Implementation and Evaluation of the MQTT-TLS profile for Authentication and Authorization in Constrained Environments

Michael Michaelides

Fourth Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2020

Abstract

With the advent of the Internet of Things, many devices with little computational power and energy constraints are being connected to the internet at scale. For the sake of energy and computational efficiency, many of these devices incorporate limited security provisions which makes them vulnerable to online attacks, and threatens to compromise both their operation and user data privacy. In this project, we focus on securing MQTT, a popular publish/subscribe messaging protocol that has rudimentary support for security. We implement the Authentication and Authorization in Constrained Environments (ACE) MQTT-TLS profile specified by the Internet Engineering Task Force, to add an OAuth2 security layer on top of MQTT. We evaluate experimentally our secure ACE-MQTT implementation versus plain MQTT systems in realistic settings in both unconstrained and constrained environments. To assess the cost of security, we measure the CPU, memory, and network usage, as well as energy consumption. The results obtained confirm that the ACE solution matches the capabilities of moderately constrained devices, hence providing an affordable mechanism to secure MQTT systems.

Acknowledgements

I would like to express my gratitude to my two supervisors, Dr. Paul Patras and Dr. Cigdem Sengul, who proposed this project, provided invaluable feedback and contributed in various ways until the very end. Without them I would have not been able to get involved and eventually complete this project.

Table of Contents

Chapter 1

Introduction

Internet of Things (IoT) deployments usually consist of multiple resource-constrained devices that collect data and transmit them to cloud-based services over the public Internet. Such deployments can be used to improve home automation[26], provide better home care of patients[20], optimize and improve visibility of manufacturing procedures[34] and reduce costs in smart cities by streamlining procedures such as waste collection[1]. IoT devices usually have few computational resources and a small energy budget to keep the costs down, but they are still expected to last a significant amount of time without a battery replacement or a recharge. As a consequence, they tend to have limited or no support for features that are not part of the core functionality. Even security features that are usually a requirement for modern devices connected to the public internet, such as authentication, data integrity, and confidentiality, remain an afterthought.

Security is critical in the IoT and trading it in favor of resource efficiency can harm much more than just the application functionality[21]. Attackers are taking control of insecure IoT devices and use them to create large botnets able to launch distributed denial of service attacks[3]. Personal fitness trackers are abused to leak personal user information[10, 35]. In other cases, a network of insecure high wattage devices could be used to disrupt the power grid[31]. This is a real threat introduced by the scale of IoT and its lack of security standards.

Currently few IoT security standards exist to mitigate this risk, thus paving the way for proprietary implementations which are not publicly scrutinized. MQTT[23] is one such IoT protocol with rudimentary security support. It is a lightweight publish/subscribe messaging protocol by OASIS currently used in 42% of all IoT deployments[15]. Its standard form is largely insecure and exploitable, as demonstrated by Andy et al[2], thus making Transport Layer Security (TLS) strongly recommended. However, according to Shodan[30], at least 45,000 MQTT deployments worldwide are not using TLS and accept unauthenticated connections.

In an attempt to fill this gap, the Internet Engineering Task Force (IETF) Authentication and Authorization in Constrained Environments (ACE) MQTT-TLS profile[29] proposes an OAuth2-based authentication and authorization layer on top of MQTT.

The ACE MQTT-TLS profile (referred to as ”the profile” from now on) is currently lacking of a complete implementation and performance evaluation, which is an important part to defend its feasibility. In this project we implement the complete profile and we comprehensively evaluate our solution, in terms of resource utilization (CPU, memory and network input and output) and energy consumption. The results confirm the viability of deploying our ACE-MQTT implementation on moderately constrained devices. Finally we author a paper with our findings, which to the best of our knowledge is the first piece of work to evaluate the upcoming profile. In the next section we present our contributions in detail.

1.1 Contributions

Throughout the course of this project, the following milestones were achieved:

- Implementation of an ACE-MQTT compliant client extension with the following features:
  - Initiation using a configuration file
  - Registration with the Authorization Server (AS)
  - Access Token (AT) request prior to the authentication phase
  - Proof of Possession (PoP) via Message Authentication Codes
  - Version 5, version 3.1.1 and challenge authentication support
  - Transport Layer Security (TLS) configuration
- Implementation of an ACE-MQTT compliant broker extension with the following features:
  - Initiation using a configuration file
  - Access Token (AT) introspection with the Authorization Server (AS)
  - Version 3, version 5 and challenge authentication support
  - Publish and subscribe request authorization
  - Transport Layer Security (TLS) configuration
- Automation and orchestration of deployment through containerization and Docker Compose setups
- Contributions in the form of bug reports and fixes to the broker library we based our broker extension on
- Standard contingency and interoperability testing of our implementation against the profile specification
- Resource utilization evaluation, based on CPU, memory and network input and output traffic

- Energy efficiency evaluation, based on instantaneous power but also cumulative energy consumption during the authentication and authorization phase
- A real time resource monitoring and visualization solution to simplify future evaluations and comparisons

Apart from implementation and evaluation, we also wrote a research paper with the name *An Experimental Evaluation of MQTT Authentication and Authorization in IoT Environments* which was submitted to the ACM Transactions on Internet of Things journal[1] and is currently under review. The paper summarizes the design and implementation of the profile and also presents a detailed resource utilization and energy consumption overhead, similar to the one in chapter 6. The significance of this paper is that it is the first ever detailed performance analysis of the profile. First, the results make it easier to decide whether an existing insecure MQTT client could handle the additional overhead of ACE-MQTT and thus whether an insecure MQTT deployment could be upgraded to ACE-MQTT. Second, the results clearly identify which of the profile requirements are the most resource and energy expensive, thus it sets a clear path for future work to optimize them. Finally, as an upcoming IETF specification, and assuming that it will become a RFC, the profile will need to be thoroughly evaluated before being adopted and this paper makes the first step towards that path.

The original objective of this project was "*A working and tested implementation of an IoT authorization solution*", as dictated by the project proposal[2]. Based on the above, I claim that the project was successfully completed, given that the implementation covers all the hard requirements of the profile, along with most of the soft requirements as well. The solution was thoroughly tested and evaluated, as detailed in chapter 6.

1.2 Roadmap of the project

The rest of the report is organized as follows. In chapter 2 we discuss existing relevant work around MQTT and ACE. Then in chapter 3 we give an overview of the MQTT protocol and we briefly explain how the profile builds on top of it to achieve security guarantees. Chapter 4 presents the design of the three components of an ACE-MQTT domain; the Authorization Server (AS) is described in 4.1, the broker in 4.2 and the client in 4.3. Chapter 5 presents the implementation details of the major features implemented during the project; client initialization is at 5.1, AS discovery at 5.2, the authentication phase and authorization phase at 5.3 and 5.4 respectively, additional features are described at 5.5 and finally a deviation we had to make from the profile is at 5.6 . Chapter 6 presents a test plan and a comparative performance evaluation of our implementation; energy efficiency is at section 6.1, resource efficiency at 6.2 and standard contingency testing at 6.3. In chapter 7 we conclude by summarizing the project key points and indicating potential directions for future work.

[1]https://dl.acm.org/journal/tiot

[2]https://dpmt.inf.ed.ac.uk/ug4/project/3568

Chapter 2

Prior art

2.1 Security in the MQTT protocol

In its plain form, MQTT is largely insecure[18] but a lot of work is being done to improve it. A solution for protecting MQTT topics based on the Augmented Password-Authenticated Key Agreement protocol is proposed by Calabretta et al[8], yet no evidence of performance is given. Esfahani et al[14] propose a lightweight mutual authentication method suitable for MQTT, however their solution provides no means to authorize client actions. Ramos et al[19] propose a way to test an implementation against malformed input vulnerabilities through fuzzing, however their work is focused on testing and does not provide any security guarantees.

Token-based authentication in MQTT has gained a lot of focus recently. Bhawiyuga et al[6] implement token-based MQTT authentication in constrained devices, but their evaluation is limited to usability and response time. Collina et al[11] propose QEST, a RESTful MQTT broker, and explore the idea of incorporating OAuth. An implementation is not provided though, and evaluation is missing. Aimaschana et al[22] introduce an OAuth1a-based system with insecure communication channels which requires clients to generate new signatures on every request. This increases complexity and requires one extra round trip during authentication, which is resource-expensive.

The work of Fremantle et al is related to ours as their design uses OAuth2 tokens for authentication and authorization[16]. However, they are using a single embedded token and insecure communication channels which makes the solution vulnerable to replay attacks and eavesdropping, as the authors acknowledge. Many other solutions consider devices constrained enough to not support SSL/TLS where communication happens over insecure channels[16, 22, 8]. These are not complete solutions, since they provide no data confidentiality or integrity. Also, solutions that rely on tokens hard-coded in the device firmware[16, 22] are not very flexible, as the device firmware must be flashed in order to change the token if compromised or if the device is moved to a different authorization domain.

Lastly, to the best of our knowledge, there are no implementations that take advantage of MQTT version 5 features to provide enhanced security. We use them to provide useful features such as the discovery of the AS location and challenge-based authentication.

2.2 ACE

The IETF ACE work-group is working on other protocol specifications in addition to the MQTT-TLS profile. First, the profile is based on the ACE-OAuth framework[28] which specifies the public API and the functionality of the AS using the OAuth2.0 protocol. The Pub-Sub Profile for ACE[25] is similar to the MQTT-TLS profile; they both use an OAuth2.0 based AS to provide authentication and authorization for a messaging protocol based on a publish-subscribe architecture, however they differ on the messaging protocol used. The MQTT-TLS profile uses MQTT whereas the other one uses Constrained Application Protocol (CoAP). Finally, the Datagram Transport Layer Security (DTLS) Profile for ACE[17] defines a lightweight protocol to allow constrained servers to delegate management of authorization information to a non-constrained AS. In our case the broker is not constrained, and we are using TCP at the transport layer thus TLS replaces DTLS, which is based on UDP transport instead.

2.3 IoT security performance evaluation

We could not find many papers that evaluate overall energy consumption of IoT secure solutions. Baranauskas et al[4] evaluate the impact that different Quality of Service levels have on energy consumption of the MQTT protocol over TLS. This is helpful to determine the cost of reliability of message delivery, however they do not compare against a plain MQTT deployment thus it is difficult to make conclusions about the cost of TLS. Tae et al[9] research how to choose the TLS cipher suite to optimize for security level, residual energy and message length. This provides useful insight and can be used as an optimization technique to our solution as well, i.e. it is orthogonal with our efforts.

Chapter 3

Background

For the sake of completeness, in this chapter we outline the MQTT protocol and we briefly explain how the profile builds on top of it to provide security. Where appropriate, we cite to the MQTT specification or the profile for more details.

3.1 The MQTT protocol

MQTT, short for Message Queuing Telemetry Transport, is a client-server protocol developed by IBM in 1999 and made public under a royalty free license in 2010. It is a lightweight publish/subscribe messaging protocol that can be used for near real time communication between clients, which are usually IoT devices. MQTT has seen wide adoption due to its efficiency and simplicity and became the most commonly used IoT messaging protocol, after HTTP[15]. Unlike HTTP, MQTT does not have a secure version, even though it is widely adopted. It mainly relies on external protocols, such as TLS, to provide security.

MQTT uses persistent TCP connections, and two standardized versions exist, namely 3.1.1[23] and 5[24]. The protocol operation is governed by a central server, called broker, which is responsible for relaying messages between clients as shown in Figure 3.1. Clients connect to the broker using a CONNECT packet, optionally followed by an authentication phase. The broker responds with a CONNACK packet, accepting or refusing the request.

Clients transmit data to a certain audience by sending PUBLISH messages to the broker specifying a topic name. The broker then forwards the message to all the clients subscribed to that topic. A topic is a resource on the broker used by clients to exchange messages and it is represented by a string of hierarchical structure, such as *factoryA/-groundfloor/temperature*.

A client can subscribe to topics by sending a SUBSCRIBE message, specifying a topic filter. The topic filter may match multiple individual topics managed by the broker. The broker will then subscribe the client to each matching topic, and it will forward messages published to these topics. A topic filter is a string in the hierarchical form of

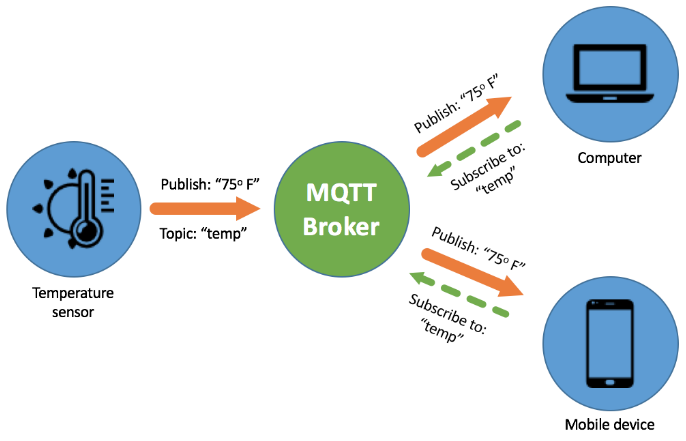

Figure 3.1: The different components and procedures of an MQTT system. Messages are delivered with publish messages through a central broker using topics subscriptions. Source: Axway developer blog[7]

a topic that allows the use of two wildcards, '+' and '#'. The wildcard '+' matches any subtopic found one level below the specified topic, whereas # matches any subtopic irrelevantly from how many levels below. For example, the filter *metric/+* matches the topic *metric/humidity* but not *metric/humidity/Edinburgh*, which is matched by the filter *metric/#*.

