An Experimental Evaluation of MQTT Authentication and Authorization in IoT

Michael Michaelides
University of Edinburgh
United Kingdom
mimichas@protonmail.com

Cigdem Sengul
Brunel University London
United Kingdom
cigdem.sengul@brunel.ac.uk

Paul Patras
University of Edinburgh
United Kingdom
paul.patras@ed.ac.uk

ABSTRACT

Security vulnerabilities make the Internet of Things (IoT) systems open to online attacks that threaten both their operation and user privacy. Among the many protocols governing IoT operation, MQTT has seen wide adoption, but comes with rudimentary security support. Specifically, while the MQTT standard strongly recommends that servers (brokers) offer Transport Layer Security (TLS), it is mainly concerned with the message transmission protocol, leaving to implementers the responsibility for providing appropriate security features. However, well-known solutions for Web Security (OAuth2) exist, which may benefit MQTT. This paper presents systematic implementation efforts and practical experimentation to evaluate the feasibility of one such approach, namely the MQTT-TLS profile for the Authentication and Authorization in Constrained Environments (ACE), recently specified by the IETF. Our implementation includes the functionality for (1) the Authorization Server (AS), to handle client registration, authorization policies, and Access Tokens; (2) the MQTT broker, to enforce authentication in both MQTT versions 3.1.1 and 5. Together, these enable ACE-MQTT clients to use (3) OAuth2-based authentication and authorization via Proof of Possession tokens. We make the source-code of our ACE-MQTT implementation publicly available, and evaluate it against plain MQTT systems in realistic settings with different computation constraints. To assess the cost of security, we measure the CPU, memory, network usage, and energy consumption. The results obtained confirm that the ACE requirements match the capabilities of moderately constrained devices, hence providing an affordable mechanism to secure MQTT systems.

CCS CONCEPTS

• **Networks** → **Network experimentation**; **Cyber-physical networks**; • **Security and privacy** → **Security protocols**.

ACM Reference Format:
Michael Michaelides, Cigdem Sengul, and Paul Patras. 2022. An Experimental Evaluation of MQTT Authentication and Authorization in IoT. In *The 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization (WiNTECH '21), January 31–February 4, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3477086.3480838

WiNTECH '21, January 31–February 4, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-8703-3/22/01…$15.00
https://doi.org/10.1145/3477086.3480838

1 INTRODUCTION

Internet of Things (IoT) deployments usually consist of devices that perform measurement and monitoring (e.g., temperature sensing) and transmit information to cloud-based services over the Internet. The primary concern in IoT is typically the need for the systems to scale up to tens of thousands of devices. Security, including authentication, data integrity, and confidentiality, often remain an afterthought. These mixed-up priorities create risks, as IoT devices become susceptible to malware infection [1], services prone to disruption [18], and user data vulnerable to privacy leaks [6, 11].

The Internet Engineering Task Force (IETF) Authentication and Authorization in Constrained Environments (ACE) workgroup is developing new IoT security standards and strengthening protocols that have already seen wide adoption, yet are still vulnerable to attacks. One such protocol is MQTT [15], a lightweight publish/subscribe message transport protocol used in over 40% of existing IoT deployments [8]. MQTT works on top of TCP and has minimal security, e.g., TLS is strongly recommended, and client authentication can be enabled via username and password. To improve MQTT security, the MQTT-TLS profile of ACE enables authentication and more fine-grained authorization using the OAuth2.0 framework and Proof of Possession (PoP) Access Tokens (ATs) [16].

To the best of our knowledge, this work is the first to document an implementation of the ACE MQTT-TLS profile and a thorough practical assessment of its performance for MQTT versions 3.1.1. and 5. The paper makes the following main contributions:

(1) We implement the ACE Authorization Server (AS), the MQTT broker and the clients. The AS handles client registrations, manages authorization policies, and grants Access Tokens (ATs). The MQTT broker accepts ACE-based authentication and authorization requests for MQTT versions 3.1.1 and 5. The clients dynamically register with the AS and authenticate with the broker. Proof of Possession (PoP) ATs[1] permit clients to publish or subscribe to specific topics based on the token's scope (i.e., permissions). We make the source code of our implementation publicly available to foster the development of future extensions by the community, and we highlight the lessons learned and challenges faced in the development process.
(2) We evaluate the performance of our solution in a testbed encompassing containerized and constrained environments. Specifically, we consider use case scenarios with small form IoT boards (with CPUs clocked at sub-GHz speeds, sub-GB RAM, and network adapter), capable of running minimal Unix-type operating systems and suitable for monitoring and control tasks,

[1] Clients need to compute a keyed hash or their digital signature to use the PoP AT, providing evidence for the symmetric key or the corresponding private key of the public key embedded in the token.

e.g. building automation, smart factories, energy distribution, automotive, etc. Our evaluation takes into consideration CPU, memory, and power budgets. Results demonstrate that the enhanced security comes at the cost of a marginal increase in resource usage, as compared to plain MQTT.

