computation.

3.5 Parser and handler

Now that we have explained the technical entities of the firewall, we show how the system is configured by the user. This is done through the parser and the handler. The former is responsible for the correct interpretation of the user configuration file while the latter initializes the Nftables daemon and the context module with its input.

3.5.1 Configuration file

In fact, the very first step in using the firewall is the design of a configuration file suitable for the desired smart home configuration. This file must follow a specific

syntax which is detailed below. The input file is encoded in JSON format with a rather permissive syntax, it is composed of optional variables defining the configuration. Some of them are only related to the context while others also define the layout of the network. In any case, most of them are defined by unique names. Different examples are contained in Appendix A.

The categorization is used to label decimal values. There can be two types of categorization, decimal or time. A time categorization is composed of an array of numbers in string format followed by an array of labels. The first array represents the range associated with the labels, so the size of the label array must always be one less than the size of the float array. We enter the numbers in string format to allow "inf" and "-inf". The categorization time is used to label a time interval with a precision in minutes such as "08:30". Then, a member is defined by specifying its source and one or more optional fields. The source is an IPv4 or IPv6 string with an optional bit mask and a port such as "192.168.1.1/24;1883", with a semicolon for the port and not a colon for parsing reasons. A field can be of two types, decimal or string. A decimal field is linked to a categorization by its value and the initial value must be chosen from the labels of this categorization. On the other hand, the value(s) of a string field can be defined directly in the table of values with an initial value from this table. A relation is conveniently defined by providing its subject and members followed by one or more optional constraints. The members are defined by their name, the publisher and the subscriber are mandatory but the broker can be omitted. Then, we can declare all the constraints. Each of them is bound to a field of the publisher from the relation and the allowed value must be taken from the value array for a string field or from the labels for a decimal field. Next, there are variables that define how the context performs transitions. Triggers are defined as a list, each containing a condition and an action. A condition can be either on a member field or on a time period. For a field, one must define the specific value to be observed. For the time period, one can use a categorization or directly insert an interval. Actions can be the activation or deactivation of the desired relation. Finally, inference and inconsistency are defined in a very similar way. Each inference is composed of a condition and an implication. The condition is the same as for triggers and the implication has the same syntax as a condition except that it defines what is inferred by the context. Each inconsistency is defined as a set of one or more member fields associated with a specific value.

Following this specification, a compliant input file can be produced. After sending it to the client, the parser is responsible for converting it. In case of illegal syntax, the parser provides specific error messages used to help the user conform the file to the correct format.

3.5.2 Building Nftables

The rules are extracted from the parsing step and used to initialize Nftables. A relation can be derived in two or four rules, depending on the presence of a broker. These rules are associated with a *mark*, and then the packets that correspond to

them are marked accordingly during the filtering step. For example, adding an MQTT rule will add four rules in total and they will all be marked with the same mark. This mechanism is useful for mapping network layer packets to relations. However, it is possible that multiple relations share the same mark and in this case we use relation mapping to map different subjects to the same network layer signature, which works since the subjects are supposed to be unique.

Before adding a rule, the Nftables ruleset is completely cleared, which ensures that no interference is created by other policies. Next, three chains are generated with the highest priority in a filter table. These chains support the `inet` family, which means that we are able to match both IPv4 and IPv6 in a single rule and we also add rules such that they match both TCP and UDP packets. Each chain is dedicated to a hook, the three mentioned previously. However, these chains do not contain any rules. In fact, each of them contains a single instruction to match anything and jump to another chain called "merge". The latter contains all the rules. The reason we did this is that we wanted to give the user the ability to put a broker anywhere, including on the firewall itself. In this way, the firewall captures packets destined for other devices as well as the firewall host itself. One last thing to mention is that the default policy is to accept, but it is possible to provide the `--no-dev` argument setting the default policy to drop.

3.6 Workflow

Now that we have presented all the pieces, we complete the puzzle by showing the entire workflow. The purpose of this section is to demonstrate the various steps in the working process, from initial configuration to the update of the context.

3.6.1 Initialization

The initialization is an unavoidable part of the process. After designing an input file, it is transmitted to the client (with the `--input` option) who analyzes it and then forwards it to the handler. The latter is responsible for the initialization of Nftables but also for generating the context input from the parsed content. Eventually, the context module can be initialized with its input.

3.6.2 Filtering process

After the initialization process, the firewall is ready to process packets. From there, the process is represented in Figure 18 where the black arrows represent the order of the steps. Each time a packet is sent, it must pass through the firewall. At this point, the firewall performs two matches: the low layer match using the Nftables component, followed by the high layer match in the processing queue.

Low layers matching

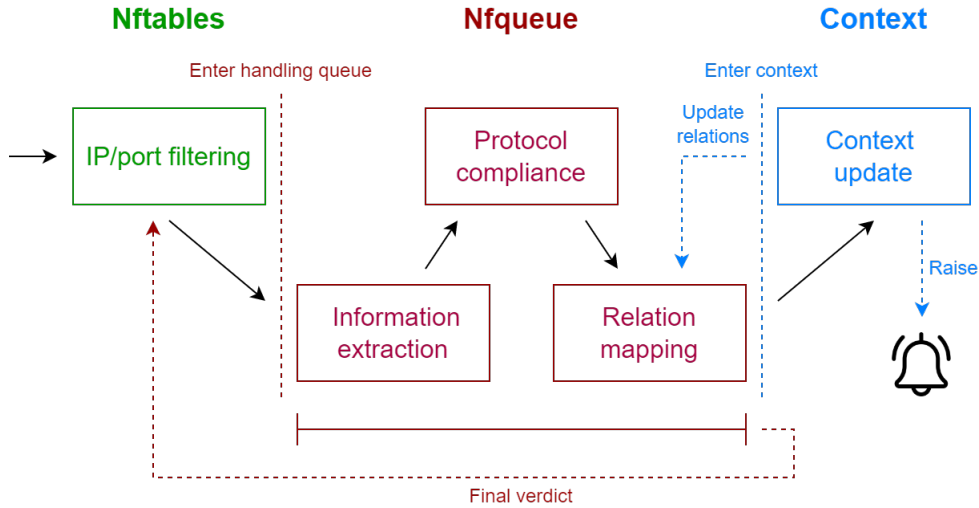


Figure 18: Workflow after initialization.

When we refer to the lower layers, we are specifically referring to IP plus TCP or UDP. In fact, data link layer frames are abstracted and already reconstructed by the network stack. However, it would have been possible to consider and filter them at a lower level. The very first step is IP/port filtering: when received, the packet will either match a rule from the Nftables ruleset produced by relations or it will not match anything and will follow the default policy. If it matches a rule, the packet is marked and placed in the handling queue as shown in red on Figure 18. This queue is unique, so all matching packets go there, which can produce an overhead during a burst, as measured in Section 4.2.

High layers matching

This matching is related to the layers above the transport layer. Currently, we only consider the application layer, but improvements could also consider the presentation layer for security aspects. Once a matched packet is out of the queue, it is used to build the abstract packet by extracting information from the network, transport and application layers. The first two are handled by external libraries while the application decoding is done by the specialized protocol decoder.

Once these information has been successfully extracted, the packet state is responsible to retrieve information from the protocol decoder to ensure protocol compliance. This information determine whether the protocol is behaving correctly but also what type of packet it is, such as a signaling packet or an actual content packet. This data influences the tuple returned to the handling queue for further processing. The tuple specifies whether the packet should already be dropped, whether it should be checked against the constraints, and whether it should be transmitted to the context. For example, a packet that does not conform to a supported protocol will not be allowed to continue processing, while a signaling packet will be allowed but will not be delivered to the context. If it is a content packet such as a publish message, the last step in the final decision is constraint matching. This involves

looking up the relation in the relation mapping dictionary using the mark, but also the subject to differentiate relations with the same mark. The relation mapping also contains information about the status of the relation, whether it is active or not. This last step returns the final decision about the packet to the Nftables daemon, either to drop it or to allow it. It is then queued in the context if necessary.

An important remark is that there are packets that do not go through this matching but are still accepted. This includes all transport layer signaling packets such as TCP SYN or SYN + ACK. In practice, this is not a major problem because they do not contain application data and therefore cannot directly violate contextual or behavioral integrity.

3.6.3 State update

Now that the firewall has finished filtering, the context is updated as shown in blue in Figure 18. The update is performed using certain packets that have been accepted and contain information relevant to the network state update. These include the publish packets of push protocols and the response packets of pull protocols. Each tuple (field, value) previously extracted from the packet payload is passed independently to the context. Then, the floating points values are categorized through label mapping.

Once the values are converted to strings, each tuple is passed into an FSM where each node corresponds to a possible state of the smart home. For each inserted field value that is different from its value in the current state, a transition occurs to a state that contains the corresponding value for the field. At the same time, if one or more triggers are attached to this transition, then one or several relations are activated or deactivated. In this way, we avoid entering inconsistent states. The likelihood of the current path is also maintained and updated each time a transition is performed, it represents the likelihood of the last k transitions traversed.

Inconsistent states are states that should be avoided at all costs by the firewall because they are considered dangerous by the user or because they can lead to a successful attack. If the context is still in one of these states, a stealth alarm is triggered. This also applies to the path probability, if its value drops below a threshold, an alarm is also fired.

4 Evaluation

We provide a small evaluation of our prototype to prove its efficiency but also to point out its shortcomings. This section begins by explaining the configuration of the environment so that any reader would be able to reproduce it. Next, we measure the latency and throughput of our system under different conditions. After that, we evaluate the context module, a crucial part of the implementation. Lastly, we describe some use cases to demonstrate the firewall’s potential and capabilities.

4.1 Experimental setup

The purpose of this section is to detail the experimental setup used during our research and experiments. By following this description, one should be able to recreate the network and reproduce the tests. You can access the entire implementation on our [GitHub repository](#). As it is a prototype, we chose to develop it in the Python programming language, as discussed in Section 5.2.1.

4.1.1 Simulation

Our entire project was conducted in virtual environments, we left the evaluation on hardware to possible future research. Here we describe how we created this simulation and how we emulated the behavior of the different devices.