Finally, subscribe and publish requests also specify a Quality of Service (QoS) level. Three levels are possible:

- *At most once (QoS 0).* This is the lightest and fastest option and provides no more guarantees that the underlying TCP session does. The receiver of the message does not acknowledge it on the application layer. Thus, a subscriber with an unreliable connection which frequently drops might not receive published packets and a publisher might fail to successfully deliver its message to the broker.

- *At least once (QoS 1).* This level guarantees that the message will be delivered but duplicates may be sent as well if a confirmation does not arrive in time. There is additional overhead on both the sending and receiving side since caching of the packet is necessary and an acknowledgement is expected.

- *Exactly once (QoS 2).* This is the slowest and heaviest option but also the most reliable since it guarantees that the message will be delivered exactly once. Apart from caching the packet, the acknowledgement procedure consists of two round trips.

Currently MQTT is used in various domains. Facebook messenger uses it to achieve fast communication without draining the phone's battery life[33]. Cloud service platforms, such as the Google Cloud Platform, Microsoft Azure and Amazon Web Services use it to either transport telemetry or messages as part of their IoT solution[13, 5, 12].

3.1.1 MQTT security considerations

In both MQTT version 3.1.1 and 5 specifications, the security section is brief and non normative. It outlines the security that MQTT is capable to support and recommends adopting it, but no security measures are enabled by default.

Version 3.1.1 allows only username/password-based authentication through a *Username* and a *Password* field in the CONNECT packet. Thus, if any other authentication mechanism is implemented over 3.1.1 then it has to overload these fields.

Version 5 addresses a set of limitations of version 3.1.1, one of them being more support for security. Two new authentication fields are introduced in the CONNECT packet: *Authentication Method* and *Authentication Data*. With these, a client can signal to the broker the method being used to authenticate and it can use arbitrary format authentication data. These additions are flexible enough to allow the broker to support multiple authentication methods, varying from username/password-based to token-based and more. Additionally, version 5 supports a new AUTH packet which can be used to extend the authentication phase, such as to add a challenge, or to re-authenticate. However, the standard does not prescribe how these new features should be used nor does it recommended an authentication mechanism over them.

Given the level of security MQTT natively supports, the following concerns are raised:

1. **Authentication of clients by the broker.** The only concrete method natively provided by MQTT is username and password, while a custom authentication method can be implemented on top of version 5 if needed. All the problems of username and password authentication arise, such as weak passwords. Authentication is also optional and not enabled by default.

2. **Authentication of the broker by the clients.** There is no native support for this feature. Where TLS is being used, the certificate of the broker can be used to verify its identity. In version 5, implementing appropriate enhanced authentication methods can achieve this as well.

3. **Authorization of client requests by the broker.** The MQTT protocol does not describe authorization techniques. An authorization mechanism can be built based on the authentication data provided during the authentication phase.

4. **Integrity of messages.** There is no native support for this in the MQTT protocol. TLS can be used to provide integrity on the transport layer.

5. **Confidentiality of messages.** There is no native support in the MQTT protocol. One can either encrypt the application message payload manually or use TLS. Any data stored on disk should also be considered.

It is clear that the MQTT protocol heavily depends on the underlying protocols to provide security, mainly through TLS/SSL. While TLS/SSL can provide some guarantees, it is not complete since it can not always provide client authentication and client request authorization. Next we describe the profile which aims to do just that.

3.2 The MQTT-TLS profile for ACE

The profile[29] adds a complete layer of security on top of MQTT, covering data confidentiality, integrity and broker authentication using TLS, and client authentication and authorization using OAuth2 Access Tokens (ATs). In order to provide these guarantees, a set of additional requirements to a typical MQTT deployment are introduced. The components of an ACE-MQTT domain consist of the following:

- **ACE Authorization Server (AS)** responsible for registering clients, maintaining authorization policies for publishing and subscribing to topics and granting ATs to clients
- **ACE MQTT broker** responsible for authenticating clients and authorizing publish and subscribe requests
- **ACE MQTT client** that communicates with other clients while adhering to the new security requirements

All three entities communicate over pairwise secure channels, as shown in Figure 3.2. The clients and the broker communicate with the AS over HTTPS, and between them using MQTT over TLS. Both the broker and the AS use their TLS certificates to authenticate to the clients.

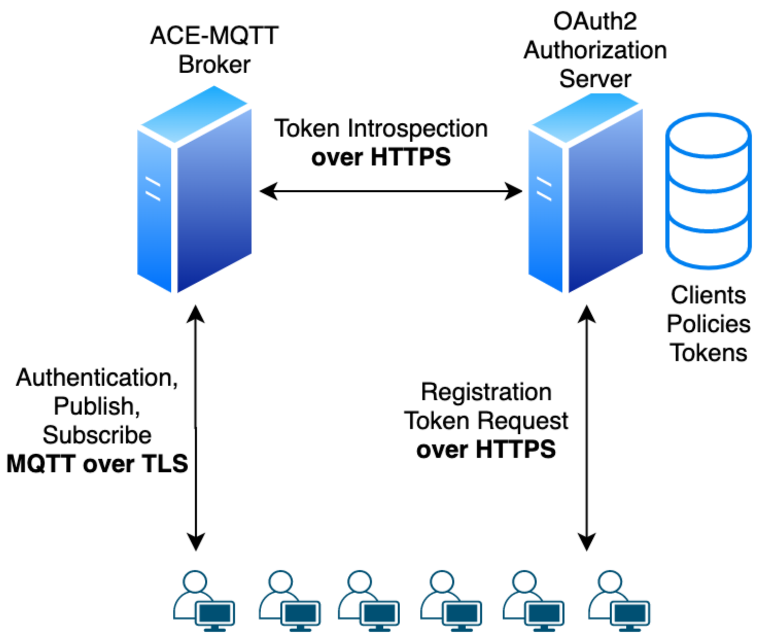
ACE-MQTT Broker
OAuth2 Authorization Server
Token Introspection **over HTTPS**
Clients Policies Tokens
Authentication, Publish, Subscribe **MQTT over TLS**
Registration Token Request **over HTTPS**

Figure 3.2: The different components of an ACE-MQTT system and the communication channels between them: AS and broker communicate over HTTPS; clients communicate with the broker using MQTT over TLS, and with the AS using HTTPS.

Clients do not necessarily have TLS certificates. They authenticate with the AS using a client id and a secret, obtained by registering to the AS. On the other hand, they authenticate with the broker using OAuth2 ATs which are issued by the AS and are accepted by the broker as a valid form of authentication. An AT is a string which is associated with additional data, such as the id of the client requesting it, an expiry date, a scope which specifies the permissions of the client and a Proof of Possession (PoP) key and algorithm necessary to prove ownership of the AT.

The ACE-MQTT broker needs to validate the AT presented by an authenticating client. To do this, the broker introspects it with the AS to check if it is valid and to obtain the associated data. If it is valid and not expired, the broker requests a PoP to ensure that the client is the legal owner of the AT, in order to prevent leaked or eavesdropped ATs from being used to impersonate clients. The default PoP method is through Message Authentication Code (MAC), where the client uses the symmetric PoP key to compute an authentication tag. Digital Signature PoP is also possible but only if the client has its own public key first. Section 2.1.3 of the profile details the AT validation phase for more details.

The profile defines different authentication methods for clients of different MQTT versions. Version 3.1.1 clients have to fill the username and password fields with the AT and PoP respectively, while version 5 clients can make use of the new Authentication Data field instead. Version 5 clients can also choose between simple or challenge-based PoP, whereas version 3.1.1 clients can only perform the simple method. In challenge-based, both the client and the broker contribute to create a nonce used for the PoP. In the simple version, the nonce created during the TLS handshake is used, thus the client can compute the PoP before sending the CONNECT packet. See Sections 2.1.2 and 2.1.4 of the profile[29] for more details of the authentication phase.

Upon successful authentication, the topics authorised to publish or subscribe to are determined by the scope associated with the client's AT. The scope is a space separated set of permissions, which follow a particular format, e.g., the scope "$publish\_topic1$ $subscribe\_topic2/\#$" allows publishing under *topic1* and subscribing under any subtopic below *topic2*. Note that the each scope entry defines a topic filter that can match multiple topics. The scope defines a white-list, thus any action not defined in a scope entry is not authorized by default.

To authorize a publish request, the broker needs to check the scope associated with the client's AT. This check is simply finding a scope entry that matches the topic in the publish packet. If the broker finds such entry, then it authorizes the request otherwise it rejects it. Next, before forwarding the message to each subscriber under that topic, the broker has to make sure that the subscriber AT is not expired. Authorization of publish messages is detailed in section 2.2 of the profile[29].

Subscribe request authorization may be more involved, because both the AT scope and the subscribe packet include topic filters that may contain wildcards. Therefore, for each topic filter in the request, the broker has to check that the AT scope contains an entry that is a super-set of that topic filter.

Being able to report authorization errors also depends on the MQTT version being used. Specifically, version 3.1.1 does not allow the broker to indicate a publish or subscribe authorization failure, nor does it allow sending a server-side disconnect. Thus, the broker drops the TCP connection if an unauthorized request is received. In contrast, version 5 provides better error reporting, allowing the broker to send a negative publish or subscribe acknowledgement, indicating an authorization failure. As such, the client can proceed to re-authenticate by obtaining a new AT and providing it to the broker, without reconnecting. This is resource efficient, since TLS session initiation is expensive as we'll see in chapter 6. Finally, a broker can also gracefully disconnect a version 5 client by sending a server-side disconnect if needed. Section 3.2 of the profile[29] details the authorization error handling.

Chapter 4

Design

In this section we present the design of the broker and the client extensions developed during this project. We also give a brief overview of the AS design, which was provided by Dr. Cigdem Sengul, one of the authors of the profile and my co-supervisor. The component architecture is shown in Figure 4.1 and described in detail in the next sections.

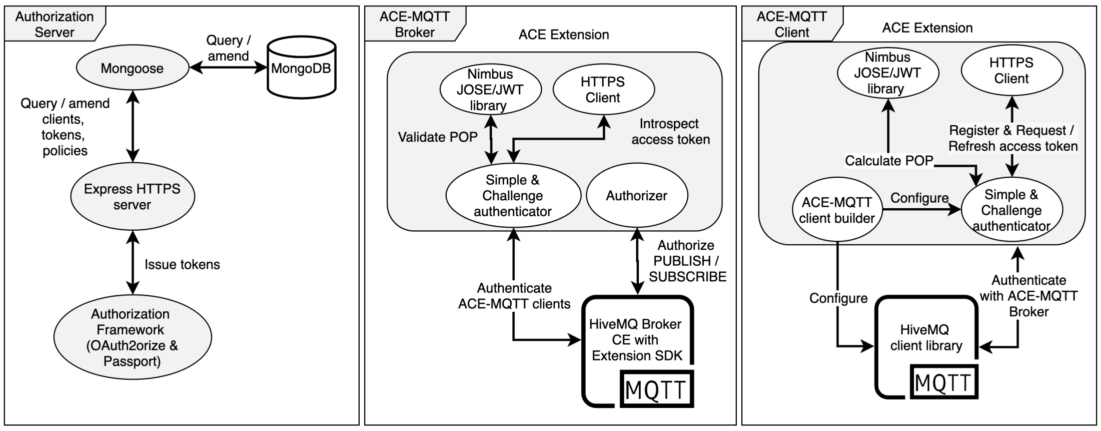

Figure 4.1: Architecture of ACE-MQTT a) AS b) broker c) client

4.1 Authorization Server

The Authorization Server (AS) is a Node.js[1] HTTPS server with OAuth2 support, which acts as a trusted third party between clients and the broker. The source code is publicly available on GitHub[2]. It provides the following public API to the ACE-MQTT broker and clients:

- *Client registration.* Unauthenticated endpoint that allows new clients to register. The client provides their unique name and URI and the AS responds with

[1]https://nodejs.org/en/

[2]https://github.com/ciseng/ace-mqtt-mosquitto

a unique client id and secret, which can then be used to access authenticated endpoints.