(3) We reveal that the energy consumption associated with the authentication procedure is mostly due to the communications overhead and running cryptographic protocols in user space. We discuss possible ways to reduce this cost via hardware cryptography extensions.

2 RELATED WORK

Token-based authentication in MQTT is an active area of research. Bhawiyuga et al. implement a solution for constrained devices, but their evaluation is limited to usability and Authorization Server (AS) response time [2]. Calabretta et al. propose protecting MQTT topics based on the Augmented PAKE (Password-Authenticated Key Agreement) protocol, yet no evidence of performance is given [4]. Collina et al. propose QEST, a RESTful MQTT broker, and explore how to incorporate OAuth support into the MQTT system, but an implementation and evaluation are not provided [7]. Niruntasukrat et al. introduce an OAuth1a-based system, where both the broker and client devices have embedded OAuth credentials [13]. As communication channels are considered insecure, clients generate a new signature upon every request which complicates the system and requires one extra round trip to the AS, which is resource-expensive. Further, using insecure channels means the system is prone to eavesdropping and man-in-the-middle attacks.

La Marra et al. propose usage control to enhance the security of MQTT, yet assume the connections are already authenticated [10]. Elliptic curve-based encryption of MQTT messages is proposed in [17], but the authors assume that the topics and subscribers to these topics are known to brokers, which is impractical. Chiwariro and Rajendran propose a lightweight encryption scheme to ensure MQTT payload confidentiality, yet client authentication is overlooked [5]. Similarly, payload encryption with broker pass-through is proposed in [12], without addressing client authentication.

The work of Fremantle et al [9] is closely related to ours as their design uses OAuth2 tokens for authentication and authorization. However, their work considers clients that do not support SSL/TLS, thus sacrifices the confidentiality and integrity of MQTT data. With a single embedded token and insecure communication channel, the solution is vulnerable to replay attacks and eavesdropping. Solutions relying on embedded tokens are not very flexible, as the device firmware must be flashed to change the token.

To our knowledge, there are no implementations and evaluations that take advantage of the latest features of MQTT version 5.

3 BACKGROUND

3.1 MQTT Overview

MQTT is a popular lightweight publish/subscribe messaging protocol. Two standardized versions exist, namely version 3.1.1 [14] and version 5 [15]. In MQTT, the central server, called broker, is responsible for relaying messages between clients. MQTT uses persistent TCP connections, and clients connect to the broker using a CONNECT packet. After connection establishment, clients

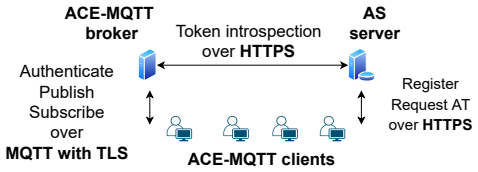

Figure 1: Components of an ACE-MQTT system and communication channels between them. AS and broker communicate over HTTPS; clients communicate with the broker using MQTT over TLS, and with the AS using HTTPS.

publish or subscribe to 'topics' hosted by the broker. When the broker receives a PUBLISH message, it forwards the message to all clients subscribed to its topic. To subscribe to topics, clients send SUBSCRIBE messages containing topic filters. A topic filter may include wildcards to match multiple topics.

In terms of security, version 3.1.1 supports optional username/password-based authentication of clients, provided in the CONNECT packet. MQTT version 5 addresses a set of limitations of version 3.1.1 and also has more support for security.[2] Two new authentication fields are introduced in the CONNECT packet: Authentication Method and Authentication Data. With these, a client can signal to the broker which authentication method to use and can provide different types of authentication data (apart from username and password). Additionally, version 5 supports a new AUTH packet that can be used to extend the authentication phase, such as to add a challenge, or to re-authenticate. These additions are flexible enough to implement different authentication methods but exact method to use is out of scope of MQTT specification.