Virtualization

We exploited virtual machine technology and in particular Vagrant [49] to produce a sandboxed environment. The latter consists of four machines, one for each type of device (publisher, subscriber and broker) and the firewall. The devices have one processor core and one gigabyte of RAM, while the firewall has twice as much. Each machine runs an instance of Fedora 35 [50] and is directly connected to all other machines in a complete mesh. The complete configuration is contained in the Vagrantfile of the repository. This type of environment has some advantages. The environment can be quickly recreated and tested on another machine. It also provides a fully isolated sandbox, ideal for network evaluation because no real delay is added. However, virtual machines suffer from host interference, such as context switches caused by scheduler choices. For this reason, our tests may not be completely accurate, but they still give a good trend.

Behavior emulation

In order to test our implementation, we had to develop scripts using high-level libraries to emulate the exchange of real devices. On the MQTT side, we mainly used Mosquitto [51] from the mosquitto package. The latter is very useful because it provides a CLI to start a broker, but also to publish and subscribe on topics. We used it in combination with the Paho Python library [52], which allows to do similar

and more advanced operations than Mosquitto through scripts. On the CoAP side, we mostly used the Python library `aiocoap` [53] which provides a friendly interface and supports many features, such as the ability to observe a resource. The tests were performed using the default settings of these libraries, such as lowest QoS for MQTT and confirmable packets for CoAP.

4.1.2 Deployment

The tests were run on Ubuntu 20.04, using an AMD Ryzen 7 3700X processor with 16 gigabytes of RAM, but the virtual machines received far fewer resources as mentioned above. In the repository, we provide a README explaining how to setup the virtual configuration. Note that the firewall must be started with the `--no-dev` option to enable the default Nftables drop policy, otherwise the default policy is accept. This option allows greater convenience with SSH, preventing the firewall from blocking its own host. We would like to emphasize that our implementation is not tied to the base configuration. It is possible to change any parameter and redeploy the firewall, for example, to change the operating system or to deploy it on real hardware, provided certain conditions are met.

The most important requirement is that the firewall must intercept all end-to-end communication, and this can be satisfied in different ways depending on the configuration. In a virtualized environment, this means enforcing traffic by setting the firewall as the default gateway with static routes and disabling redirection, or simply not interconnecting devices with each other. With real hardware, wireless communication should be preferred, which really simplifies the process. Indeed, if the firewall acts as an access point, it is already intercepting all traffic from connected devices, unless the wireless network is configured otherwise. In any case, make sure that IP forwarding is enabled.

The firewall must also meet certain requirements. It must support Python 3.10 and install the necessary libraries. It must also have some computing power, at least equal to that of our virtual configuration, but more is better. Not surprisingly, using our firewall will completely erase the host's Nftables configuration to fulfill its own policies. Other than that, there are no restrictions on IoT devices, as long as they meet the standards of the protocols we use.

4.2 Latency

We ran several tests to measure latency and show the concrete impact of our firewall on the end-to-end delay of message exchanges between devices. Keep in mind that these tests are not completely accurate since they are based on a virtual environment, but they do show some interesting trends to discuss. The scripts used to perform them can be found in the [evaluation part](#) of our repository.

4.2.1 Burst test

The purpose of this test is to measure the delay induced by the firewall without considering the context module. In fact, the major impact on delay is not caused by the FSM but by the handling queue. The latter is implemented as a separate queue that runs in its own thread. For this reason, all arriving packets are queued in the same place, which is a source of delay in case of a burst. We seek to experiment with the limit of this handling queue and estimate the maximum latency. All these tests were performed with the same parameters for the network configuration which are five dummy members with one field of two values and 25 dummy relations. To this we can also add the three real members used for the test: the publisher, the subscriber and the broker. These members have no fields except for the publisher which has one to store a timestamp. Finally, we add a relation with the broker for MQTT and a relation without it for CoAP.

MQTT

We start by testing MQTT with and without firewalls. For this test, we use a subscriber, a publisher and a broker. The publisher sends bursts of publish messages of different sizes sequentially with a waiting time between each burst. To create a burst, the messages are also sent sequentially with a time interval. These messages contain a single field value with the time in nanoseconds, for a total payload length of 27 bytes. On the other side, the subscriber waits for the messages and calculates an average of the end-to-end delay for each burst. A total of 10 bursts are sent to calculate the average delay for a burst. Each published message contains a timestamp and the delay is calculated by subtracting this timestamp from the current timestamp upon arrival at the subscriber. Thus, the delay includes the time to reach the broker and then from the broker to the subscriber. All these exchanges are done with a QoS equal to zero, so a single publish includes one publish from the publisher to the broker and one from the broker to the subscriber.

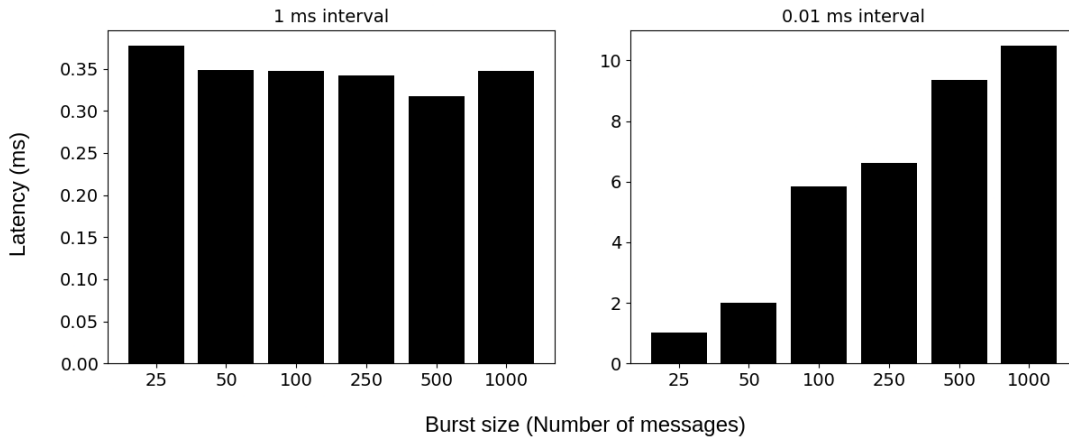


Figure 19: Latency versus burst size for different time intervals between each message with firewall disabled in MQTT.

Figure 19 shows the result without firewall. As we can see, with 1 ms, the interval time is large enough to cause almost no end-to-end delay. On the other hand, with a smaller interval such as 0.01 ms, we start to see a trend of the delay increasing with the number of messages, which is what we would expect. Now Figure 20 shows the results with the firewall. As we can see, with the 1 ms interval, we observe close measurements. This means that the messages are sent slowly enough not to cause any additional delay. Therefore, we can see the fixed latency added by the firewall which is maximum 2.5 ms. With an interval of 0.01 ms, we clearly notice jumps as the number of messages increases and this shows a trend of increasing latency with burst size. During the experiments, we noticed that this delay was not caused by our complex matching process, but by the extraction of packet information, which is managed by an external library. If we compare the firewall on and off, we can conclude that the firewall adds delay but in a reasonable way. Thus, using our firewall with MQTT should have no noticeable impact. However, we noticed that an interval less than 0.01 ms with large bursts (e.g., 10000 messages) would start to generate packet loss. We therefore consider the lowest stable interval to be 0.01 ms.

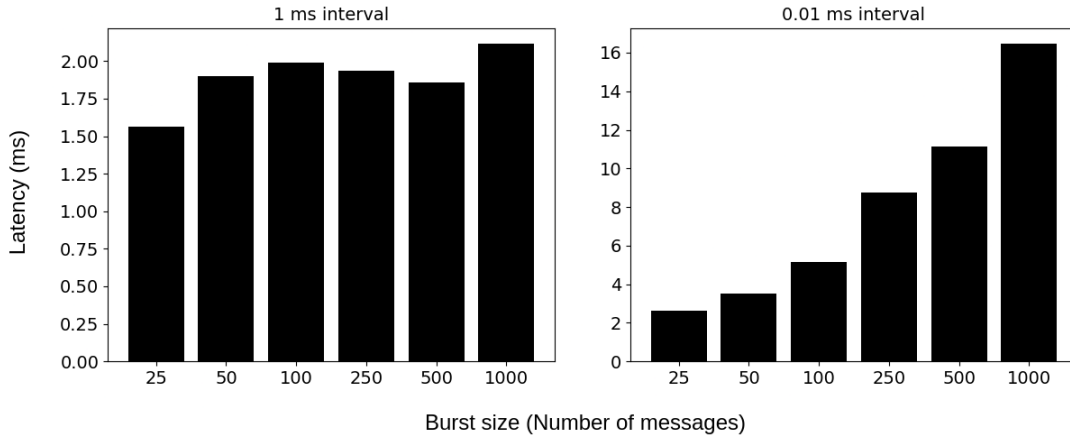


Figure 20: Latency versus burst size for different time intervals between each message with firewall enabled in MQTT.

CoAP

We performed a similar comparison for CoAP, but recall that it is a pull protocol. For this reason, we performed the test by sending confirmable GET requests from the subscriber to the publisher's "bench" resource. Upon reception, the publisher places a timestamp in the payload and returns the response. The subscriber is then able to calculate the difference with the current timestamp. To simulate a burst with different time intervals between each message, each request task is assigned an increasing delay before sending. As in the MQTT test, each burst is separated by a waiting time and the average end-to-end delay for a burst is computed in the same way. In the end, the delay includes the response time from the publisher to the subscriber, but keep in mind that the firewall queue is already filled with requests when the response arrives.