- *AT request.* Authenticated endpoint accessed by clients to request ATs. The client provides the desired scope and the AS responds with an AT along with associated data, as shown in Table 4.1. The AT is valid until the time specified by the Expiry Date field. The PoP key is a secret which provides message authentication during the authentication phase, to ensure that the token can not be used by anyone else apart from the client who requested it.
- *AT Introspection.* Authenticated endpoint accessed by the broker to introspect client ATs to obtain the associated data. The input is the AT and the output is identical to the AT request endpoint, shown in Table 4.1.
- *Policy management.* Authenticated endpoint accessed by resource owners to manage client authorization policies. New policies can be created and existing ones can be updated or deleted. A single policy dictates that a client can publish or subscribe to a set of topics defined by a topic filter. Resource owners are entities that can authoritatively decide permissions of access to an MQTT topic.

Field	**Interpretation**
Access Token	Credential of the client to the broker
Expiry Date	Token lifetime
Scope	Authorization permissions associated with this AT.
PoP Algorithm	MAC or Digital Signature algorithm
PoP Key	Symmetric or asymmetric key used with the PoP algorithm

Table 4.1: AS Token Response

The AS uses Express[3], a minimal and flexible web application framework, to support the HTTPS server. A MongoDB[4] database is used to store the **registered clients**, **policies** over each client's authorization permissions and active **Access Tokens**. The Mongoose object data modelling library[5] is the interface between the server and the database, defining the schema and performing the read and write operations. The OAuth2 authorization framework is implemented using OAuth2orize[6] and Passport[7].

[3]https://expressjs.com/
[4]https://www.mongodb.com/
[5]https://mongoosejs.com/
[6]https://github.com/jaredhanson/oauth2orize
[7]http://www.passportjs.org/

4.2 ACE MQTT broker

Our broker is a Java extension to the HiveMQ Broker Community Edition[8]. The source code is publicly available in GitHub[9]. The extension enhances the base broker with the following additional capabilities:

1. *TLS and HTTPS support*, to secure communications with ACE-MQTT clients and the AS, respectively
2. *Authentication* for version 3.1.1 and version 5 clients supporting both simple and challenge-based.
3. *PoP verification* of authenticating clients
4. *AT Introspection* to obtain AT associated data during the authentication phase
5. *Token caching and periodic validation*, to spot expiring ATs and disconnect/re-authenticate clients
6. *Authorization* of publish and subscribe requests
7. *AS Discovery*, to inform version 5 clients of the AS location

We decided to use the HiveMQ broker because of simplicity and existing support of desired features. At the beginning of the project, we researched different open source broker implementations that could be used for the project. We found out that both HiveMQ and Mosquitto[10] were suitable and could fulfill the requirements at that time. We compared the profile specification with each of these implementations and both had their pros and cons; HiveMQ has a simple extension Software Development Kit (SDK) that can be used to easily extend the broker without modifying the source code. It is written in Java which is already familiar to me thus we could have a head start with the project. Mosquitto is more popular and written in C, which is not as familiar to me. It does not support an extension SDK thus a plugin would have to be a patch over the library source code, which would make the plugin dependent on the broker implementation details and more complicated to understand and deploy. Since both of them were adequate back then, we chose HiveMQ for simplicity but new requirements that came along the way (described in section 5.6) made it more clear that Mosquitto might have been a better choice.

It is easy to incorporate our extension to the HiveMQ broker. The maven[11] build phase packages it into a zip archive that can be extracted under the extensions directory inside a HiveMQ broker installation. However, the HiveMQ broker also needs to be configured externally to support TLS. We automate this procedure by providing a container image that includes the whole setup and is ready to run out of the box. More details for the automation can be found in section 5.5.1.

[8]https://www.hivemq.com/developers/community/
[9]https://github.com/michaelg9/HiveMQACEextension
[10]https://mosquitto.org/
[11]https://maven.apache.org/

The major components of our extension are shown in Figure 4.2. The two authenticators, AuthenticatorV3 and AuthenticatorV5 handle the authentication of clients version 3.1.1 and 5 respectively. They inherit from the AceAuthenticator base class which provides common functionality such as requesting AT introspection through the HttpsClient class, validating PoPs through the MacCalculator class and recording authenticated clients and their ATs in the ClientRegistry class. MacCalculator verifies MAC PoPs using the Nimbus JOSE/JWT library[12]. ClientRegistry is queried by the authorization class AceAuthorizer to check client ATs for expiry and to enforce permissions of publishers and subscribers. It is also equipped with an instance of a PublishOutboundInterceptor class which checks the expiry date of the AT of subscribers before the broker forwards them a publish message.

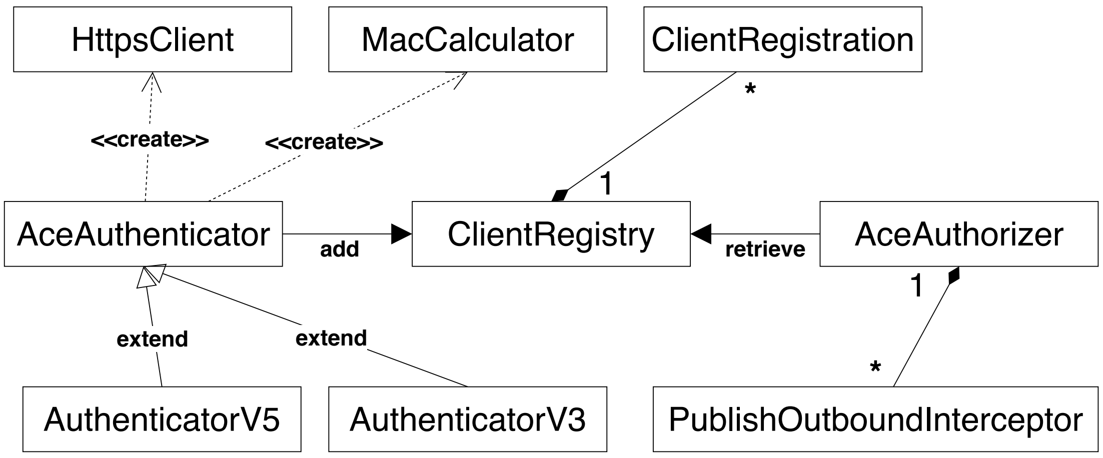

Figure 4.2: Major components of the ACE broker extension used for authentication (AceAuthenticator with sub-classes AuthenticatorV3 and AuthenticatorV5 and with dependencies HttpsClient and MacCalculator), authorization (AceAuthorizer and PublishOutboundAuthorizer) and AT tracking (ClientRegistry and ClientRegistration)

4.3 ACE MQTT client

Our client is a Java extension of the HiveMQ MQTT client[13]. The source code is publicly available in GitHub[14]. The extension enhances the HiveMQ client with the following additional capabilities:

1. *TLS and HTTPS protocol support* used to securely communicate with the broker and the AS respectively

2. *Client bootstrapping*. Support for initial client configuration via a config file and functionality to complete missing information, such as client registration to obtain client id and secret, or AS discovery to figure out the location of the AS

[12]https://connect2id.com/products/nimbus-jose-jwt
[13]https://github.com/hivemq/hivemq-mqtt-client
[14]https://github.com/michaelg9/HiveACEclient

3. *Simple public API* that allows the user to easily instantiate an ACE-MQTT client and to choose between different options, such as authentication type
4. *Proof of Possession* calculation
5. *Authentication with the broker* through the simple and challenge-based method
6. *AS discovery* to request the location of the AS from the broker, if needed
7. *Re-authentication* to renew an expired AT without terminating the MQTT session

We chose the HiveMQ client for its simplicity and feature support. The available options for a base library were limited because most implementations were lacking complete MQTT version 5 support by that time, which has been released recently. We identified HiveMQ and Paho[15] as potential options that could support all the profile requirements back then. The advantage of using HiveMQ was simplicity, since it allows extending it without modifying the source code. On the other hand, it is written in Java which is not usually supported by embedded devices. Paho is written in C, which is the language usually supported by embedded devices but it is not extensible. We chose HiveMQ for its simplicity and because our focus was not specifically on embedded devices but on moderately constrained devices instead that can support TLS and source languages apart from C. Looking back at it, Paho might have been a better choice since it could have covered new requirements that came along the way (more details in section 5.6).

The architectural structure of our extension is shown in Figure 4.3. The public API is provided by the ClientBuilder sub-classes, i.e. Ace3ClientBuilder and Ace5ClientBuilder which build version 3.1.1 and version 5 clients respectively. They extend from the base ClientBuilder which provides common functionality such as loading the initial configuration into the ClientConfig class, registering with the AS and requesting ATs using the HttpsClient. Furthermore, three different authentication mechanisms (DiscoveryAuthMechanism, SimpleV5AuthMechanism, ChallengeAuthMechanism) allow a version five client to perform AS discovery, simple and challenge authentication. Simple authentication for version 3.1.1 clients has built in support into the HiveMQ library thus there were no additional classes needed. Finally, an instance of the MacCalculator helps the different version 5 authenticators and the version 3 client builder to perform PoP.

Our extension is very simple to use and acts as a wrapper over the HiveMQ client. The two ClientBuilder sub-classes shown in Figure 4.4 make up the public API with two public methods each; *withAuthentication* and *connect*. The method *withAuthentication* allows the user to select the authentication method. A version 3.1.1 client has a choice between simple authentication, which is the default, and no authentication at all, to create a regular MQTT client. On the other hand, a version 5 client has the choice between simple, challenge-based and no authentication, by passing the appropriate AuthMechanism subclass as a parameter. Any additional actions or settings required to make the client comply with the profile, such as AS discovery, client registration, AT

[15]https://github.com/eclipse/paho.mqtt.c

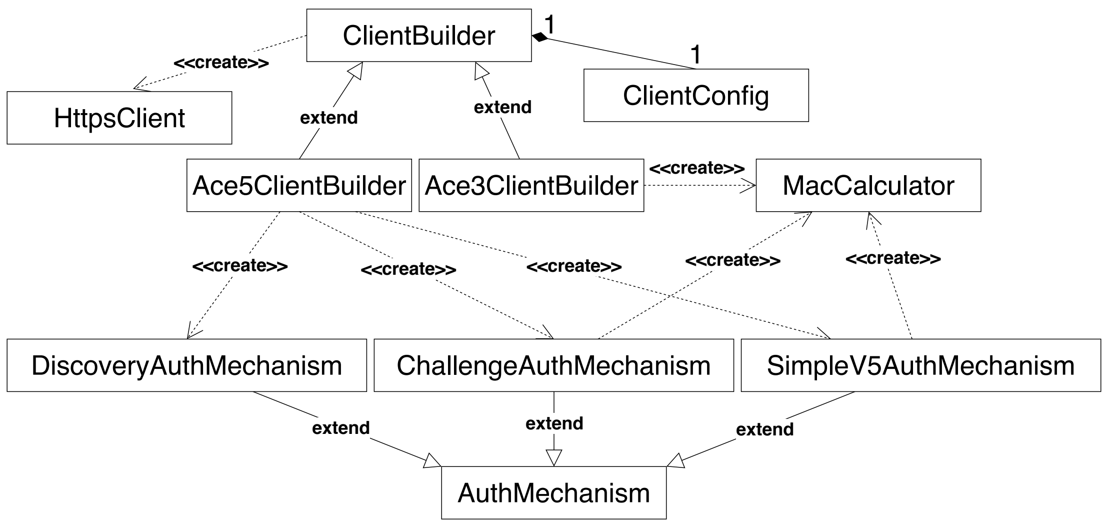

Figure 4.3: UML class diagram of client showing the base ClientBuilder class on top, along with its dependencies (HttpsClient, ClientConfig), the version 3 and 5 client builder sub-classes (Ace3ClientBuilder, Ace5ClientBuilder), the three version 5 authentication mechanisms (DiscoveryAuthMechanism, SimpleV5AuthMechanism, ChallengeAuthMechanism) and the MacCalculator used to calculate the PoP

request and transport protocol settings are performed internally by the ClientBuilder class, with no user interaction. Finally, the connect method sends an authentication request to the broker. If successful, a connected HiveMQ client instance of the appropriate version is returned with the only difference being that if the client attempts to perform an unauthorized action then it is refused by the broker.