3.2 ACE MQTT TLS Profile

The ACE MQTT-TLS profile [16] describes MQTT client authentication and authorization using OAuth2 ATs. To this end, MQTT topics hosted by a broker are treated as resources to be protected. An OAuth2.0 framework entity called Authorization Server (AS) is responsible for registering clients, maintaining authorization policies for publishing and subscribing to these protected topics, and granting ATs to clients.

All three entities - AS, broker, and client - communicate over pairwise secure channels, as shown in Figure 1. The clients and optionally, the broker communicate with the AS over HTTPS, and between them use MQTT over TLS. Both the broker and AS use TLS certificates to authenticate to the clients.

Clients are not assumed to have TLS certificates; thus, they authenticate with the broker using OAuth2 ATs. ATs issued by the AS are accepted by the broker as a valid form of authentication. Each AT is associated with the client requesting it, and specifies the permissions of the client, i.e., to which topics the client may publish or subscribe. The ACE-MQTT broker needs to validate the AT presented by a connecting client before authenticating it. The token may be self-contained, and can be directly inspected by the broker. Optionally, the broker can introspect the AT with the AS.

For valid tokens, the broker performs PoP verification so that the client can prove the ownership of its AT (see Sec. 2.2.4 of [16] for details). PoP tokens prevent leaked or eavesdropped ATs from

[2]We sometimes refer to version 3.1.1 as version 3, for simplicity.

being used to impersonate other clients. By default AS generates a symmetric PoP key for the client and binds it to the token. Asymmetric key pairs can also be used, but the client needs to add its public key to its token request for AS to embed into the AT.

Different authentication methods are available for clients using different MQTT versions. Clients using version 3 have to overload the username and password fields with the AT and PoP respectively, while version 5 clients can make use of the new Authentication Data field. Version 5 clients can also choose between a simple or challenge-based PoP, whereas version 3 clients can only use the simple method. In the challenge-based PoP, both the client and the broker contribute to creating a nonce to compute a PoP. This method requires an extra round trip. In the simple version, the client pre-computes the PoP, just using channel binding, i.e., based on a challenge associated with the TLS session using the TLS exporter (see Sections 2.2.4.1 and 2.2.4.2 of [16] for more details).

Upon successful authentication, the topics the clients are authorized to publish or subscribe to are determined by the 'scope' parameter in AT. We implemented the 'scope' based on an earlier version of the profile as a space-separated set of permission strings, where each permission authorizes publishing or subscribing to a topic filter, e.g., *publish_topic*1 *subscribe_topic*2/# (current MQTT-ACE profile represents scopes using the Authorization Information Format (AIF) for ACE [3]).

To authorize a PUBLISH message from a client, the broker needs to check the client permissions by finding a scope sub-string in the AT that matches the topic field in the PUBLISH packet. Next, the broker verifies that each subscriber AT is still valid before it forwards the message to a particular subscriber.

SUBSCRIBE message authorization may be more involved because both the scope sub-strings and the SUBSCRIBE topics are topic filters, which can contain wildcards. For each topic filter in the request, the broker checks if the AT contains a scope sub-string that is a super-set of that topic filter. If this is the case, the broker authorizes that topic filter.

Support for authorization errors also depends on the MQTT version. MQTT version 3 does not allow the broker to indicate a publish or subscribe authorization failure, nor it allows a server-side disconnect. Thus, the broker can only drop the TCP connection in case of an authorisation failure. MQTT version 5 enables better error reporting: the broker can send a negative publish or subscribe acknowledgement for authorization failures. As such, the client can re-authenticate by obtaining and submitting a new AT without reconnecting. Re-authentication is resource-efficient, since TLS session initiation is expensive. Finally, a broker can also gracefully disconnect version 5 clients by sending a server-side disconnect.

4 IMPLEMENTATION

In this section, we summarize the architecture and the implementation of the ACE-MQTT system.

4.1 Architecture

The overall architecture is shown in Figure 2.