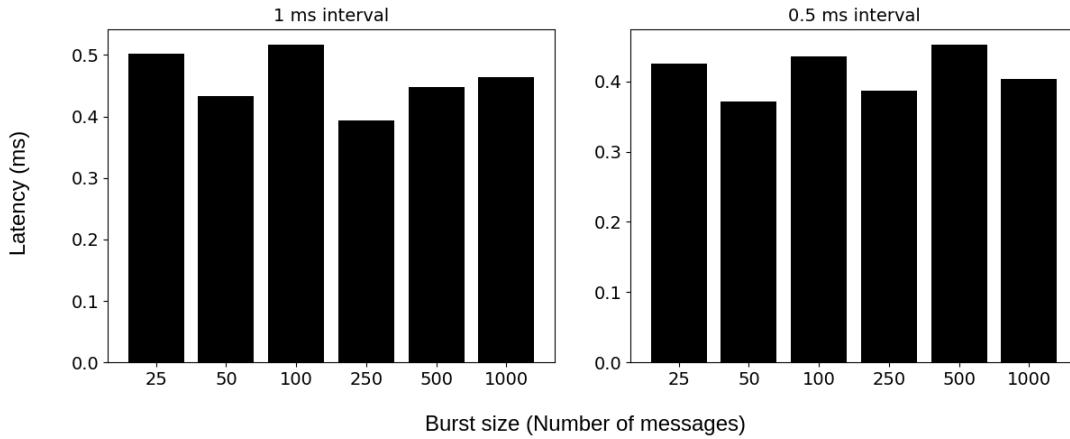


Figure 21: Latency versus burst size for different time intervals between each message with firewall disabled in CoAP.

Figure 21 shows the measurements without the firewall. Notice first that the intervals we used are not the same as for MQTT. This is because the test with CoAP becomes unstable much faster than the one with MQTT, for this reason we had to increase the interval time in order to make a consistent test. In the figure, we can see that for any time interval between packets and for any burst size, the average latency does not vary significantly. This means that these time intervals are high enough to not create congestion.

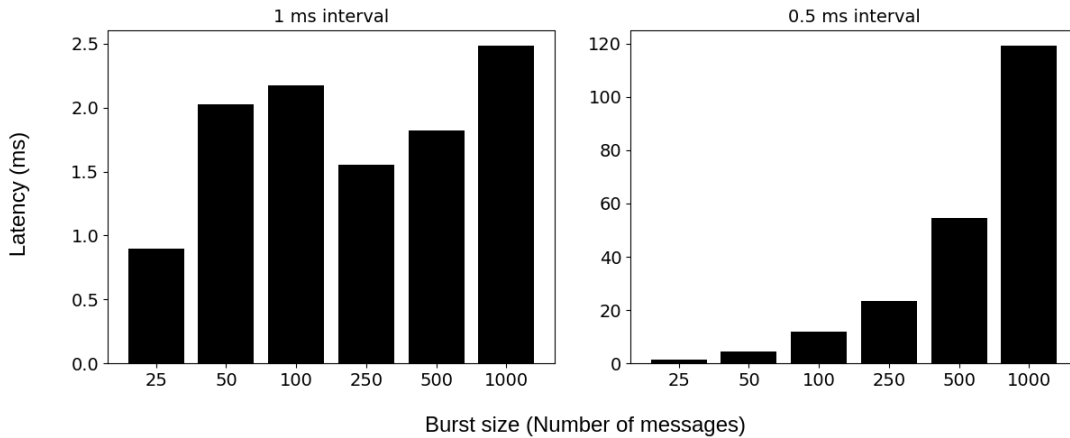


Figure 22: Latency versus burst size for different time intervals between each message with firewall enabled in CoAP.

On the other hand, Figure 22 shows the impact of the firewall. For a 1 ms interval, we still do not see a major difference based on burst size and the delay is quite similar to MQTT. However, with an interval of 0.5 ms, the latency literally explodes to over 100 ms, showing the actual impact of the firewall. In this case, we can already tell from the 1 ms test that the firewall adds a fixed latency of maximum 2.5 ms. The latency observed below 1 ms can be partly explained by the fact that CoAP is a pull protocol, i.e. with each response generated upon a request. In addition, it should be remembered that CoAP is based on UDP and

that it is good practice to send messages as confirmable, which is what we did. Thus, each packet sent actually generates four packets: the request, the response and their acknowledgments. In addition, we begin to observe retransmissions below 1 ms, resulting in more packets and thus more delay. Nevertheless, we believe that some of the delay is also generated by random factors in the simulation itself.

4.2.2 Scaling test

We said earlier that the context module does not cause much delay. This is mainly due to the fact that the time taken by the FSM to update itself according to the observed values does not depend on the number of states and transitions, it is done in constant time. To verify this, we performed a small-scale test by testing the end-to-end delay with the same techniques presented in the burst test, but this time with a fixed burst of 100 messages every 1 ms in MQTT. For this test, a script was used to generate different parameters for the input file. The idea is to provide three parameters M , F and V corresponding respectively to the number of members, fields and values. Each member is given the same number of fields and each field the same number of values. Then, a relation is created between each member until the upper limit of 100 relations is reached. For example, for $M = 5$, $F = 2$ and $V = 2$, five members with two fields of two values are created and 25 relations are formed. In this specific scenario, the total number of states is equal to $V^{(M \cdot F)}$.

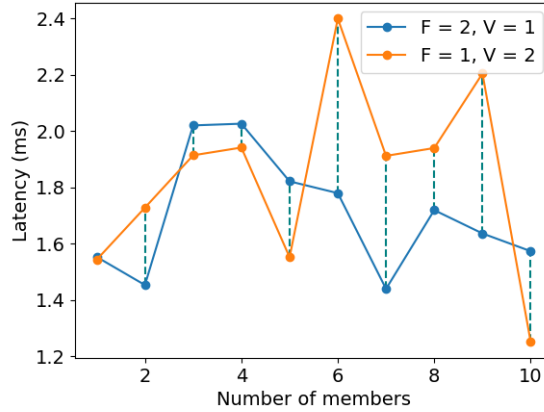


Figure 23: Latency versus number of members with different network configuration settings in MQTT.

The results of the test are shown in Figure 23. It is interesting to focus on the yellow curve because the number of states is mainly influenced by the number of values. On this curve, the initial number of states is two but it is exponential and it is already equal to 1024 when $M = 10$. For the blue curve, since $V = 1$, the number of states is always equal to one. As we can see, in both cases, the delays fluctuate randomly, mainly due to random factors of the environment and not because of the size of the FSM. One note on this test is that the scale is kept quite small, and this is because of the state explosion problem. In fact, while adding members and fields is still manageable, adding values quickly increases the build time and takes up a lot

of memory. The build times for the FSM and Nftables are given in the Appendix B and solutions for this problem are proposed in Section 5.3.1.

4.3 Throughput

Another interesting measurement to make for a network application is the maximum throughput. In our case, we define it as the maximum number of messages that can be sent while maintaining an acceptable end-to-end delay. The tests are performed by sending a constant stream of publish/requests with a constant time interval between each message. We determine the threshold for both MQTT and CoAP for different intervals, and again, the results may not be completely accurate due to the simulation.

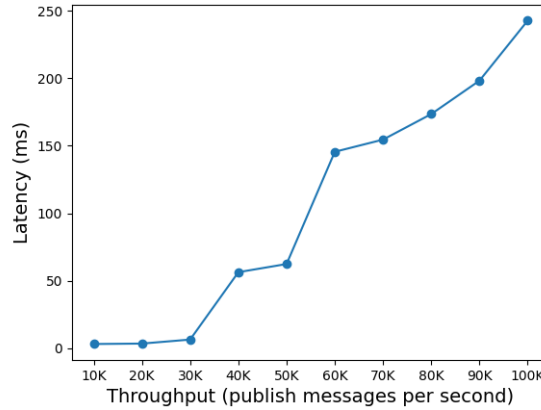


Figure 24: Firewall throughput for MQTT.

As can be seen in Figure 24, MQTT can handle a large number of publish messages until it becomes unstable. Indeed, we notice that the delay remains relatively low until the throughput reaches 30000 messages per second. After this point, the delay starts to fluctuate a little but still increases with the number of messages. At the end, we reach a maximum delay of about 250 ms for 100000 messages, which is quite large. Looking at the curve, we can estimate that the maximum stable throughput is close to 30000 messages per second because this is the last point before the delay explodes and exceeds 10 ms.

In figure 25, we can see that the latency starts to explode after a throughput of 800 requests per second, and then it reaches a latency of more than one second for 2000 messages per second. Thus, we can deduce that 800 requests per second is the maximum stable throughput for CoAP because the graph follows an exponential trend after this point. One comparison we can make is that the firewall is able to handle much higher throughput with MQTT than with CoAP, for the same reasons as explained in the previous subsection.

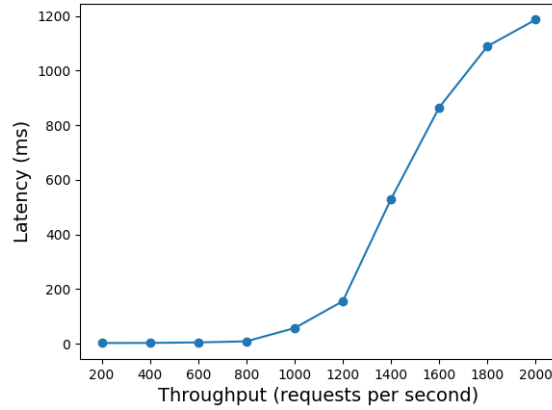


Figure 25: Firewall throughput for CoAP.

4.4 Context module assessment

The performance of the context module is a crucial point because it determines how fast the firewall is able to update the network context and adapt its policies accordingly. In order to evaluate this module, we conducted three measurements: mean queue time, mean service time and mean queue size. The first measure is the average time a packet spends in the context module's queue. The second is related to the time taken by a packet to update the current state. Finally, the third is the average number of packets in the queue. These measurements are shown in Figure 26, they were performed with CoAP and six devices in the network for a total of 64 states in the FSM. As we did previously in the latency tests, we performed the measurements by varying the burst size and also considered different time intervals between packets.

For the average time in queue, we measured the time between a packet entering the queue and when it begins being served. Unsurprisingly, the average time in the queue increases with burst size, but there is no significant difference between the three time intervals. It is important to note that a packet will experience a delay of almost half a second if the burst size is 1000. Fortunately, this implies having a very large amount of traffic.

The second test on the average service time confirms that the latter is not influenced by the burst size. We can observe a slight decrease between 250 and 1000 burst size, this is due to the fact that the delay in the queue becomes so high that retransmission packets are sent. This causes the current state in the FSM to self-loop because the same packet is duplicated several times. However, the execution of a self-loop is faster than a normal transition and thus the average service time is slightly reduced.