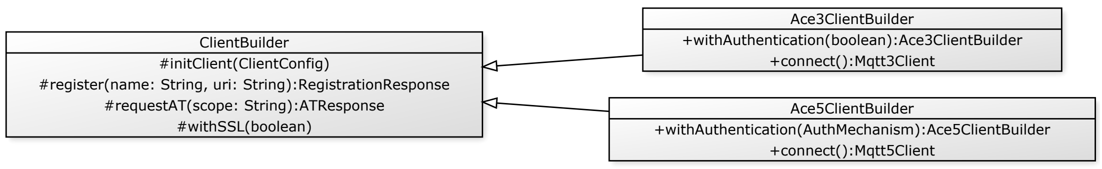
ClientBuilder
#initClient(ClientConfig)
#register(name: String, uri: String):RegistrationResponse
#requestAT(scope: String):ATResponse
#withSSL(boolean)
Ace3ClientBuilder
+withAuthentication(boolean):Ace3ClientBuilder
+connect():Mqtt3Client
Ace5ClientBuilder
+withAuthentication(AuthMechanism):Ace5ClientBuilder
+connect():Mqtt5Client

Figure 4.4: Client builder classes forming the public API of our client. On the left it is the base abstract ClientBuilder class that provides common functionality (load initial configuration, register client, request AT, configure transport protocol) and on the right the builders for version 3.1.1 and 5 clients. Public methods are indicated with the plus symbol while private methods with the hash

Chapter 5

Implementation

In this section we describe the different features implemented during the project, emphasizing on the client and broker side. As noted before, the AS implementation was provided by my external supervisor, Dr. Cigdem Sengul, and only minor modifications were made to it.

5.1 Client initialization and registration

A client instance requires initial configuration before it can execute. The client takes a single required command line argument specifying the location of a Java properties configuration file. The configuration file contains key-value pairs, with the keys being parameters shown in Table 5.1.

Parameter	**Required**
Broker IP address	Yes
Broker port	Default if missing
AS IP address	Yes, unless discovery is possible
AS port	Default if missing
TLS key & trust store location	Yes, if TLS is used
Transport Protocol	Defaults to TLS
Client Username	Yes, unless already registered
Client URI	Yes, unless already registered
Client ID	Unless no Username and URI is provided
Client Secret	Unless no Username and URI is provided
Scope	Yes

Table 5.1: Client initial configuration parameters

Upon execution, the ClientBuilder class parses the configuration file into an instance of a ClientConfig and performs initialization steps. First, it looks up the AS IP address in the configuration. If found then it checks if the client is registered, i.e. if the configuration includes a Client ID and Client Secret. If that is not the case, then the

HttpsClient sends a HTTPS POST request to the AS registration endpoint, providing the Client Username and Client URI obtained from the configuration. Upon successful registration, the client retrieves the Client ID and Client Secret and persists them in the configuration file in order to avoid the registration phase next time. Then, the ClientBuilder class configures the transport layer of the MQTT session according to the Transport Protocol configuration parameter; the default value is ”TLS”, according to the profile, however the user may set it to ”TCP” instead if the client will not be using ACE. Finally, if the AS IP address is not found, the ClientBuilder proceeds to discover it if possible. The discovery procedure is described next.

5.2 Authorization Server discovery

The client needs the DNS name or the IP address of the AS to register and request ATs. If not found in the configuration file, a version 5 client can request it from the broker, thus simplifying the initial configuration and allowing a non-static AS location. If ClientBuilder does not find the AS IP address parameter in the configuration, it will create a new version 5 client instance with an instance of DiscoveryAuthMechanism passed to the withAuthentication method. The DiscoveryAuthMechanism is responsible for forming a CONNECT packet with empty Authentication Data and an ”ace” Authentication Method, which signals a discovery request. Then the CONNECT packet is sent and the client receives back a CONNACK response refusing the request but containing the AS absolute URI as a user property field in the packet. Finally the ClientBuilder persists the location in the configuration file to avoid this procedure next time.

5.3 Authentication

In this section we explain the authentication phase. The ClientBuilder sub-classes allow the user to select the authentication method through the *withAuthentication* method. When the connect method is called, the client goes through the following steps:

1. Access Token (AT) request
2. Client authentication request
3. Access Token (AT) introspection
4. Proof of Possession (PoP)
5. Authentication request response

These steps are shown in Figure 5.1 and Figure 5.2 for simple and challenge based authentication. In the next sections we see each of these in more detail.

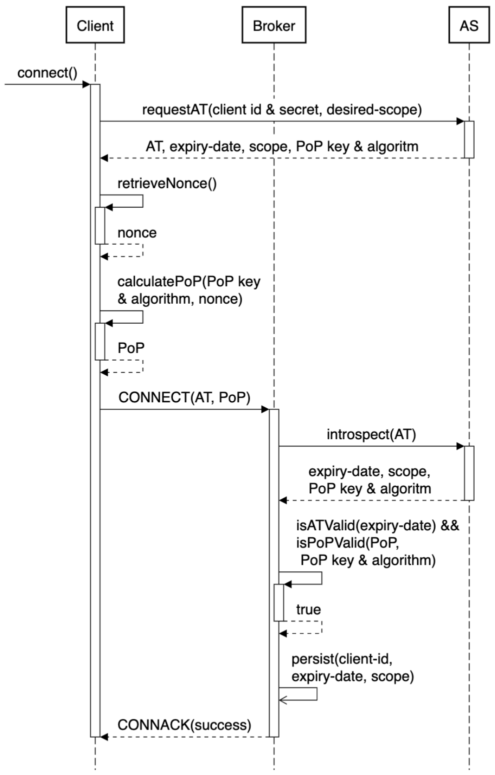
Client
Broker
AS
connect()
requestAT(client id & secret, desired-scope)
AT, expiry-date, scope, PoP key & algoritm
retrieveNonce()
nonce
calculatePoP(PoP key & algorithm, nonce)
PoP
CONNECT(AT, PoP)
introspect(AT)
expiry-date, scope, PoP key & algoritm
isATValid(expiry-date) && isPoPValid(PoP, PoP key & algorithm)
true
persist(client-id, expiry-date, scope)
CONNACK(success)

Figure 5.1: Simple authentication showing the AT request, authentication request, AT introspection, PoP validation and finally the authentication response

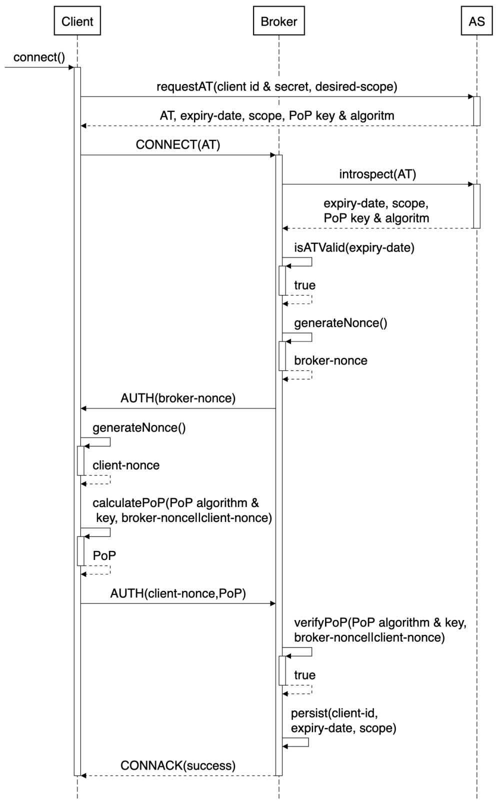
Client
Broker
AS
connect()
requestAT(client id & secret, desired-scope)
AT, expiry-date, scope, PoP key & algoritm
CONNECT(AT)
introspect(AT)
expiry-date, scope, PoP key & algoritm
isATValid(expiry-date)
true
generateNonce()
broker-nonce
AUTH(broker-nonce)
generateNonce()
client-nonce
calculatePoP(PoP algorithm & key, broker-nonce||client-nonce)
PoP
AUTH(client-nonce,PoP)
verifyPoP(PoP algorithm & key, broker-nonce||client-nonce)
true
persist(client-id, expiry-date, scope)
CONNACK(success)

Figure 5.2: Challenge authentication showing the AT request, authentication request, AT introspection, the challenge PoP based on a nonce created jointly by the broker and the client and finally the authentication response

5.3.1 Access Token request

First, the client needs to obtain an AT from the AS. The ClientBuilder class creates an instance of a HttpsClient and sends a POST request to the AT request endpoint. The parameters of the request are as follows:

- A bearer authorization header. The client authenticates with the AS through a bearer token containing the client id and secret.

- A JSON body containing the requested scope, retrieved from the client configuration.

The AS verifies the client credentials and permissions and then it issues an AT along with associated data (see Table 4.1). Then the ClientBuilder initializes the appropriate AceAuthenticator subclass, according to the user choice of authentication. The AS response is forwarded to the authenticator, which will be used during the authentication phase described next.

5.3.2 Client authentication request

Given a valid AT, the client creates and sends an authentication request to the broker via a CONNECT packet. The ClientBuilder and AceAuthenticator create the CONNECT packet whose format depends on MQTT version and authentication type. The AceAuthenticator is responsible for populating the authentication related fields, and the builder for everything else. Version 3.1.1 connect packets are an exception to this; the whole packet is created by the builder which has enough built in support by the HiveMQ client library.

Simple authentication is supported by both MQTT version 3.1.1 and 5 clients and it takes a single round trip to complete, like a regular MQTT connection request. For version 3.1.1 clients, ClientBuilder puts the AT inside the Username field and the PoP inside the Password field, as shown in Table 5.2. Additionally, the Clean Session flag is set to true as required by the profile. For version 5 CONNECT packets, the AceAuthenticator puts both the AT and the PoP inside the Authentication Data field, as length prefixed binary values. The Authentication Method field is set to 'ace', to indicate the protocol the client is authenticating against. Finally, the ClientBuilder leaves the username and password fields empty and sets the Clean Session Flag to true and the Session Expiry Interval to 0, as shown in Table 5.3.

Field	Value
Username	Access Token
Password	PoP
Clean Session Flag	True

Table 5.2: Simple authentication version 3 connect packet

Field	Value
MQTT Version	5
Authentication Method	ace
Authentication Data	AT,PoP
Username	empty
Password	empty
Session Expiry Interval	0
Clean Session Flat	True

Table 5.3: Simple authentication version 5 connect packet

Challenge authentication is supported only by version 5 clients and it takes an additional round trip to complete. The CONNECT packet is identical to the case of simple authentication for version 5, apart from that the Authentication Data field does not contain the PoP, which is exchanged in the second round trip.

5.3.3 Connect packet validation and Access Token introspection

Upon receiving the authentication request, the broker has to validate the format of the CONNECT packet and check that the AT is valid. The request is forwarded to an ACE extension authenticator, either AuthenticatorV3 or AuthenticatorV5, according to the client version. Since the CONNECT packet does not contain the AT associated data, the authenticator introspects it with the AS to retrieve them.

For a version 5 CONNECT packet, AuthenticatorV5 goes through the following procedure to validate a version 5 CONNECT packet and introspect the AT:

1. Ensure that the Authentication Method is set to 'ace'. If not, the authenticator ignores the packet and takes no decisive action. The HiveMQ broker forwards the packet to other authenticators if they exist. If no authenticator handles the request, the broker refuses the request with a CONNACK packet with reason code BAD AUTHENTICATION METHOD. This allows the broker to provide different authentication mechanisms.
2. Extract the AT from the Authentication Data field as a UTF-8 encoded string. If it is missing then the request is treated as an AS discovery request described in section 5.2 and the authenticator refuses the request replying back with the location of the AS.
3. Introspect the AT using the HttpsClient class to obtain AT associated data (see Table 4.1). If the provided AT is invalid, then the introspection request fails and the authentication request is refused with error code NOT AUTHORIZED.
4. Check the AT expiry date and refuse the request with error code NOT AUTHORIZED if expired.

The procedure is similar for a version 3.1.1 request. Step 1 is skipped since there is no Authentication Method field, thus the authenticator assumes that the client is authenticating using ACE. This does not allow the broker to support more than one authentication methods for version 3.1.1 clients. In step 2, the AT is retrieved from the Username field instead and the rest of the procedure is the same.

If the authentication phase reached this far successfully, then the broker continues to the PoP phase described next.