Regarding the queue size, the test clearly shows that the average increases with the burst size. For a time interval of 0.5 ms between packets, the average queue size

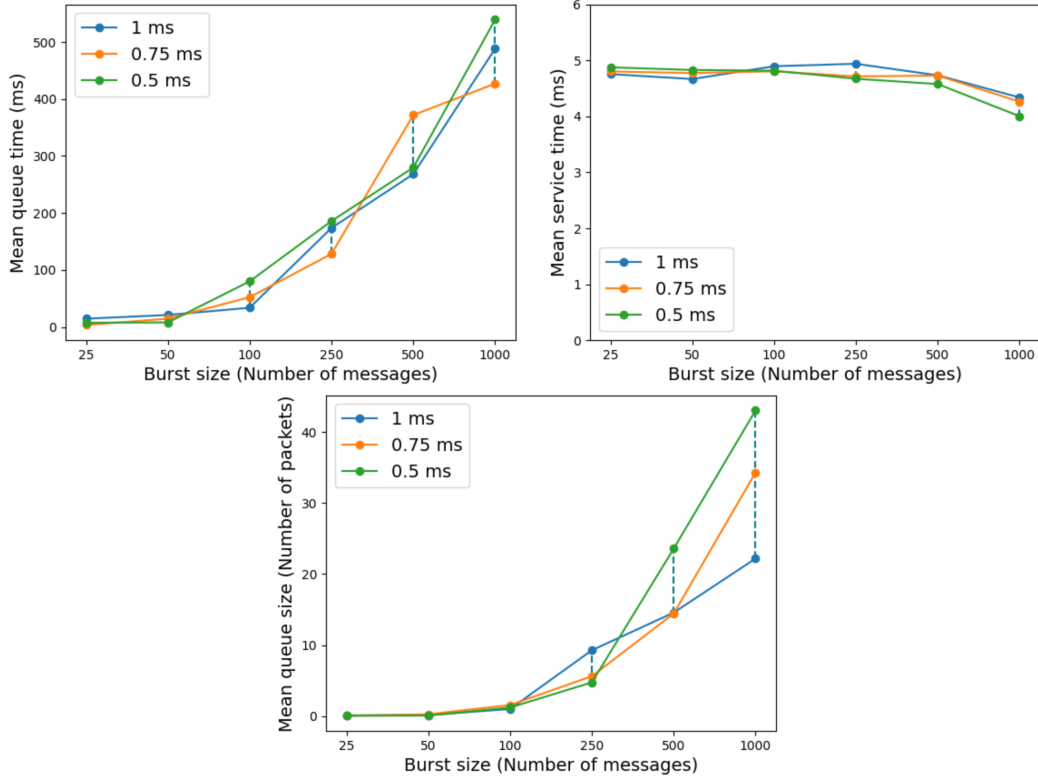


Figure 26: Context module performance measurements, with its mean queue time, mean service time, and mean queue size versus burst size.

is 43 packets while it is only 22 packets for a time interval of 1 ms, indicating that the queue size increases significantly with the packet sending rate. It is important to note that the average queue size was measured by taking the queue size every five milliseconds in a thread running in parallel, which may not be the optimal way to proceed and may have led to some inaccuracies.

4.5 Use cases

Now that we have shown the performance side of the evaluation, it might be interesting to present some concrete use cases. The point is to demonstrate the usefulness of such an installation. We first consider a scenario in which the state of the relations depend on the presence of the occupants of the house. Second, we consider a scenario where the relations are disabled during the night. Finally, we show a more unconventional example with a vacuum cleaner. We provide the corresponding input files in Appendix A, all models are quite small and they build instantly. These configuration files are hypothetical in that they can be modified in different ways to fit the scenarios presented. For example, we create self relations (a relation to self) to update some sensor states in the firewall because they do not have a subscriber. This might be different in scenarios with subscribers. Also note that the IP addresses are random and are used for demonstration purposes only.

4.5.1 Home presence

In this scenario, we highlight a smart home that detects when the user enters and exits the house as shown in Figure 27. There is a motion sensor in the entrance, which maintains a binary state for detection. If it has no subscriber, it could share a relation with itself to update its state in the firewall. Next, consider a relation to trigger a potentially dangerous home device, such as a heater. If we assume that the sensor is not corrupted, this relation could be linked to the motion state so that as soon as motion is detected, the relation is activated. In this way, if the occupant enters (resp. leaves) the house, it activates (resp. deactivates) the relation. Nevertheless, let's consider the case where the motion detector is corrupted. In this case, we could combine the motion detection with a door sensor, which detects the opening and closing of the door. Then, we could detect the presence of the house occupant with the binary state of the door sensor and the motion detector. This makes it more difficult to deceive the firewall. However, there are some limitations to this scenario, as the motion sensor may get stuck in one state due to an unexpected detection. It is in fact a false positive, which could be triggered by an animal approaching the sensor for example. To solve this problem, we can simply rely on the door sensor alone or add another factor such as the state of the living room light.

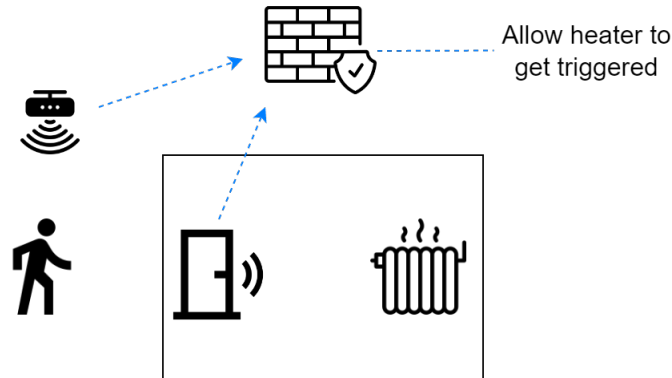


Figure 27: Home presence use case.

4.5.2 Night time

In this second scenario, we explore the use of time-based triggers. A typical situation in smart homes is that certain devices should not be triggered at specific moments. For example, it is not generally desired that the window opens during the night (at least in winter), this is exactly the situation that the Figure 28 is depicting. A set of devices such as windows, shutters or even lights are associated with a triggering relation. However, when night comes, we could decide to restrict all these relations, thus prohibiting any access to them during this period. In fact, one could decide to do this at any time interval, for example during absences from the home. This example shows the usefulness of temporal triggers and how they can be used to express complex policies.

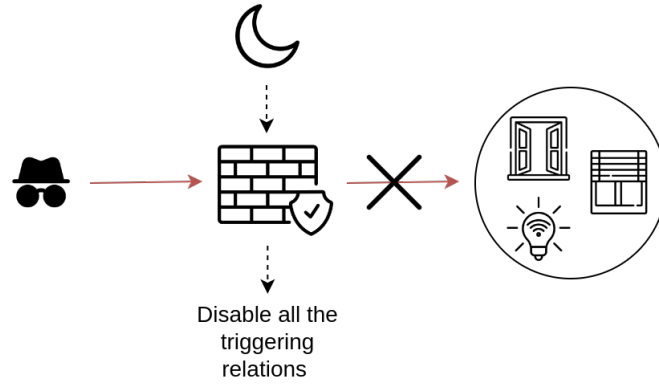


Figure 28: Night time use case.

4.5.3 Vacuum cleaner

The last scenario is a bit more sophisticated and considers a self-moving vacuum cleaner, a connected window with its switch and a motion detector, as shown in Figure 29. In the red part, an attacker is able to corrupt the switch and wants to open the window by publishing a message. The relation between the window and its switch is disabled when no motion is detected by the sensor. However, the attacker finds a vulnerability in the vacuum cleaner. Thus, he is able to issue messages to this object in order to reschedule the cleaning at a time when he knows that no one is at home. The vacuum cleaner then triggers the motion sensor which issues a message indicating motion. The firewall now thinks that someone is home and activates the window relation. The attacker is now able to open the window by triggering the switch.

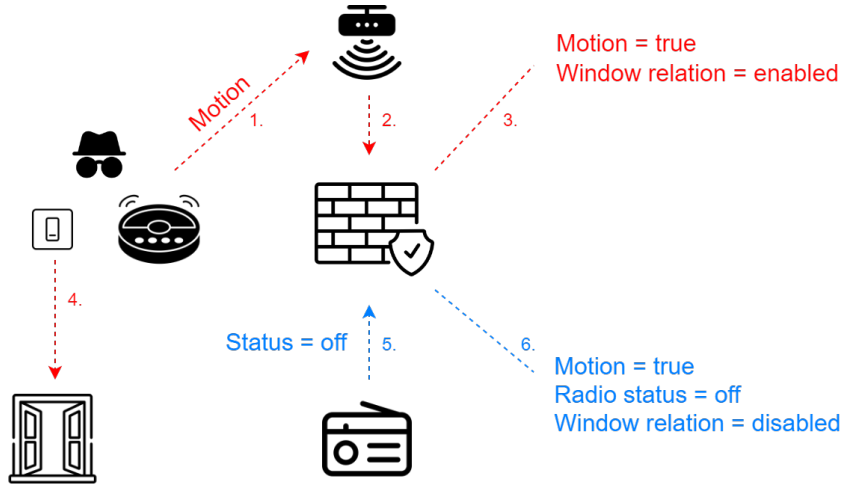


Figure 29: Vacuum cleaner use case.

In this situation, a physical interaction between a device and a sensor was used to trick the firewall. It is possible to mitigate this type of attack by adding additional conditions to the triggers. In the blue part of the figure, we see that if we add the fact that the radio must be turned on to activate the window relation, the adversary will also have to corrupt this device to perform the same attack. We can conclude that

network security is highly dependent on user specifications. Note that if the vacuum cleaner is not corrupted, it would already trigger false positives on the scheduled times and we would also have to add another state factor to mitigate it.

5 Discussion and limitations

This chapter mostly focuses on the downsides of our system, but also highlights other implementation choices that could have been considered. First, we focus on the security aspect of the firewall and how it could be enhanced. Next, we discuss some design elements. Finally, we talk about the issues encountered in the context module and propose solutions to solve them.