5.3.4 Proof of Possession

After successful AT validation, the broker makes sure that the client is the rightful owner of that token through a Proof of Possession (PoP). The PoP procedure depends on the type of authentication being used, i.e. either simple or challenge-based. A challenge-based PoP is performed as follows:

1. The broker AuthenticatorV5 initiates the PoP procedure as follows:
   (a) It generates a secure cryptographic nonce using the Java SecureRandom API and caches it so it can be retrieved after the client response arrives.
   (b) It creates an AUTH packet with the broker nonce contribution in the Authentication Data field, the string 'ace' in the Authentication Method field and the value CONTINUE AUTHENTICATION as Reason Code. The AUTH packet is sent to the client.
2. The client ChallengeAuthMechanism receives the AUTH packet and proceeds to perform PoP as follows:

   (a) ChallengeAuthMechanism generates the client nonce contribution using the SecureRandom Java API and concatenates it with the received nonce to form the final nonce

   (b) ChallengeAuthMechanism creates an instance of MacCalculator and passes the final nonce and the PoP algorithm and key as parameters

   (c) MacCalculator uses the Nimbus library to calculate the PoP by instantiating a JWSSigner with the algorithm and key and signing the final nonce with it

   (d) MacCalculator returns the PoP to the ChallengeAuthMechanism which creates an AUTH packet with Authentication Data field set to the client nonce contribution and the PoP as a length prefixed binary array, the Reason Code field to CONTINUE AUTHENTICATION and the Authentication Method field to the string 'ace'

   (e) The ChallengeAuthMechanism sends the AUTH packet to the broker

3. The broker AuthenticatorV5 receives the AUTH packet and validates the PoP as follows:

   (a) It extracts the client nonce contribution and the PoP from the packet

   (b) It concatenates the client nonce contribution with the cached broker nonce contribution to form the final nonce

   (c) It creates an instance of MacCalculator and parameterizes it with the PoP key and algorithm and the final nonce

   (d) MacCalculator calculates the expected PoP following the same procedure as at the client side

   (e) The client PoP is validated if it is identical to the expected one.

4. If the client PoP is not validated, the broker AuthenticatorV5 refuses the request with a CONNACK with error code NOT AUTHORIZED, otherwise, the client is authenticated as shown in the next section.

For simple authentication, the process is simpler and consists of a single round trip, rather than two, because the client defines the nonce. The client calculates the PoP before sending the CONNECT packet but after receiving the AT from the AS. The contents of the CONNECT packet are used as a nonce instead. The PoP is calculated using the MacCalculator in the same way as in challenge authentication. When the broker receives the authentication request and finds the PoP already included in the packet, it proceeds to validate it the same way as it does in challenge authentication. We note that how we perform PoP during simple authentication is a deviation from the latest profile specification. Instead of the CONNECT packet contents, the nonce should have been the TLS handshake client nonce. This deviation is explained in section 5.6.

We also note that Digital Signature PoP is not supported by our implementation. This would require extending the AS to support binding an AT to a client public key and it would also require that clients had their own SSL/TLS certificates. This increases the complexity and overhead and it is not the default way to perform PoP thus, due to time constraints we left this as part of future work.

5.3.5 Authentication request response

If the AT and PoP are valid, the broker authenticates a client by sending a CONNACK packet with a SUCCESS reason code. Additionally, it has to cache the client AT and its associated data in order to authorize future publish and subscribe requests from the client.

First the authenticator registers the client AT and its associated data in the ClientRegistry. The ClientRegistry holds an in memory map with key being the client id and value being a ClientRegistration class instance holding the client AT and its associated data. The map is used to periodically check AT expiry.

Second, the scope of the client is enforced at this stage. The authenticator uses the HiveMQ native authorization framework to create an Access Control List (ACL) which represents the scope of the client AT during run-time. The ACL is a white-list, which means any request not covered by a rule in this list is not authorized. Each rule in the ACL covers a scope entry by defining a topic filter and an associated action (subscribe or publish).

5.4 Client Authorization

Client authorization requires the broker to cover the following requirements, according to the profile; 1) ensure clients can only access topics defined in their scope and only for the allowed action (publish or subscribe); 2) check AT expiration for each publish or subscribe request; 3) provide error reporting when the AT is expired or when the request is not authorized. Next we describe how each of these features is implemented in the broker extension.

The scope of the client is enforced with the help of the HiveMQ native authorization framework as described in subsection 5.3.5. We implement the scope using ACLs during the authentication phase. To authorize publish requests, the topic name in the packet needs to match with the topic filter of at least one rule in the ACL that permits publishing. This is shown in Figure 5.3, via the call to *isTopicInScope* method. On the other hand, to authorize subscribe requests, each topic filter in the request is compared to all the rules in the ACL until a match is found. A match here means that there is a rule with a topic filter of equal or broader scope than the filter in the request. For example, a rule with a topic filter *metric/humidity/#* authorizes a request with a topic filter *metric/humidity/+*. After all the requested topic filters are checked, the broker authorizes the subscription request only for the topic filters that were authorized and refuses the rest. An acknowledgement is sent to the client for each request filter, indicating success or failure. These checks are handled natively by the HiveMQ broker library, given the ACL we create after authentication.

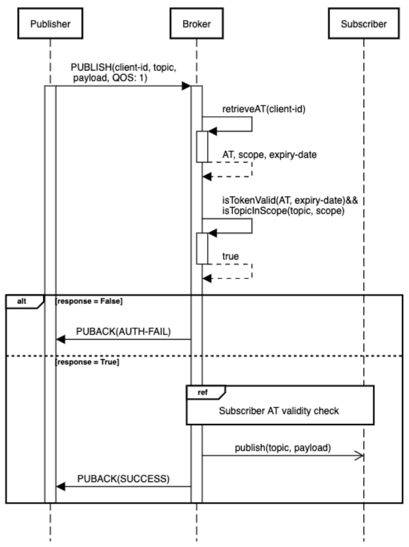

Figure 5.3: Publish message authorization

Checking for an expired AT is done by the AceAuthorizer class. The AceAuthorizer is executed before the ACL is applied and it refuses authorization if the AT is expired. When a publish or subscribe request is received, the AceAuthorizer retrieves the client AT from the ClientRegistry and then it checks that the expiry date is in the future. If not, then the request is refused and AceAuthorizer either drops the connection (for version 3.1.1 clients), or sends a server side disconnect message or a negative acknowledgement (version 5 clients). If it is in the future, then the AT is valid and AceAuthorizer lets the request be handled according to the ACL rules.

Finally, before sending a publish message to a subscriber, the subscriber's ATs must be validated. This happens each time a publish message arrives, in order to ensure that a subscriber with an expired AT does not receive the message. To check this, AceAuthorizer has an instance of an OutboundPublishInterceptor which is invoked every time the broker sends a publish message to a subscriber. The interceptor checks the validity of the subscriber's AT in the same way as described above and lets the packet go through if found valid. Otherwise, it drops the packet and disconnects the client, either by dropping the TCP connection, for a version 3.1.1 client, or by sending a server side disconnect, for a version 5 client.

5.5 Additional features

In this section we describe additional work that we made during this project but it either did not make it into the final version or it is a side feature which was not explained anywhere else.

5.5.1 Automation and monitoring

We automated the deployment of all the server side components, to improve usability and decrease management effort. When we started working on the project, the components had to be set up manually and in the correct order; first, the MongoDB database instance should be created, then the AS should be configured with the location of the database and then launched and finally the broker should be configured with the location of the AS and then launched. This process was time consuming thus we automated it using a Docker-Compose setup that deploys the system automatically. For this, we used the official Docker image of the MongoDB database. We modified slightly the AS implementation to receive the database location through command line arguments rather than a hard-coded location, which was necessary for the setup to work. Then we created a Docker image of the AS implementation that accepts the location of the MongoDB instance through command line arguments. Finally, we created an ACE-MQTT broker container image by building on top of the image of the HiveMQ CE broker and providing the essential configurations for TLS and embedding our extension. At the end, we created a Docker compose setup that puts all the components in the same network to allow inter-communication between them. Deploying our setup is a matter of a single command.

Furthermore, we improved visibility into the system by providing a real time resource utilization monitoring platform, to allow us to collect data about how each component is coping during different tasks. The setup monitors each component in real time and collects resource utilization statistics such as CPU, memory utilization and network inbound and outbound traffic. Finally, the information is presented in a real time updating dashboard. The monitoring platform is also easily deployed using a Docker Compose setup and it consists of the following services:

- Google Container Advisor[1] to query the CPU, memory and network input and output utilization from each container and make them available for querying
- Prometheus node exporter[2] to query the Google Container Advisor and store the data in Prometheus time series database
- Prometheus[3] time series database to store the readings according to the time they were observed and make them available for querying.
- Grafana[4] visualization platform to visualize all the real time or history metrics. We created our own dashboard, shown in shown in Figure 5.4, to present each piece of information in an appropriate format.

This setup can easily be used to evaluate different implementation versions, to quantify optimizations and to compare our implementation's efficiency against others.

[1]https://github.com/google/cadvisor
[2]https://github.com/prometheus/node_exporter
[3]https://prometheus.io/
[4]https://grafana.com/

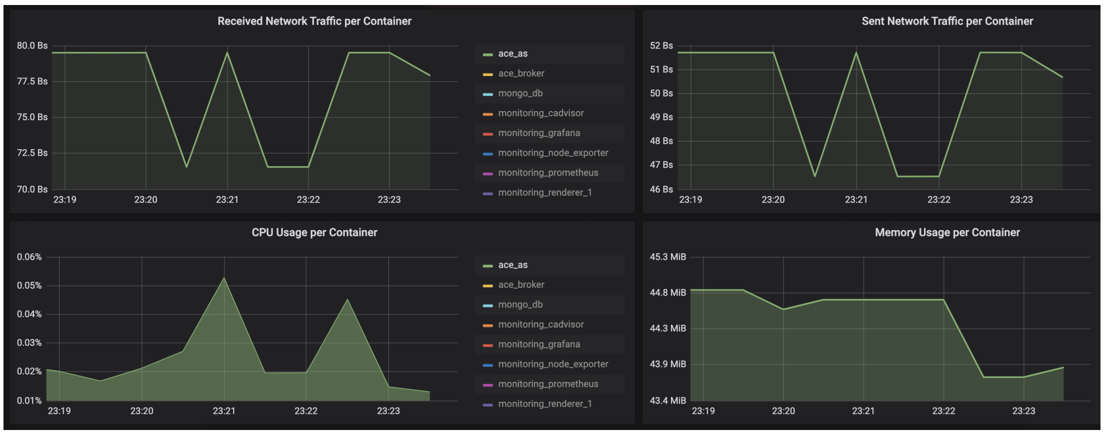

Figure 5.4: Our Grafana visualization dashboard, showing resource utilization for the AS; from top left to bottom right: a) Received Network traffic b) Sent Network traffic d) CPU usage f) Memory utilization

5.5.2 HiveMQ broker modifications

MQTT version 5 is relatively new thus the HiveMQ broker extension SDK did not fully support it when we started working on the project. Specifically, there was no support for extended authentication using AUTH packets, thus we could not easily implement challenge-based authentication. We got in touch with the HiveMQ team to find out that this feature was being worked on but we did not get a clear timeline of when it would be released, thus we worked around it. We created our own fork of the HiveMQ broker[5] and modified the source code of both the base library and the SDK in order to add the missing support. Then we based our ACE extension on top of this modified SDK. Fortunately, there was partial support in the HiveMQ broker implementation to handle extended authentication but it had not been exposed to the SDK, thus we did not have to implement the whole feature from scratch. Specifically, we had to do the following changes (full details can be found in the corresponding GitHub Pull Request[6]):

- Modify the HiveMQ broker to forward AUTH packets to and from the extension SDK authenticators. To do this, we figured out that a Netty[7] ChannelHandler class responsible for handling AUTH packets was not yet implemented thus we provided our own implementation of it.
- Extend the SDK API to allow extensions to make use of our new feature without further modifications to the base library. We added support for an ExtendedAuthenticator class which could send and receive AUTH packets.

Even though our implementation was working, having a modified base library was increasing the complexity of our components because we could not use the official release of the broker, instead we had to package the whole base library along with the extension. Fortunately, HiveMQ version 4.3 was released recently with support for

[5]https://github.com/michaelg9/hivemq-community-edition
[6]https://github.com/michaelg9/hivemq-community-edition/pull/2
[7]https://netty.io/

extended authentication. We easily migrated our code to the new version because the interface we created was not much different than the one officially released. Apart from source code modifications, we also contributed to the HiveMQ broker as explained in the next section.

5.5.3 HiveMQ broker contributions

Throughout the course of the project we identified three bugs in the HiveMQ broker and raised them up with the official development team. The first one was a minor bug in the authentication error reporting mechanism, providing the client with the wrong error message why the request was refused. We raised an issue[8] and a bugfix[9] that has been accepted. Then another one was found where the broker was not following the MQTT version 5 specification during the authentication phase. The specification requires the broker to respond to a CONNECT packet with a CONNACK with identical *Authentication Method* value, however the HiveMQ broker was leaving the field empty. This resulted in the client refusing connection establishment, even if authentication was successful. We raised a bug report[10] for this issue but since the HiveMQ implementation was not fully supporting version 5 yet, they did not take our bug fix[11] we implemented for our own fork of the broker. Finally, we reported a severe bug[12] where a client using TLS was always disconnected seconds after successfully authenticating. We identified and included the root cause analysis in our report; a timer was used to terminate unfinished stale connection requests but it was terminating all sessions indiscriminately because it was not being stopped after the client was successfully authenticated. We proposed a solution but the development team informed us that they are already working on a fix.

5.6 TLS Exporter

In this section we describe why our implementation of simple authentication PoP deviates from the profile with respect to the choice of nonce. The profile is a draft which was actively being modified during the project. The original proposal was to use the CONNECT packet as a nonce which is how we implemented it. However, the IETF review team noted that the CONNECT packet might not contain enough randomness and they suggested using a nonce generated during the TLS handshake instead, which could be retrieved using a TLS exporter as described in RFC-5705[27]. Unfortunately, unlike the OpenSSL[13] library, Java does not support this feature. The OpenSSL library is used by lower level broker and client implementations, such as the Mosquitto broker

[8] https://github.com/hivemq/hivemq-community-edition/issues/119

[9] https://github.com/hivemq/hivemq-community-edition/pull/120

[10] https://github.com/hivemq/hivemq-community-edition/issues/110

[11] https://github.com/michaelg9/hivemq-community-edition/pull/1

[12] https://github.com/hivemq/hivemq-community-edition/issues/149

[13] https://www.openssl.org/docs/man1.1.1/man3/SSL_export_keying_material.html

and the Paho client, however we could not switch to them halfway through project. Thus we searched for alternative solutions.

One potential solution we found was BouncyCastle[14] which is an extended cryptography API supporting Java. BouncyCastle supports a TLS Exporter through the TlsContext API[15] and specifically through the exportKeyingMaterial method. Unfortunately we could not easily implement BouncyCastle in our project because all the cryptographic functions are handled internally by the HiveMQ broker implementation which uses the native Java cryptographic API. We decided not to attempt to heavily modify the broker source code to migrate to BouncyCastle because that would require a lot of changes and effort for a far from ideal solution which would be prone to bugs and would make our extensions dependent on the modified version of the base library.

Another potential solution we found involved using reflection to access attributes not provided by the Java cryptography public API. Specifically, TLS keying material such as the master secret and nonces are stored in memory in private fields by the SSLSessionImpl implementation class of the SSLSession[16] Java API. We found a StackOverflow post[17] describing a way to use reflection to access this information. However, even this way, we still had to modify the HiveMQ source code to obtain the instance of the MQTT SSL session which is internally handled by HiveMQ and not publicly accessible. We decided not to pursue this solution either because accessing private fields through reflection meant that our solution would be dependent on the Java version and platform.

Given these options, we decided that putting time to achieve a bad quality workaround could prevent us from achieving higher priority goals, such as comprehensively evaluating our solution. We decided to stay with the solution that was originally stated in the profile which is less secure but it is a clean solution and it bears a very similar overhead to the TLS exporter idea, thus allowing us to evaluate the solution and argue about the overhead of the profile.

[14]https://www.bouncycastle.org/
[15]https://www.bouncycastle.org/docs/tlsdocs1.5on/org/bouncycastle/tls/TlsContext.html
[16]https://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLSession.html
[17]https://stackoverflow.com/questions/15566480/how-do-i-get-master-key-in-ssl-session-in-java

Chapter 6

Evaluation

In this section, we evaluate our ACE-MQTT implementation and compare the performance of different authentication mechanisms against that of MQTT systems with plain insecure clients that do not use TLS nor any kind of authentication. We considered metrics identified as most relevant for a constrained IoT device, which are power demand, overall energy consumption, and resource utilization in terms of memory, CPU, and network load. Energy consumption is relevant because IoT devices are usually battery-powered and expected to last a significant amount of time without a battery replacement. Resource utilization is also relevant because IoT devices are usually constrained and connect over low-bandwidth networking infrastructure, thus cannot tolerate large overheads. Finally, we evaluate the implementation against compliance to the profile specification, in order to make sure our solution will be interoperable with different implementations of ACE-MQTT brokers or clients.

6.1 Energy consumption

We use a Raspberry Pi (RPi) 3 Model B+ device running the Raspbian Stretch Linux distribution, to measure the energy consumption overhead. We employ a UM34C USB power meter, which we place between the RPi and the power source, measuring voltage and current drain per second, and the power consumed by the device as a whole. There is no straightforward way to isolate the MQTT-related power consumption from the overall of the RPi thus we measure the whole consumption while reducing measurement noise, by disabling all the unnecessary features and peripherals of the RPi, including the USB controller, the Graphical User Interface and the video interface, and we operate the device via SSH.

We use the ACE-MQTT executable jar to launch ACE-MQTT and plain MQTT clients separately, and we measure the difference in the power consumption of the device when operating with each of these. To put things into perspective, we also examine the power footprint of the RPi when idle. We run multiple experiments and monitor a client repeatedly performing the following actions with a short sleep interval in-between:

Parameter	Value
Authentication repeat interval	2 seconds
Publish repeat interval	2 seconds
Pub/Sub QoS	At least once
Publish message length	17 bytes
Plain client MQTT client version	5
Authentication PoP	HS256
Pub/Sub client version	5
Client connectivity	Wi-Fi
Environment	RPi 3 B+
Client library	Executable jar
TLS cipher suite	TLS_ECDHE_RSA_WITH_ AES_128_GCM_SHA256
AS & broker network location	Same LAN as client

Table 6.1: Energy consumption experiment settings

- Complete authentication phase with AT request and broker authentication for both simple and challenge based.
- Publish request
- Subscribe request

The experiment settings are summarized in Table 6.1.

We first examine the power consumed over a 60-second interval, during which different types of authentications are performed, and messages are published and received by clients. We summarize the results in Figure 6.1. The first 4 sub-plots at the top show the power cost associated with challenge-based and simple authentication using MQTT version 3.1.1 and 5 respectively, and the footprint of plain MQTT unauthenticated CONNECT requests. Unsurprisingly, the insecure approach bears the smallest power cost, but it is interesting to observe that the power budget of all the authentication methods considered is very similar.

The ACE-MQTT client has an average power consumption of 1.2W per second during authentication, which is however only 10% higher than the average of a plain MQTT client and corresponds to a 15% increase from the idle state. This means that a system designer would only need to worry about the capabilities of the device and the most suitable authentication scheme for their deployment. We also measure the power consumption of plain MQTT and ACE-MQTT publishers and subscribers when sending/receiving messages on a certain topic, which is illustrated in the next 4 sub-plots in Figure 6.1. Clearly, the messaging costs are nearly the same for all approaches; once the connection has been established, the security guarantees of ACE-MQTT are the only thing that remains.

We then compute the average amount of time needed to establish a connection, and the associated power and energy consumption for the different authentication methods.

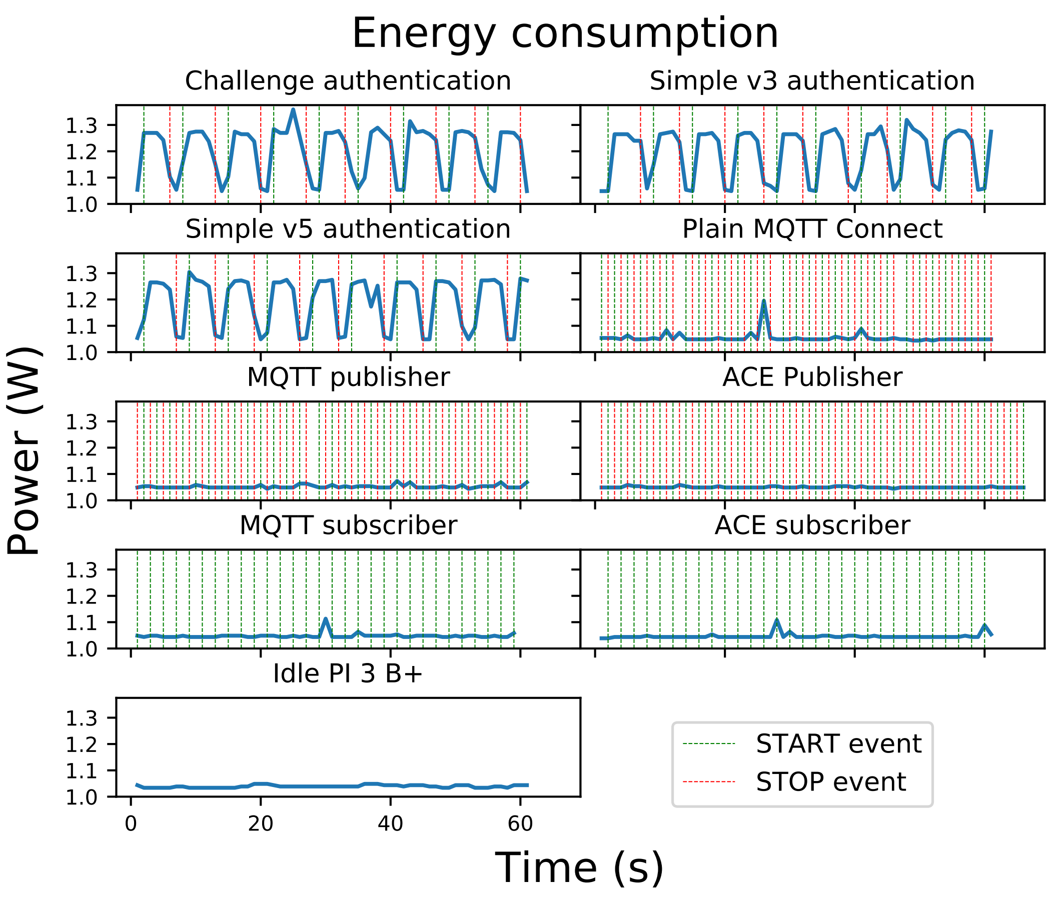

Figure 6.1: Power consumption during authentication (top 4 sub-plots), publish/subscribe actions (4 sub-plots in the middle), and during idle operation (bottom).

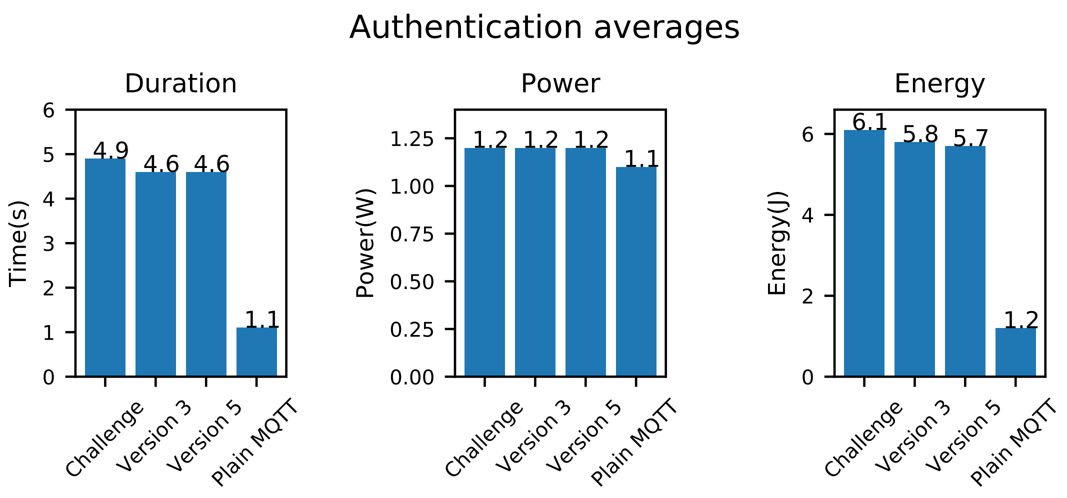

Figure 6.2: Average cost of authentication with the challenge-based authentication, MQTT v3 and v5 simple authentication, and plain MQTT, in terms of duration (left), power utilization (middle), and energy consumption (right).

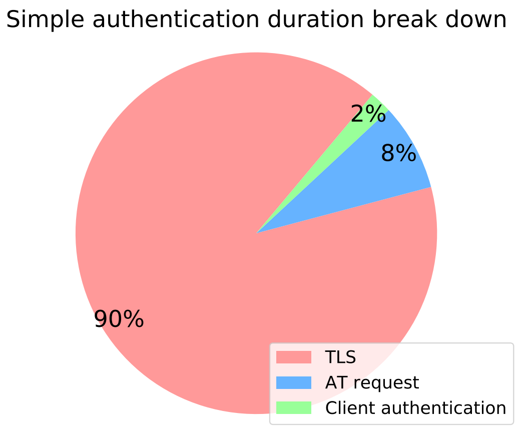

Figure 6.3: Break-down of version 5 simple authentication duration into a) TLS overhead b) AT request c) client authentication

We report the results obtained in Figure 6.2. As expected, since it involves the largest number of messages, the challenge-based authentication method takes the longest to complete, but the length is comparable to that of simple authentication in MQTT version 3.1.1 and 5. The insecure MQTT variant completes the connection establishment more than 4 times faster. In terms of instant power utilization, all methods are comparable. Thus, the overall energy consumption of each strictly depends on the duration of the authentication/connection phase. The challenge-based authentication requires around 5% more energy than simple authentication, due to the fact that it involves one extra round trip time. The plain MQTT method consumes around 5 times less energy to complete the connection.