5.1 Security

Our security goals are quite limited, in fact we have reduced the security guarantees to preserve the behavioral and network state integrity. Moreover, we have also omitted two very important aspects of security: encryption and authentication. Here we discuss how we might add them to the system. After that, we also discuss packet state storage and how we could improve it.

5.1.1 Encryption

Encryption is the most important security feature used to ensure integrity, but its use renders our firewall unable to understand the intercepted traffic. Fortunately, this problem has been well-studied and there are many approaches to mitigate it. Here we describe some solutions that could be implemented with our firewall model. We present a solution specific to MQTT and then more general solutions based solely on TLS [54]. Finally, we present a machine learning alternative. While the previous methods involve data decryption, the latter operates directly on the encrypted traffic. Some solutions provide full encryption in both the internal and external network, while others provide only external encryption. Considering a solution that does not protect the internal network is interesting because the network can already be protected using a secure wireless protocol such as WiFi. When we talk about encryption here, we are talking about application layer encryption and in particular TLS encryption.

MQTT bridging

Before discussing general solutions, it is interesting to look at a solution specific to MQTT. Indeed, MQTT has an interesting feature called "bridging". It allows to connect two or more brokers together, thus creating a bridge. A side benefit of this mechanism is that it allows the creation of two sides, a certified side and a non-certified side, as shown in Figure 30 where a bridge is built between brokers A and B. As can be seen, devices B and C can represent the external network and use encryption when talking to broker B. Then, the messages enter the internal network and are transmitted between the brokers still in an encrypted manner. Finally, they reach the internal network device in clear text. Messages can also flow in the other direction. In this scenario, since the internal network traffic is not encrypted, it is not even necessary to update the implementation.

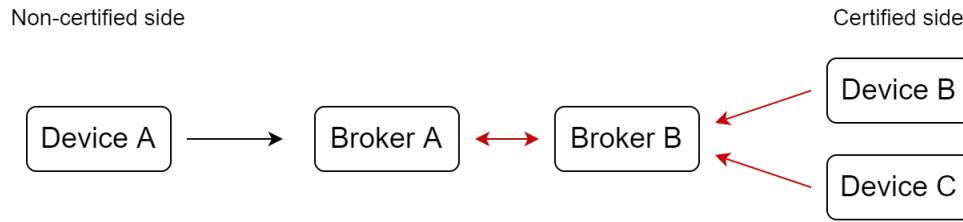


Figure 30: MQTT TLS bridge.

TLS termination proxy

There are also solutions that work for any TLS-enabled application protocol. The first approach we present is called TLS offloading [55]. It is actually quite similar to the MQTT bridging approach, it consists in defining a device such as the firewall as the termination proxy of the TLS encryption tunnel. In this scenario, any device connecting to the firewall from the external network must establish a secure connection. Then the firewall would be able to decrypt the traffic and inject it into the local network in clear text. Again, only traffic from the external network is encrypted.

If wireless protocol encryption is not sufficient, a solution such as a TLS bridging proxy should be considered. O'Neill *et al.* [56] provide a good description of how it works. As shown in Figure 31 inspired by the article, it consists in setting up an intermediate device, once again the firewall in our case, as a proxy. However, in that case, the firewall is responsible for providing a substitute certificate for the original certificate so that it can decrypt the traffic and then re-encrypt it. However, this approach has some drawbacks. It can require a large number of certificates for all devices and can quickly become cumbersome. In addition, the performance cost to the firewall becomes significant as it performs more operations.

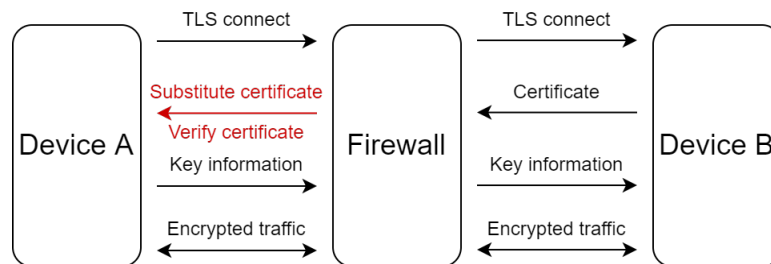


Figure 31: TLS bridging proxy.

Packets recognition

Previous approaches have focused on decrypting network traffic in order to analyze it. Nevertheless, it is also interesting to explore techniques to inspect encrypted traffic directly, without trying to decipher it. These solutions are based on machine

learning technologies that require a learning phase and a lot of data that we lacked, which is why we did not go down this path. They allow to recognize the packets and link them to specific behaviors. An example of such a technique is called PING-PONG, developed by Trimananda *et al.* [57], which is able to extract packet-level signatures for device events from network traffic, and then identify those events. The signatures consist of very specific information about the packets, such as their length or direction. In the paper, they were able to achieve 97% accuracy with signatures uniquely identifying events among hundreds of millions of packets. This type of technique works on both encrypted and unencrypted traffic, and event detection is fast because no statistical calculations are performed to identify an event. Indeed, with this solution, each packet is considered individually or in a longer sequence, and it is sufficient to retrieve some information in constant time without making any advanced calculation. Several types of events can be distinguished within the same packet flow, such as the difference between "on" and "off" for packets coming from a switch. Another advantage is that PINGPONG does not require any knowledge of devices and is able to generate its training data by triggering the events on which it will extract signatures. Such a solution does not offer perfect accuracy, but it is still a good trade-off to overcome the encryption problem.

5.1.2 Authentication

Authentication is one of the security principles that we do not respect in our system and the reasons for that are multiple. It is a well-known problem and there are many solutions to solve it. The problem is that either the solutions are not secure enough and there is no real interest in supporting them, or the solution is secure but coupled with encryption that we do not provide either.

If we take the example of MQTT, it offers different ways to authenticate. This can be done with a client ID, to which an optional username and password can be added. In theory, this solves the authentication problem, but in practice, without encryption, the credentials are sent in clear text and an attacker could sniff out the traffic. That leaves us with conventional authentication using TLS/SSL X.509 certificates. This authentication is more secure, but the recommended ciphers are coupled with an encryption scheme, which brings us back to the encryption problem. It is always possible to use a cipher suite without encryption, but this is not a good practice. Another problem we face is that both sides of an exchange need to be authenticated, which actually requires generating many certificates to ensure good security. Not surprisingly, CoAP faces the same problem because its authentication is primarily based on DTLS [58]. Some implementations provide a token-based authentication scheme [59] that is ineffective without encryption since the token is exchanged in clear text. In the following, we address the issue of spoofing against the firewall, and then consider a fingerprinting approach that does not require TLS.

Spoofing

One of the most annoying attacks we can face against authenticity is definitely

spoofing. In fact, one of our limitations is that we recognize members by their network/transport layer signature, which makes the firewall vulnerable to spoofing. This leaves the door open to multiple attack scenarios. If a device A is corrupted, it could forge its identity to act on behalf of another device B that shares a relation with device C . In this case, the firewall would have an active relation such as $B \rightarrow C$, and the corrupted device would be able to send messages through this relation. Even if a device is not corrupted, this type of scenario can also occur, for example, if the internal network is open to the Internet. Fortunately, in practice, this attack is "naturally" countered by TCP handshake. Indeed, if an adversary tries to establish a connection on behalf of a network device, the real device will issue an RST message and block the adversary. Now, what happens to UDP-based protocols such as CoAP? In this case it no longer works, but thankfully some libraries [53] offer the possibility to use TCP instead of UDP. We are aware that this is very insecure, but because of this mechanism, spoofing is still limited.

Fingerprinting

Even without using a dedicated authentication scheme as in TLS, it is still possible to uniquely identify devices. A solution proposed by Bezawada *et al.* [60] uses behavioral fingerprinting of IoT devices to identify devices with high reliability. They extract features from network traffic information to approximate device behavior, and then train a machine learning algorithm with these features to classify devices. The extracted information is diverse, such as TCP window size or payload length, with very specific values for certain types of devices. The main idea behind their solution is that each IoT device has a unique fingerprint that is revealed during packet exchanges. This makes spoofing an IP address insufficient to deceive the firewall.

5.1.3 Packet state storage

In the packet state class, pending requests for CoAP and subscriptions for MQTT are stored in two different dictionaries to be mapped to responses and publish messages respectively. This becomes a problem in case of heavy load on the network, especially in case of a Denial-of-Service (DoS) attack. Indeed, the firewall does not protect against such an attack and, it can cause the software to exceed the memory allocated to its process. To mitigate this attack, a first solution would be to limit the size of the dictionaries, but the attack will still prevent other legitimate devices from communicating with each other since the firewall will not be able to store their requests/subscriptions. A second solution is to timeout the request dictionary entries if a request goes unanswered for too long. It is also possible to limit the number of subscriptions a device can make in order to limit the size of the subscription dictionary. Countering DoS attacks is not the role of a firewall, but these mechanisms can at least attenuate them.

5.2 Implementation choices

The design choices we made are not flawless. First of all, the use of the Python programming language is not optimal. Also, relying on Nftables instead of providing a custom solution has an impact on performance. Finally, the firewall layout could also have been designed in a distributed fashion. We discuss these choices below.

5.2.1 Programming language

During the development phase, we decided to use Python instead of C or C++, and the reasons for this choice are multiple. Python is more programmer friendly, i.e. programs are generally written faster and are easier to read. This is due to the fact that Python alleviates tedious tasks because it has memory management, many useful built-in functions and a ton of importable libraries. Nevertheless, we are aware that Python is not suitable for network-critical application software. Mainly because it is interpreted, which makes it generally slower than a compiled language such as C++. The garbage collection also makes it slower than manual memory management. From a research perspective, Python is an acceptable choice, but a compiled language should be preferred for a full version of the firewall.

5.2.2 Low layers matching

Nftables is a very powerful solution and provides exactly what we need for lower layer matching. However, it has a few drawbacks: it is too powerful for the few features we need and, most importantly, using the "nfqueue" target requires processing the packet in user space, which is not optimal. Performing a critical operation such as packet processing should be done as quickly as possible, and that means in the kernel. This is the idea we had at the beginning. Instead of manipulating Nftables, we had implemented the hook functions with our own custom matching solution directly in the kernel. Again, we quickly realized that this was too advanced for a prototype. In fact, we had developed the basic IPv4 matching functionality, but we were missing many other features such as IPv6 matching. Moreover, kernel development is very burdensome and difficult to debug. Therefore, we decided to stick with Nftables because it is a reliable and complete solution. We still believe that implementing a custom solution that interacts directly with the kernel hooks is the way to improve performance. Such a solution would prevent the processing queue from being constantly interrupted as it runs in a low priority process.