We examine closely the duration of simple authentication(version 5), by breaking down the procedure into its different constituent parts and quantifying the contribution of each from a time requirement perspective. We remove a single component of the authentication phase in each test and measure the average time that a client needs to complete the authentication phase. First, we remove TLS and measure only the the AT request and the broker's authentication of the client. Without the time associated with the TLS handshake and data encryption of the CONNECT and CONNACK messages, the average time required to complete authentication comes down to half a second on average. Then, to deduct the overhead of the AT request, we reuse the same AT among different authentication requests and measure the new duration. The average time measured is 4.3 seconds corresponding to the TLS overhead and the client authentication. Finally, to quantify the overhead incurred by the client authentication, which includes AT introspection and PoP, the client request is authenticated automatically, as if the broker allowed unauthenticated requests. In this case, the average authentication time, including only the AT request and TLS overhead, is approximately 4.5 seconds. The breakdown of the time consumed by each component is summarized in Figure 6.3. Note that the AS was in the same Local Area Network as the broker and the client during the experiment, thus we would expect the AT request and client authentication to be slightly higher in reality, due to internet latency. We conclude that the TLS session overhead bears by far the highest overhead, which is not surprising, since our RPi does not support hardware cryptographic extensions, and thus, all the computations are dealt with directly in software.

6.2 Resource utilization

Next, we measure the computing resources used by ACE-MQTT and insecure MQTT clients to assess the additional overhead that comes with security. For this purpose, we ran simultaneously inside containers a secure ACE-MQTT client and a plain MQTT one, using the Docker compose setup. We measure the CPU, memory, and network utilization per second, for each container using our monitoring setup described in subsection 5.5.1. The only notable difference is that memory is measured using JMX[1] instead, in order to observe the usage of the Java process and not that of the whole JVM. The settings of these experiments were almost identical to those in the energy consumption measurement campaign, apart from the differences that the host environment is MacOS and the client library is an executable jar in a Docker container.

We test the different authentication methods against a plain MQTT client connecting with no authentication nor TLS and we report the results obtained over a 2-min window in Figure 6.4. Again, it comes at no surprise that plain MQTT bears the smallest demand. All ACE authentication methods incur significant overhead during the authentication phase, but require the same amount of resources in terms of memory, CPU, and network bandwidth. In particular, when compared to a plain MQTT client, an ACE client requires 10× more network resources, 3× more CPU cycles and 1.2× more memory to complete the authentication phase.

We also examine the overhead incurred by an ACE-MQTT client during normal operations, such as when publishing or receiving messages on subscribed topics. The results are shown in Figure 6.5. An ACE-MQTT publisher requires identical amount of CPU cycles and inbound network resources, but carries 15% more outbound network traffic and and 25% more memory. Meanwhile an ACE-MQTT subscriber requires almost identical amount of CPU cycles and inbound network resources, but around 15% more outbound network and memory resources.

We conclude that our ACE-MQTT implementation incurs a notable cost in terms of energy and resources only during the authentication phase, whereas the footprint of the different variants is on par with that of plain MQTT during publish/subscribe operations. This confirms the viability of the solution implemented to secure IoT ecosystem using MQTT.

[1]https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

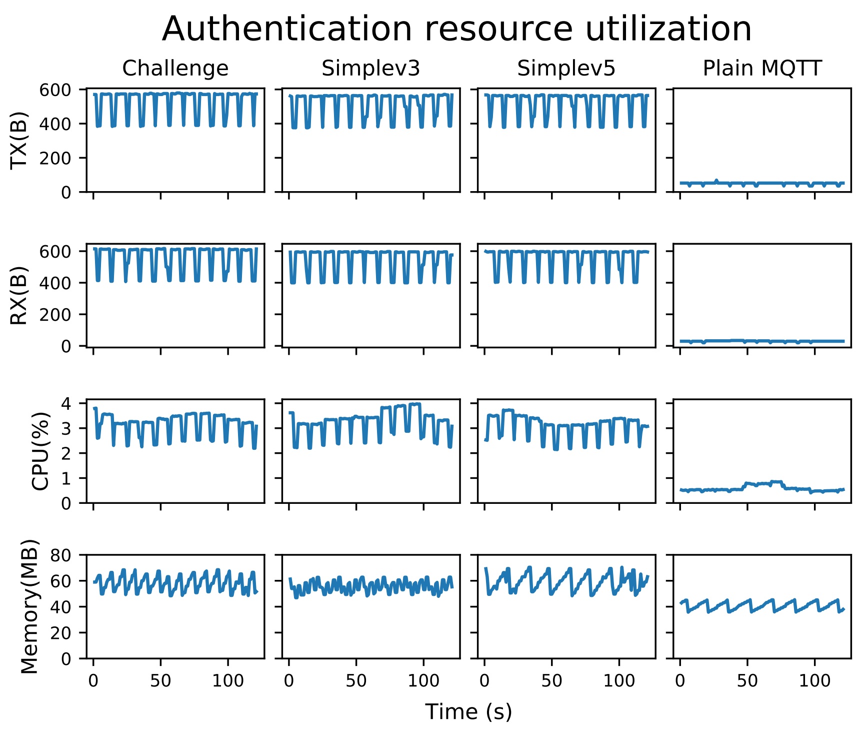

Figure 6.4: Resource utilization of ACE-MQTT with different authentication methods vs plain MQTT connection set-up. Rows present different resources (Network outbound, inbound, CPU utilization, memory) and columns different authentication methods.

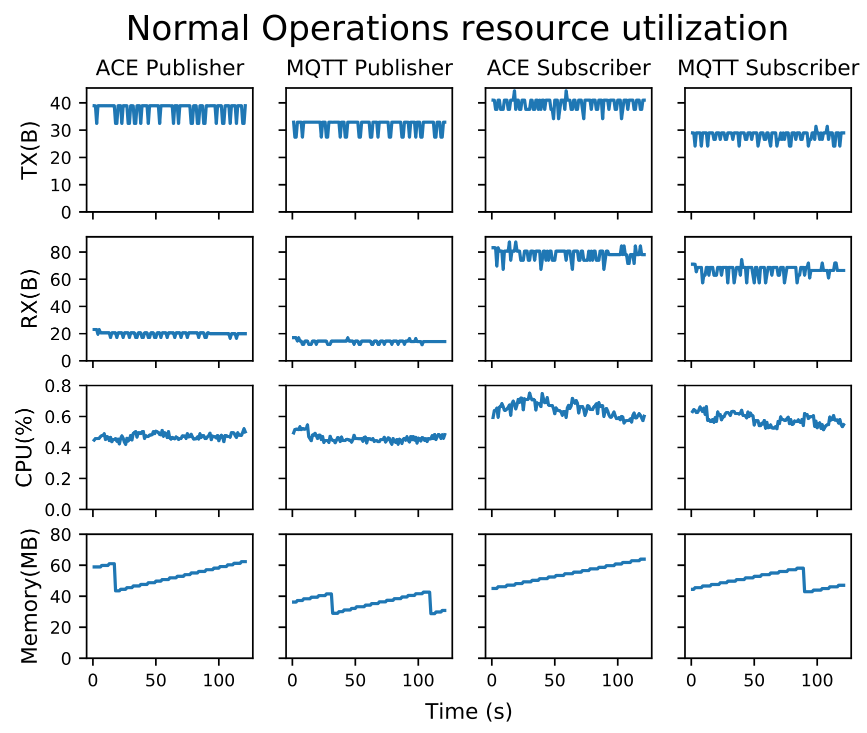

Figure 6.5: Resource utilization during normal operations. Rows present different resources (Network outbound, inbound, CPU utilization, memory) and columns contrast an ACE against an MQTT publisher and subscriber.

6.3 Standard compliance and interoperability

Finally we make sure that our implementation follows the profile specification, to ensure interoperability with other ACE-MQTT implementations that will arise in the future. There is no other public complete implementation yet to be used as a point of reference, thus we test compliance by capturing a trace of messages exchanged between the client and the broker and we contrast them with the specification.

To capture messages we use verbose logging in both the client and broker libraries. We could not use a packet sniffer because of TLS encryption. We initialized a client with the AT seen below, indicating a HMAC PoP based on the algorithm SHA-512. Both the AT and PoP key are base 64 encoded binary strings.

```
Access Token: 8pqr26vpucvx2fnyadcuhge18uzz3mywmfmx9pp11
d20uneafba846vnv46ztbxt9pu8ntw2t7t8gbm7dd69kavjjknv1ymmy3rmufgtw
0cpe24v5ym3bthq21nzcpyp87k19h969hct7wt0tx8udb8cwyantqwm84jr46hud
aggaynp8dbxnfz6xqmh6x3q40g8jeyp5ja73wf2bx27h49kz5ujkf8b859hqtv27
35nh8w7xfhm8rkcz1nt2qmd5q8d3r82z1v05akd7dzj4hh4tj1rx7w7p5tpw9wrk
x12hprx8td928kb461rq43cgq27c1qvdy2zb9ex2k3ymeejv8br6x84ttq332wzm
3mz7fuxkxbvk1m5djh9pehzzhyrqgnymuk4kkgauf26qpjakw,
PoP algorithm: HS512,
PoP key: g6ZBouHXtecOba3YNT5y8FY90lPLxb8GgN6bSghSx6ucAcqgxtDQsaGuc
V+xfgXDlW9kJ6nNCy2S0iuGgOypihj85SlEGrwClLWD8Cah3UU4SuTEn9HMNzfK
h4Sg6v4XwHKXs+tdCBtME+8jUduDMxqii628S2J6lmgnHCpvs58=
```

Next we present a trace of the authentication phase messages between the client and the broker and we contrast them with the profile specification.

6.3.1 Simple authentication with client version 3.1.1

First we captured a CONNECT packet sent from a version 3.1.1 ACE client to the broker which can be seen below. Observe that the *Clean Start* flag is set to true, the *Username* is set to the AT and the *Password* is set to the PoP. Note that the password value is the hexadecimal representation of the PoP length (0x0040 in hexadecimal equals 64 to decimal, the length of a SHA-512 hash in bytes) and the PoP, which is the HMAC of the whole packet under SHA-512.

```
Protocol version: V_3_1_1,
Clean Start: true,
Keep Alive: 60,
Maximum Packet Size: 268435460,
Receive Maximum: 65535,
Topic Alias Maximum: 0,
Request Problem Information: true,
Request Response Information: false,
Username: 8pqr26vpucvx2fnyadcuhge18uzz3mywmfmx9pp11d20uneafba84
6vnv46ztbxt9pu8ntw2t7t8gbm7dd69kavjjknv1ymmy3rmufgtw0cpe24v5ym3b
thq21nzcpyp87k19h969hct7wt0tx8udb8cwyantqwm84jr46hudaggaynp8dbxn
fz6xqmh6x3q40g8jeyp5ja73wf2bx27h49kz5ujkf8b859hqtv2735nh8w7xfhm8
```

```
rkcz1nt2qmd5q8d3r82z1v05akd7dzj4hh4tj1rx7w7p5tpw9wrkx12hprx8td92
8kb461rq43cgq27c1qvdy2zb9ex2k3ymeejv8br6x84ttq332wzm3mz7fuxkxbvk
1m5djh9pehzzhyrqgnymuk4kkgauf26qpjakw,
Password (hex):
0040
1a87bf5bcbb628850ae29212f3d4b727f44cdc7de97bfb3be63f02270bb523fd
abe6c531a0fa45fecf057674401ea59bd642241447b9c8d6ce640a8a349eff58,
```