5.2.3 Firewall layout

One of our goals was to limit the constraints applied to smart devices. This means that we intended to keep changes to the devices (except for the firewall) to a minimum. For this reason, we opted for a centralized firewall layout. In this situation, a single entity is responsible for filtering and the smart objects are not even aware of its presence. This approach has some disadvantages, as the firewall can only intercept traffic passing through it and not traffic directed to the devices directly. Nevertheless, the distributed firewall approach strives to solve this problem. Yu *et*

al. developed such an approach in their IoTSec prototype [61]. Within it, they instantiate what they call a *µmbox* that acts as a gateway on the devices to intercept traffic. Then, activity on the devices can be reported to the IoTSec controller that manages the interactions. If we consider an approach with a global controller communicating with each device, then it becomes more challenging for an attacker to tamper with the device directly because they report any suspicious activity and are also self-protected. But unlike the centralized solution, it requires modifying the device directly and this is often inconvenient or even impossible if the device is too constrained. We chose not to do that, but it is an interesting direction to explore.

5.3 Context

The context component is one of the most important in our firewall. Nevertheless, the FSM-based implementation suffers from some drawbacks, mainly the state explosion problem that leads to poor scaling. In addition, if two correlated packets are received in a too short time interval, it may cause problems. The following paragraphs propose solutions to mitigate these two issues.

5.3.1 State explosion

FSMs have many advantages such as interpretability or flexibility, but they have one major problem: state explosion. A node in the DFA represents one possible state of the smart home, and all of them must be considered. Indeed, we must consider every value of every field, and each time a field is added, the size of the FSM is multiplied by the number of values that field can take. Thus, the temporal and spatial complexities are exponential, which makes such a structure not very scalable. Fortunately, this is not a dead end and there are still ways to improve the situation.

Categorization

The first mechanism that mitigates the state explosion is categorization, which maps floating point values to string labels. This step is currently implemented and mandatory to eliminate continuous values that would otherwise create an infinite state space. In addition, a second possible improvement would be to add a field selection step where fields that are not relevant to the context are filtered out and never considered again in the DFA.

FSM decomposition

Until now, all fields were considered together in the same condition because the initial assumption was that they were not independent, but this is not always the case in practice. For example, if we consider a radiator connected to the second floor and an automatic lamp on the top floor, they are not likely to influence each other or appear in the same condition of a trigger. These fields are then considered as completely independent and could be considered in separate states in two FSMs

running in parallel. This idea of partitioning the machine was inspired by Cho *et al.* [62]. Reusing the previous example, if we also consider a motion sensor and a thermometer, we can create a DFA with states containing three fields: lamp status, heater status, and temperature. Figure 32 shows the resulting FSM with all possible states and transitions without parallelism. If the lamp is truly independent from the heater and thermometer, then it is possible to create an equivalent model with two FSMs each containing subsets of the fields of the original FSM, as shown in Figure 33. In the first solution, there are a total of eight states and 28 transitions while in the second solution, there are six states and ten transitions. We observe that the number of transitions decreases significantly and that 25% of the states have disappeared. This is due to the fact that many combinations of fields were redundant and many combinations of independent fields were removed.

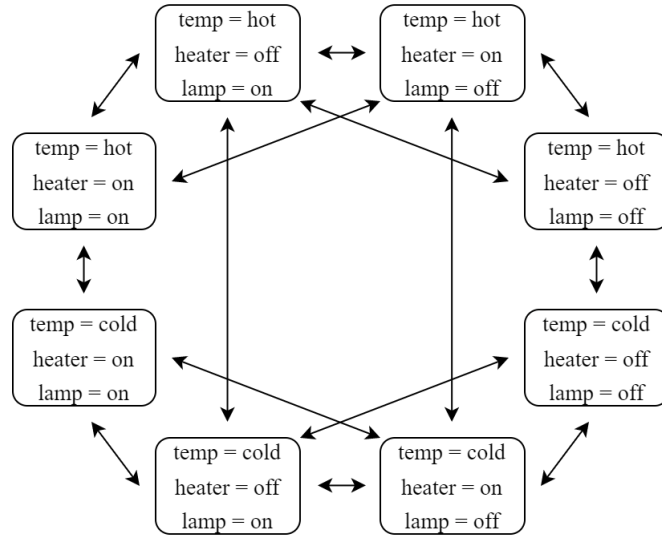


Figure 32: Complete FSM with all the states and transitions.

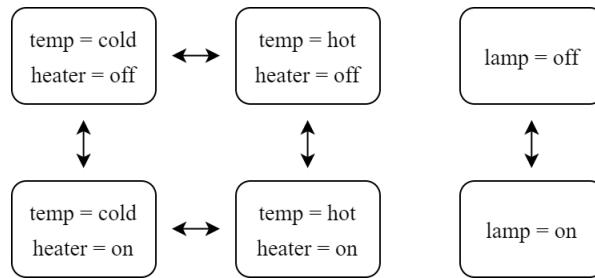


Figure 33: FSM decomposition.

Hierarchical FSM

As Yannakakis explains [63], a second improvement that can be considered is the use of a hierarchical state machine, i.e., a DFA in which some fields make sense only under a specific value of another field. In this case, it is possible to embed

these fields in a black box state that contains an internal FSM. For example, in the previous example with the lamp, heater, and thermometer, the user might consider the heater state relevant only when the temperature is high. In Figure 34, states with temperature set to cold do not have a heater field because the heater field is only present in the nested state. The number of states is also reduced to six, but the number of transitions is further reduced to 14, half as many as the original FSM. Note that the previous improvements are supported by the library we used for the prototype.

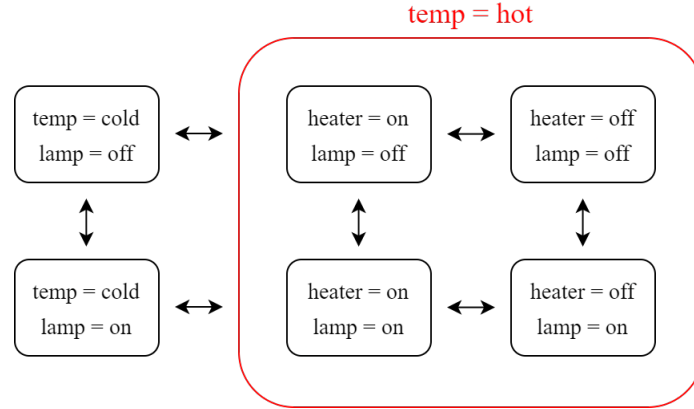


Figure 34: Hierarchical FSM.

Network implementation

Another way to improve performance would be to completely change the way the network context is maintained. This can be done by simply reimplementing the network class with the two functions needed by the trigger class. For example, code generation can be used with the user specification to create at runtime all the if statements that check if the condition of a trigger is true and thus apply the corresponding function (enable/disable) to the relation. This would solve the memory problem but would take more time to process each packet in the context. The choice of this compromise depends uniquely on the user's needs.

5.3.2 Inter-arrival time

Inter-arrival time can start being a problem if the delay between two correlated packets is too short. Packets are correlated if they are linked to relations that are bound by triggers. Here we consider this scenario and discuss the impact on the context.

The first issue arises if two packets follow each other closely. For example, consider a relation between the window switch and the window and another one between the heater switch and the heater. If one of the triggers is the deactivation of the window relation to avoid having to heat the house with the window open then the problem arises if a packet to turn on the heating is directly followed by

a packet to open the window. The packet to open the window will be already accepted and waiting in the queue to be processed by the context when the firewall will realize that the window relation should be disabled. The DFA will therefore enter an inconsistent state and raise an alarm. This scenario is also true during congestion, as the processing of the heat activation packet is delayed in the queue, giving time for the open window packet to be accepted.

A second issue is the reordering of the packets, it is possible that packets close in time do not arrive in the order they were sent. This can happen, for example, if some packets are lost before they reach the firewall and need to be retransmitted. In this case, the first packet arrives later than it should, giving the second packet the time to be processed first. Using the previous example, if the user tries to turn off the heater and then open the window but the window packet arrives before the heater packet, the window packet will be blocked by the firewall because the heater is still on. Reordering packets close in time so that the order matches the most likely scenario could help mitigate this problem, but this must be done with care because an attacker could abuse this mechanism to fool the firewall.

6 Conclusion

Nowadays, IoT security is an important concern and its applications need to be secured carefully. In this thesis, we tried to propose a solution to secure smart home environments. We have developed a smart firewall prototype that is able to dynamically adapt its policies based on the context. These policies are fully controlled and defined by the user. The principle of action of the firewall is divided into two phases, the filtering process and the context update.

The first phase of filtering consists in matching the network and transport layers using the Nftables technology. Next, the application layer matching is in charge of abstracting the information and behavior of packets. This abstraction step is performed by specializing the protocol decoder into a supported protocol decoder. In addition, the second matching is tasked with comparing the payloads to strict constraints. At the end of the filtering process, a verdict is issued to drop or accept the packet and it can also be passed to the context.

This context is designed as a finite state machine and is responsible for tracking the state of the smart home. This state is composed of the current state of various fields attached to the smart objects. Upon receiving a packet, the context is also instructed to perform transitions and apply various triggers. These triggers activate and deactivate relations, which represent interactions between members of the smart home. Finally, the context module also triggers stealth alarms when inconsistent states are reached or when unusual behaviors occur.

Our solution proved to be efficient, as we observed from the firewall-induced delay and the performance of the context module. Some of the numbers are high, but they only appear in extreme burst scenarios. However, the design also displayed some weaknesses. The main one being the state explosion that limits the size of the configuration a user can define. In addition, the configuration burden involved with our system is not negligible and we are aware that it is difficult to deploy.