6.3.2 Simple authentication with client version 5

We captured a CONNECT packet sent from a version 5 client to the broker which can be seen below. Note that the *Clean Start* flag is set to true, the *Session Expiry Interval* is set to 0, the *Username* and *Password* are empty and finally *Authentication Method* is set to 'ace' while *Authentication Data* contains both the AT and the PoP as length prefixed values. The first four hexadecimal characters represent the length of the AT, which is 312 in decimal, then the AT itself follows, and finally the PoP prefixed by its length.

```
Protocol version: V_5,
Clean Start: true,
Session Expiry Interval: 0,
Keep Alive: 60,
Maximum Packet Size: 268435460,
Receive Maximum: 65535,
Topic Alias Maximum: 0,
Request Problem Information: true,
Request Response Information: false,
Username: null,
Password: null,
Auth Method: ace,
Auth Data (hex):
0138
f29aabdbabe9b9cbf1d9f9f269d72e8607b5f2ecf3de6cb099f9b1f69a75d5dd
b4ba779a7db6bce3abe7bf8eb3b5bc6df69bbc9edc36b7bb7c81b9bb75debd91
abe38e49efd729a6cb7ae6b9f82dc347297b6e2fe729b76ed86adb59f3729ca9
f3b935f61f7af6172def0b74b71f2e75bf1cc326a7b6ac26f388ebe3a86e75a8
206b29e9f1d6f19dfcfac6a9a1eb1deae3483c8deca9e636bbdf07f66f1dbb87
8f64cf9ba391ff1bf39f61aadbf6ef7e6787cc3bc5f866f2b91ccf59eddaa99d
e6af1ddebf36cf5bf4e5a91deddce3e21878b63d6bc7bc3ba79b69c3dc2b931d
76869af1f2d77ddbc91be3ad6bab8ddc82adbb735aaf772db36fd7b1da4df299
e7a3bfc6ebeb1f38b6dab7df6c339b79b3edfbb19316ef9359b976387da5e873
ce1cabaa09f29ae93892481ab9fdbaaa98da93
0040
1a87bf5bcbb628850ae29212f3d4b727f44cdc7de97bfb3be63f02270bb523fd
abe6c531a0fa45fecf057674401ea59bd642241447b9c8d6ce640a8a349eff58,
User Properties: null
```

6.3.3 Challenge authentication

We captured the CONNECT packet of a challenge authentication request shown below. Note that the only difference is the *Authentication Data* field contains only the AT prefixed with its length.

```
Protocol version: V_5,
Clean Start: true,
Session Expiry Interval: 0,
Keep Alive: 60,
Maximum Packet Size: 268435460,
Receive Maximum: 65535,
Topic Alias Maximum: 0,
Request Problem Information: true,
Request Response Information: false,
Username: null,
Password: null,
Auth Method: ace,
Auth Data (hex):
0138
f29aabdbabe9b9cbf1d9f9f269d72e8607b5f2ecf3de6cb099f9b1f69a75d5dd
b4ba779a7db6bce3abe7bf8eb3b5bc6df69bbc9edc36b7bb7c81b9bb75debd91
abe38e49efd729a6cb7ae6b9f82dc347297b6e2fe729b76ed86adb59f3729ca9
f3b935f61f7af6172def0b74b71f2e75bf1cc326a7b6ac26f388ebe3a86e75a8
206b29e9f1d6f19dfcfac6a9a1eb1deae3483c8deca9e636bbdf07f66f1dbb87
8f64cf9ba391ff1bf39f61aadbf6ef7e6787cc3bc5f866f2b91ccf59eddaa99d
e6af1ddebf36cf5bf4e5a91deddce3e21878b63d6bc7bc3ba79b69c3dc2b931d
76869af1f2d77ddbc91be3ad6bab8ddc82adbb735aaf772db36fd7b1da4df299
e7a3bfc6ebeb1f38b6dab7df6c339b79b3edfbb19316ef9359b976387da5e873
ce1cabaa09f29ae93892481ab9fdbaaa98da93,
User Properties: null
```

The broker responds with an AUTH packet with the *Authentication Method* set to ’ace’, *Reason Code* set to CONTINUE AUTHENTICATION, and *Authentication Data* set to the broker’s 8 byte long nonce contribution prefixed by its length.

```
Reason Code: CONTINUE_AUTHENTICATION
Auth Method: ace
Auth Data (hex):
0008
3E54178F52F93F82
```

Finally the client responds with a similar AUTH packet that contains both the 8 byte length prefixed client nonce contribution and the PoP over the final nonce, which is the concatenation of the broker and client nonce contributions in that order.

```
Reason Code: CONTINUE_AUTHENTICATION
Auth Method: ace
Auth Data (hex):
0008
3BB3A47049F27E58
```

```
0040
9b4ed0d8aed3359145c11aa51c21f87a73e659649595d11af7e72505e5f9ac94
d7e9e03261e22483437f1e1f51288191ff2d649c736e864ae0fe63faa5d085ec
```

Chapter 7

Conclusions

7.1 Summary

The aim of this project was to secure MQTT, a popular IoT messaging protocol, by providing an implementation of the complete MQTT-TLS profile of ACE which adds a layer of authentication, authorization, data integrity, and confidentiality. We built on top of the HiveMQ client to create an ACE-MQTT client library that is easy to use and simple to migrate to from existing insecure deployments. We also created an extension to allow the HiveMQ CE broker to support the profile by authenticating and authorizing clients according to tokens provided by the Authorization Server (AS). We then employ automation mechanisms through containers to simplify the deployment of the system. Finally, we measure and provide a comparative upper bounded performance evaluation of resource utilization and energy consumption. The results show that deploying our implementations on constrained devices is viable and overhead is mainly incurred during the authentication phase. Since MQTT is based on persistent TCP connections, the additional resources and energy costs are easily amortized throughout sessions.

7.2 Future Work

There is a number of aspects worth exploring in future projects, in order to improve the implementation performance, increase the security level, optimize the code footprint and provide a more detailed evaluation.

First, future work can focus on profiling the different operations involved in the ACE-MQTT authentication process, to understand better at what stage improvements can be made. Our evaluation considers the authentication process as a whole, even though it consists of the TLS session establishment, AT request, connection request, AT introspection and finally PoP. Each of these steps can be evaluated separately to gain better insights about the cost of each and consequently what part to optimize.

Second, it is worth remembering that our implementation runs in user space and as such there is scope to investigate the use of hardware acceleration for cryptography, which is increasingly present in Arm and x86 processors that offer, e.g., specialized AES and SHA instructions. The Raspberry Pi (RPi) we used during the constrained device evaluation does not support hardware cryptography extensions. However, other moderately constrained devices, such as the Freedom Development Platfrom[1], have such support which could be a major factor in optimizing the authentication phase overhead.

Third, optimizing the choice and parameters of the TLS cipher suite has the potential to decrease the initial overhead of session establishment further. We did not greatly explore this option due to constrains from the Java environment, as well as due to time constraints. For example, Raw Public Keys[32] could be used if the broker is using a self signed certificate, which would also decrease the network overhead as a benefit. Unfortunately Java does not provide great flexibility in terms of cryptographic support, thus exploring lower level programming languages will be advantageous.

The choice of a lower level language could scale down the resources needed and decrease the code footprint as well. During the evaluation phase, we observed that the Java Virtual Machine adds significant overhead in terms of memory consumption. Our client implementation as a Java process uses less the 80MB as shown in the evaluation phase. However, the memory utilization of the whole JVM was consistently above 100MB. Assuming that we are running a single client process on each constrained device, then this is a significant amount of overhead.

Finally, we note that the client has to store sensitive information on board which needs to be properly secured. The client secret, if stolen, could be used to impersonate the client. Secure storage could be achieved with the help of a permission oriented operating system running on top of the client application, or with hardware support in the case of embedded devices. For example we could use Zymbit,[2] which provides an encrypted filesystem and key management support to secure Raspberry Pi (RPi) devices, as those we used for testing.

[1]https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F

[2]https://www.zymbit.com/blog-security-module-raspberry-pi/

Bibliography

[1] T. Anagnostopoulos, A. Zaslavsky, and A. Medvedev. Robust waste collection exploiting cost efficiency of iot potentiality in smart cities. In *2015 International Conference on Recent Advances in Internet of Things (RIoT)*, pages 1–6, 2015.

[2] S. Andy, B. Rahardjo, and B. Hanindhito. Attack scenarios and security analysis of mqtt communication protocol in iot system. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 1–6, 2017.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association.

[4] Edgaras Baranauskas, Jevgenijus Toldinas, and Borisas Lozinskis. Evaluation of the impact on energy consumption of mqtt protocol over tls. 2019.

[5] Jeff Barr. AWS IoT Cloud Services for Connected Devices. `https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/`, 2015.

[6] A. Bhawiyuga, M. Data, and A. Warda. Architectural design of token based authentication of mqtt protocol in constrained iot device. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–4, 2017.

[7] Leor Brenman. API Builder and MQTT for IoT. `https://devblog.axway.com/apis/api-builder-and-mqtt-for-iot-part-1/`, 2018.

[8] M. Calabretta, R. Pecori, and L. Veltri. A token-based protocol for securing mqtt communications. In *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, 2018.

[9] Tae Ho Cho and Jin Hee Chung. Adaptive energy-efficient ssl/tls method using fuzzy logic for the mqtt-based internet of things. *International Journal of Engineering and Computer Science*, 5(12), Nov. 2016.

[10] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. Anatomy of a vulnerable fitness tracking system: Dissecting the fitbit cloud, app, and firmware. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(1), March 2018.

[11] M. Collina, G. E. Corazza, and A. Vanelli-Coralli. Introducing the qest broker: Scaling the iot by bridging mqtt and rest. In *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC)*, pages 36–41, 2012.

[12] Google Cloud IoT Platform Documentation. Publishing over the MQTT bridge. `https://cloud.google.com/iot/docs/how-tos/mqtt-bridge`.

[13] Microsoft Azure Documentation. Communicate with your IoT hub using the MQTT protocol. `https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support`, 2018.

[14] A. Esfahani, G. Mantas, R. Matischek, F. B. Saghezchi, J. Rodriguez, A. Bicaku, S. Maksuti, M. G. Tauber, C. Schmittner, and J. Bastos. A lightweight authentication mechanism for m2m communications in industrial iot environment. *IEEE Internet of Things Journal*, 6(1):288–296, 2019.

[15] Eclipse Foundation. IoT Developer Survey 2019. `https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf`, 2019. Protocol specification for MQTTv3.1.1.

[16] P. Fremantle, B. Aziz, J. Kopeck, and P. Scott. Federated identity and access management for the internet of things. In *2014 International Workshop on Secure Internet of Things*, pages 10–17, 2014.

[17] S. Gerdes, O. Bergmann, C. Bormann, G. Selander, and L. Seitz. Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE). Internet Draft, RFC Editor, December 2019.

[18] M. S. Harsha, B. M. Bhavani, and K. R. Kundhavai. Analysis of vulnerabilities in mqtt security using shodan api and implementation of its countermeasures via authentication and acls. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 2244–2250, 2018.

[19] Santiago Hernández Ramos, M Teresa Villalba, and Raquel Lacuesta. Mqtt security: A novel fuzzing approach. *Wireless Communications and Mobile Computing*, 2018, 2018.

[20] Y. Liu, J. Niu, L. Yang, and L. Shu. ebplatform: An iot-based system for ncd patients homecare in china. In *2014 IEEE Global Communications Conference*, pages 2448–2453, 2014.

[21] Trend Micro. Silex Malware Bricks IoT Devices with Weak Passwords. `https://www.trendmicro.com/vinfo/`

fr/security/news/cybercrime-and-digital-threats/-silex-malware-bricks-iot-devices-with-weak-passwords, 2019.

[22] A. Niruntasukrat, C. Issariyapat, P. Pongpaibool, K. Meesublak, P. Aiumsupucgul, and A. Panya. Authorization mechanism for mqtt-based internet of things. In *2016 IEEE International Conference on Communications Workshops (ICC)*, pages 290–295, 2016.

[23] OASIS. Mqtt version 3.1.1. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2014. Protocol specification for MQTTv3.1.1.

[24] OASIS. Mqtt version 5. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, 2019. Protocol specification for MQTTv5.

[25] F. Palombini. Pub-Sub Profile for Authentication and Authorization for Constrained Environments. Internet Draft, RFC Editor, January 2020.

[26] D. Pavithra and R. Balakrishnan. Iot based monitoring and control system for home automation. In *2015 Global Conference on Communication Technologies (GCCT)*, pages 169–173, 2015.

[27] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). RFC, RFC Editor, March 2010.

[28] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig. Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth). Internet Draft, RFC Editor, February 2020.

[29] Kirby A. Fremantle P. Sengul, C. MQTT-TLS profile of ACE (draft-ietf-ace-mqtt-tls-profile-04). Internet Requests for Comments, 2020. Internet Draft.

[30] Shodan. Insecure MQTT deployments report. https://www.shodan.io/report/G0y5T1eD, 2020. MQTT deployments worldwide with no TLS transport security nor client authentication.

[31] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. Blackiot: Iot botnet of high wattage devices can disrupt the power grid. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 15–32, Baltimore, MD, August 2018. USENIX Association.

[32] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, RFC Editor, June 2014.

[33] Lucy Zhang. Building Facebook Messenger. https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920, 2011.

[34] Ray Y. Zhong, Xun Xu, and Lihui Wang. Iot-enabled smart factory visibility and traceability using laser-scanners. *Procedia Manufacturing*, 10:1 – 14, 2017. 45th SME North American Manufacturing Research Conference, NAMRC 45, LA, USA.

[35] W. Zhou and S. Piramuthu. Security/privacy of wearable fitness tracking iot devices. In *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–5, 2014.