As the IoT will continue to grow and become increasingly ubiquitous in our daily lives, we wanted to convince the reader that its security is an important matter. Our work demonstrates that it is possible to manage the diversity of the IoT and improve its security. We hope that the research community will carry on in this direction and make our future more secure.

References

- [1] OSI Model. <https://www.imperva.com/learn/application-security/osi-model>. Accessed: 20/02/2022.
- [2] Satyajit Sinha. State of IoT 2021: Number of connected IoT devices growing 9% to 12.3 billion globally, cellular IoT now surpassing 2 billion. <https://iot-analytics.com/number-connected-iot-devices>. Accessed: 16/02/2022.
- [3] Abhishek Khanna and Sanmeet Kaur. Internet of things (iot), applications and challenges: a comprehensive review. *Wireless Personal Communications*, 114(2):1687–1762, 2020.
- [4] Hannes Hartenstein and LP Laberteaux. A tutorial survey on vehicular ad hoc networks. *IEEE Communications magazine*, 46(6):164–171, 2008.
- [5] Fadi Al-Turjman, Muhammad Hassan Nawaz, and Umit Deniz Ulusar. Intelligence in the internet of medical things era: A systematic review of current and future trends. *Computer Communications*, 150:644–660, 2020.
- [6] Richard Harper. *Inside the smart home*. Springer Science & Business Media, 2006.
- [7] Smart home automation from Google. <https://home.google.com/welcome/>. Accessed: 19/05/2022.
- [8] IFTTT. <https://ifttt.com>. Accessed: 16/02/2022.
- [9] Wei Zhou, Yan Jia, Anni Peng, Yuqing Zhang, and Peng Liu. The effect of iot new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. *IEEE Internet of things Journal*, 6(2):1606–1616, 2018.
- [10] Anna Triantafyllou, Panagiotis Sarigiannidis, and Thomas D Lagkas. Network protocols, schemes, and mechanisms for internet of things (iot): Features, open challenges, and trends. *Wireless communications and mobile computing*, 2018, 2018.
- [11] LoRa Alliance. <https://lora-alliance.org>. Accessed: 16/02/2022.
- [12] Sigfox. <https://www.sigfox.com>. Accessed: 16/02/2022.
- [13] Wireless Sensor Network. <https://www.sciencedirect.com/topics/engineering/wireless-sensor-network>. Accessed: 19/05/2022.
- [14] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82, 2007.
- [15] Zigbee: The Full-Stack Solution for All Smart Devices. <https://csa-iot.org/all-solutions/zigbee/>. Accessed: 19/05/2022.

-
- [16] Better and safer smart homes are built on Z-wave. <https://www.z-wave.com/>. Accessed: 19/05/2022.
- [17] Vittorio Trecordi and Giacomo Verticale. An architecture for effective push/pull web surfing. In *2000 IEEE International Conference on Communications. ICC 2000. Global Convergence Through Communications. Conference Record*, volume 2, pages 1159–1163. IEEE, 2000.
- [18] MQTT: The Standard for IoT Messaging. <https://mqtt.org>. Accessed: 16/02/2022.
- [19] Quality of Service (QoS) 0,1, & 2 MQTT Essentials: Part 6. <https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/>. Accessed: 27/05/2022.
- [20] What Is an X.509 Certificate? <https://www.ssl.com/faqs/what-is-an-x-509-certificate/>. Accessed: 19/05/2022.
- [21] The Constrained Application Protocol (CoAP). <https://www.rfc-editor.org/rfc/rfc7252.html>. Accessed: 19/05/2022.
- [22] Observing Resources in CoAP. <https://tools.ietf.org/id/draft-ietf-core-observe-09.html>. Accessed: 07/05/2022.
- [23] What is the Mirai Botnet? <https://www.cloudflare.com/en-gb/learning/ddos/glossary/mirai-botnet>. Accessed: 18/02/2022.
- [24] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.
- [25] Ismail Butun, Patrik Österberg, and Houbing Song. Security of the internet of things: Vulnerabilities, attacks, and countermeasures. *IEEE Communications Surveys & Tutorials*, 22(1):616–644, 2019.
- [26] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *IEEE Communications Surveys & Tutorials*, 21(3):2702–2733, 2019.
- [27] Yimin Song, Chao Yang, and Guofei Gu. Who is peeping at your passwords at starbucks?—to catch an evil twin access point. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 323–332. IEEE, 2010.
- [28] Netfilter. <https://www.netfilter.org>. Accessed: 20/02/2022.
- [29] What is a Web Application Firewall (WAF)? <https://www.crowdstrike.com/cybersecurity-101/web-application-firewall>, 04 2021. Accessed: 20/02/2022.

-
- [30] Chickowski Ericka. Deep packet inspection explained. <https://cybersecurity.att.com/blogs/security-essentials/what-is-deep-packet-inspection>, 10 2020. Accessed: 20/02/2022.
- [31] What is Sandboxing? <https://www.barracuda.com/glossary/sandboxing>. Accessed: 20/02/2022.
- [32] The netfilter.org "iptables" project. <https://www.netfilter.org/projects/iptables/index.html>. Accessed: 03/05/2022.
- [33] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.
- [34] Livinus Obiora Nweke. A survey of specification-based intrusion detection techniques for cyber-physical systems. *International Journal of Advanced Computer Science and Applications*, 12, 2021.
- [35] Yongheng Chen, Linhai Song, Xinyu Xing, Fengyuan Xu, and Wenfei Wu. Automated finite state machine extraction. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 9–15, 2019.
- [36] Merve Nur Cakir, Mehwish Saleemi, and Karl-Heinz Zimmermann. On the theory of stochastic automata. *arXiv preprint arXiv:2103.14423*, 2021.
- [37] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlenice Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. Contextlot: Towards providing contextual integrity to appified iot platforms. In *NDSS*, volume 2, pages 2–2. San Diego, 2017.
- [38] Connected Life with SmartThings. <https://www.samsung.com/us/smartthings/>. Accessed: 27/05/2022.
- [39] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. HAWatcher : Semantics-Aware Anomaly Detection for Appified Smart Homes. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4223–4240, 2021.
- [40] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. Pfirewall: Semantics-aware customizable data flow control for home automation systems. *arXiv preprint arXiv:1910.07987*, 2019.
- [41] General Data Protection Regulation. <https://gdpr-info.eu/n>. Accessed: 19/05/2022.
- [42] Wenbo Ding, Hongxin Hu, and Long Cheng. Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery. In *the 28th Network and Distributed System Security Symposium (NDSS 2021)*, 2021.
- [43] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

-
- [44] Biljana L Risteska Stojkoska and Kire V Trivodaliev. A review of internet of things for smart home: Challenges and solutions. *Journal of cleaner production*, 140:1454–1464, 2017.
- [45] Sotiris Ioannidis, Angelos D Keromytis, Steve M Bellovin, and Jonathan M Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199, 2000.
- [46] Smart light bulbs. <https://www.philips-hue.com/en-us/products/smart-light-bulbs>. Accessed: 25/05/2022.
- [47] Finite State Machine: Mealy State Machine and Moore State Machine. <https://www.elprocus.com/finite-state-machine-mealy-state-machine-and-moore-state-machine/>. Accessed: 28/05/2022.
- [48] Speech and Language Processing (3rd ed. draft). <https://web.stanford.edu/jurafsky/slp3/8.pdf>. Accessed: 28/05/2022.
- [49] Vagrant by HashiCorp. <https://www.vagrantup.com/>. Accessed: 14/05/2022.
- [50] Fedora/35-cloud-base. <https://app.vagrantup.com/fedora/boxes/35-cloud-base>. Accessed: 27/05/2022.
- [51] Eclipse Mosquitto An open source MQTT broker. <https://mosquitto.org/>. Accessed: 15/05/2022.
- [52] Eclipse Paho The Eclipse Foundation. <https://www.eclipse.org/paho/index.php?page=clients/python/index.php>. Accessed: 15/05/2022.
- [53] Aiocoap – The Python CoAP library. <https://github.com/chrysn/aiocoap>. Accessed: 15/05/2022.
- [54] The Transport Layer Security (TLS) Protocol Version 1.3. <https://www.rfc-editor.org/rfc/rfc8446>. Accessed: 25/05/2022.
- [55] SSL/TLS Offloading, Encryption, and Certificates with NGINX and NGINX Plus. <https://www.nginx.com/blog/nginx-ssl/>. Accessed: 25/05/2022.
- [56] Mark O’Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. Tls proxies: Friend or foe? In *Proceedings of the 2016 Internet Measurement Conference*, pages 551–557, 2016.
- [57] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. Packet-level signatures for smart home devices. In *Network and Distributed Systems Security (NDSS) Symposium*, volume 2020, 2020.
- [58] The Datagram Transport Layer Security (DTLS) Protocol. <https://www.rfc-editor.org/rfc/rfc9147.html>. Accessed: 22/05/2022.
- [59] What Is Token-Based Authentication? <https://www.okta.com/identity-101/what-is-token-based-authentication/>. Accessed: 22/05/2022.

-
- [60] Bruhadeshwar Bezawada, Maalvika Bachani, Jordan Peterson, Hossein Shirazi, Indrakshi Ray, and Indrajit Ray. Behavioral fingerprinting of iot devices. In *Proceedings of the 2018 workshop on attacks and solutions in hardware security*, pages 41–50, 2018.
 - [61] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.
 - [62] Hyunwoo Cho, Gary D Hachtel, Enrico Macii, Bernard Plessier, and Fabio Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1465–1478, 1996.
 - [63] Mihalis Yannakakis. Hierarchical state machines. In *IFIP International Conference on Theoretical Computer Science*, pages 315–330. Springer, 2000.

Appendix A

Input file syntax

A.1 Home presence example

```
1 {
2   "member":{
3     "any":{
4       "src":"0.0.0.0/0"
5     },
6     "broker":{
7       "src":"192.168.33.11;1883"
8     },
9     "motion_sensor":{
10      "src":"192.168.33.12",
11      "field":{
12        "motion":{
13          "type":"string",
14          "value":[
15            "true",
16            "false"
17          ],
18          "init":"false"
19        }
20      }
21    },
22    "heater":{
23      "src":"192.168.33.13"
24    },
25    "door_sensor":{
26      "src":"192.168.33.14",
27      "field":{
28        "presence":{
29          "type":"string",
30          "value":[
31            "true",
32            "false"
33          ],
34          "init":"false"
35        }
36      }
37    }
38  },
39  "relation":{
40    "trigger_heater":{
41      "subject":"th",
```

```

42         "broker": "broker",
43         "publisher": "any",
44         "subscriber": "heater"
45     },
46     "trigger_motion": {
47         "subject": "tm",
48         "broker": "broker",
49         "publisher": "motion_sensor",
50         "subscriber": "motion_sensor",
51         "constraints": [
52             {
53                 "field": "motion",
54                 "value": [
55                     "true",
56                     "false"
57                 ]
58             }
59         ]
60     },
61     "trigger_door": {
62         "subject": "td",
63         "broker": "broker",
64         "publisher": "door_sensor",
65         "subscriber": "door_sensor",
66         "constraints": [
67             {
68                 "field": "presence",
69                 "value": [
70                     "true",
71                     "false"
72                 ]
73             }
74         ]
75     }
76 },
77 "trigger": [
78     {
79         "condition": {
80             "field": {
81                 "motion_sensor": {
82                     "motion": "true"
83                 },
84                 "door_sensor": {
85                     "presence": "true"
86                 }
87             }
88         },
89         "action": {
90             "enable": "trigger_heater"

```

```

91     }
92   }
93 ]
94 }

```

A.2 Night example

```

1 {
2   "categorization":{
3     "night":{
4       "type":"time",
5       "value":[
6         "23:00",
7         "06:00"
8       ]
9     }
10  },
11  "member":{
12    "any":{
13      "src":"0.0.0.0/0"
14    },
15    "light":{
16      "src":"192.168.33.12"
17    },
18    "window":{
19      "src":"192.168.33.13"
20    },
21    "shutter":{
22      "src":"192.168.33.14"
23    }
24  },
25  "relation":{
26    "trigger_light":{
27      "subject":"tl",
28      "publisher":"any",
29      "subscriber":"light"
30    },
31    "trigger_window":{
32      "subject":"tw",
33      "publisher":"any",
34      "subscriber":"window"
35    },
36    "trigger_shutter":{
37      "subject":"ts",
38      "publisher":"any",
39      "subscriber":"shutter"
40    }
41  },

```

```

42     "trigger":[
43         {
44             "condition":{
45                 "time":[
46                     "night"
47                 ]
48             },
49             "action":{
50                 "disable":"trigger_light"
51             }
52         },
53         {
54             "condition":{
55                 "time":[
56                     "night"
57                 ]
58             },
59             "action":{
60                 "disable":"trigger_window"
61             }
62         },
63         {
64             "condition":{
65                 "time":[
66                     "night"
67                 ]
68             },
69             "action":{
70                 "disable":"trigger_shutter"
71             }
72         }
73     ]
74 }

```

A.3 Vacuum cleaner example

```

1  {
2      "member":{
3          "window_switch":{
4              "src":"192.168.33.12",
5              "field":{
6                  "status":{
7                      "type":"string",
8                      "value":[
9                          "on",
10                         "off"
11                     ],
12                     "init":"off"

```

```
13         }
14     },
15     "radio_switch":{
16         "src":"192.168.33.17",
17         "field":{
18             "status":{
19                 "type":"string",
20                 "value":[
21                     "on",
22                     "off"
23                 ],
24                 "init":"off"
25             }
26         }
27     },
28     "radio":{
29         "src":"192.168.33.13",
30         "field":{
31             "status":{
32                 "type":"string",
33                 "value":[
34                     "on",
35                     "off"
36                 ],
37                 "init":"off"
38             }
39         }
40     },
41     "window":{
42         "src":"192.168.33.14",
43         "field":{
44             "status":{
45                 "type":"string",
46                 "value":[
47                     "open",
48                     "closed"
49                 ],
50                 "init":"closed"
51             }
52         }
53     },
54     "motion_sensor":{
55         "src":"192.168.33.15",
56         "field":{
57             "motion":{
58                 "type":"string",
59                 "value":[
60                     "true",
61
```

```

62         "false"
63     ],
64     "init": "false"
65 }
66 }
67 }
68 },
69 "relation": {
70     "trigger_motion": {
71         "subject": "tm",
72         "publisher": "motion_sensor",
73         "subscriber": "motion_sensor",
74         "constraints": [
75             {
76                 "field": "motion",
77                 "value": [
78                     "true",
79                     "false"
80                 ]
81             }
82         ]
83     },
84     "trigger_window": {
85         "subject": "tw",
86         "publisher": "window_switch",
87         "subscriber": "window",
88         "constraints": [
89             {
90                 "field": "status",
91                 "value": [
92                     "on",
93                     "off"
94                 ]
95             }
96         ]
97     },
98     "trigger_radio": {
99         "subject": "tr",
100        "publisher": "radio_switch",
101        "subscriber": "radio",
102        "constraints": [
103            {
104                "field": "status",
105                "value": [
106                    "on",
107                    "off"
108                ]
109            }
110        ]

```

```

111     }
112 },
113 "trigger":[
114     {
115         "condition":{
116             "field":{
117                 "motion_sensor":{
118                     "motion":"true"
119                 },
120                 "radio":{
121                     "status":"on"
122                 }
123             }
124         },
125         "action":{
126             "enable":"trigger_window"
127         }
128     }
129 ],
130 "inference":[
131     {
132         "condition":{
133             "window_switch":{
134                 "status":"on"
135             }
136         },
137         "implication":{
138             "window":{
139                 "status":"open"
140             }
141         }
142     },
143     {
144         "condition":{
145             "window_switch":{
146                 "status":"off"
147             }
148         },
149         "implication":{
150             "window":{
151                 "status":"closed"
152             }
153         }
154     },
155     {
156         "condition":{
157             "radio_switch":{
158                 "status":"on"
159             }

```

```

160         },
161         "implication":{
162             "radio":{
163                 "status":"on"
164             }
165         }
166     },
167     {
168         "condition":{
169             "radio_switch":{
170                 "status":"off"
171             }
172         },
173         "implication":{
174             "radio":{
175                 "status":"off"
176             }
177         }
178     }
179 ],
180 "inconsistency":[
181     {
182         "window_switch":{
183             "status":"on"
184         },
185         "window":{
186             "status":"closed"
187         }
188     },
189     {
190         "window_switch":{
191             "status":"off"
192         },
193         "window":{
194             "status":"open"
195         }
196     },
197     {
198         "radio_switch":{
199             "status":"on"
200         },
201         "radio":{
202             "status":"off"
203         }
204     },
205     {
206         "radio_switch":{
207             "status":"off"
208         },

```

```
209         "radio":{
210             "status":"on"
211         }
212     }
213 ]
214 }
```

Appendix B

Build time

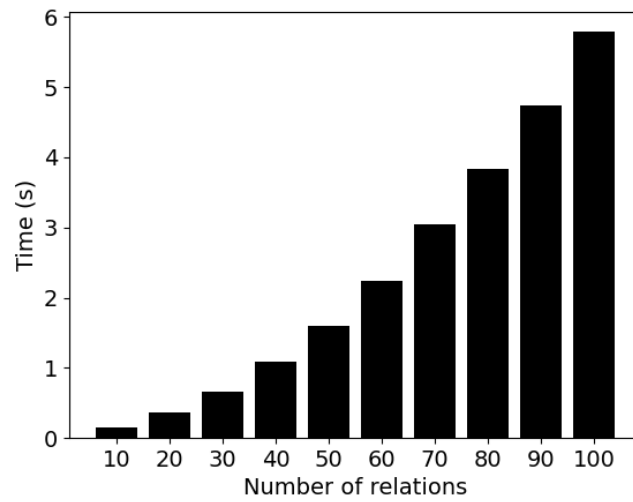


Figure 35: Build time to initialize Nftables for different numbers of relations.

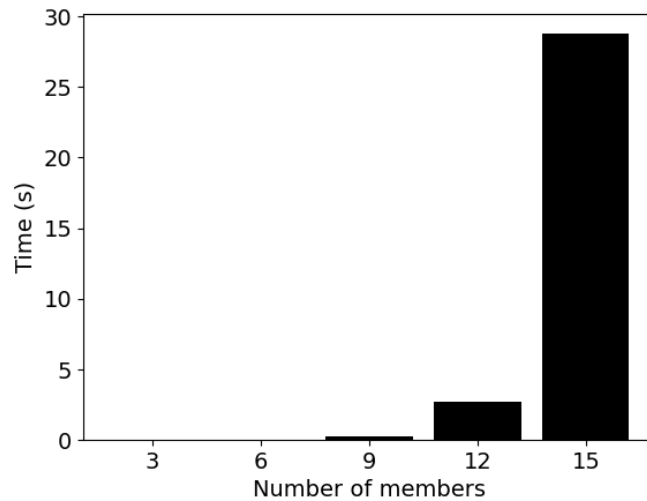


Figure 36: Build time to initialize the context for different numbers of members with a field of two values. An upper limit of 100 relations is set.

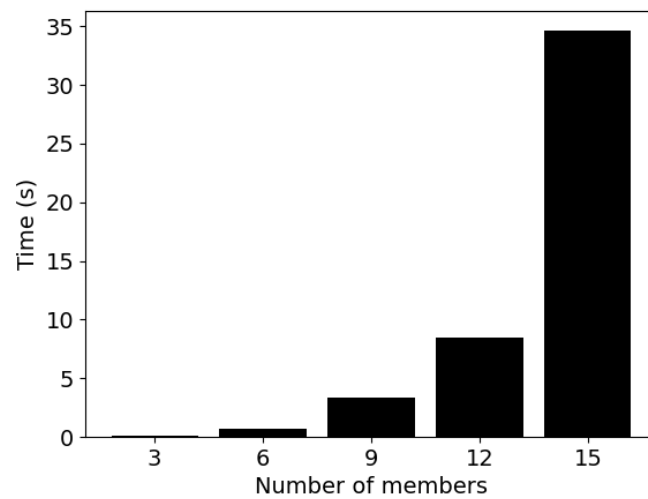


Figure 37: Total build time for different numbers of members with a field of two values. An upper limit of 100 relations is set.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

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