4.1.1 Authorization Server. The AS is an Express-based HTTPS server with OAuth2 support, and uses OAuth2orize and Passport to implement the authorization framework.[3] The server uses its TLS certificate to authenticate to the clients and the broker. The Mongoose object data modelling library[4] is the interface between the server and a mongoDB database, which contains client credentials, active ATs and policies. The AS supports the following public API:

- *Client registration endpoint* is an unauthenticated endpoint that allows new clients to register with the AS. It expects the client name and URI parameters, which can be pre-configured in an ACE-MQTT client, and the name should be unique in the domain. On successful registration, the AS retains the new client details in its database, and then it issues client credentials, which include the client ID and client secret. The secret is tied to an expiry date, after which it has to be reset.
- *Policy management endpoint* is accessed by resource owners to add, update and delete client authorization policies. Resource owners are any entity that can authoritatively decide access permissions to an MQTT topic. With authorization policies, resource owners dictate which clients can access which topics and how. A single policy consists of a client ID, scope and expiry date.
- *Access Token (AT) request endpoint* is accessed by registered clients to request a token. The AS checks its policy database to determine if the client is authorized for the requested scopes.
- *Introspection endpoint* is accessed by the broker to introspect ATs. This endpoint is optional.

4.1.2 ACE-MQTT Broker. The broker is a Java extension to the HiveMQ Broker Community Edition[5]. We choose to build on the HiveMQ instead of other publicly available Mosquitto-based implementations[6] due to the flexible and well-documented extension SDK that HiveMQ offers. We believe the overhead of the Java Virtual Machine can be managed by MQTT brokers, which typically have sufficient resources, e.g. to provide high availability.

Our implementation provides the HiveMQ 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 and version 5 clients;
(3) *PoP verification* of authenticating clients;
(4) *AT Introspection* to obtain token associated information, such as PoP and authorization details;
(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 clients of the AS location.

The major components of the broker extension are shown in Figure 3. 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

[3] AS source code is public: https://github.com/ciseng/ace-mqtt-mosquitto. Express: https://expressjs.com/. OAuth2orize: https://github.com/jaredhanson/oauth2orize. Passport: http://www.passportjs.org/.
[4] https://mongoosejs.com/
[5] We make our source code publicly available on GitHub. https://github.com/michaelg9/HiveMQACEextension HiveMQ: https://www.hivemq.com/developers/community/
[6] https://mosquitto.org/ and Machine-to-Machine bridges such as Ponte based on Mosquitto https://www.eclipse.org/ponte/

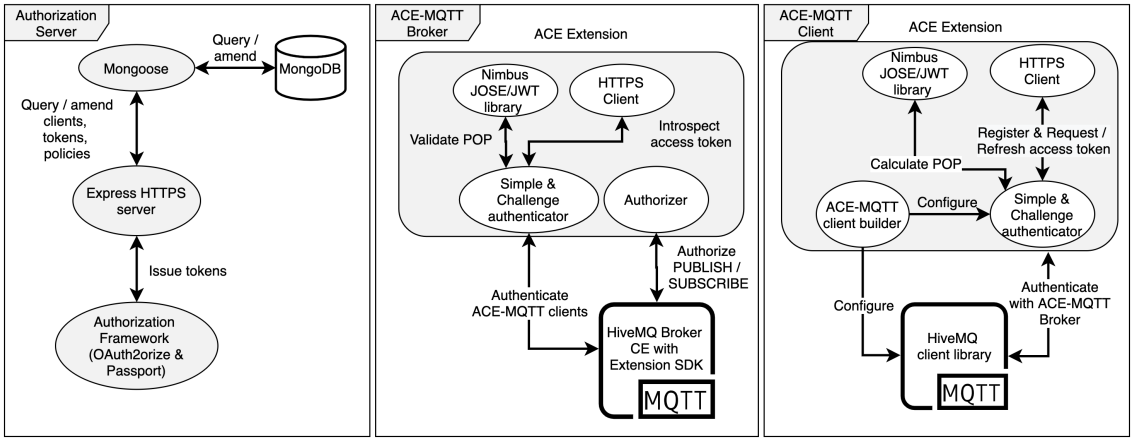

Figure 2: The architecture of all the entities in an ACE-MQTT system.

MacCalculator class and recording authenticated clients and their ATs in the ClientRegistry class. MacCalculator verifies Message Authentication Codes (MACs) using the Nimbus JOSE/JWT library.[7] ClientRegistry is queried by the authorization class AceAuthorizer, to enforce permissions and check expired ATs. AceAuthorizer contains methods for checking the AT expiry date for publishers or subscribers. It is further equipped with an instance of a PublishOutboundInterceptor class, which checks the expiry date of the AT of subscribers before the broker forwards them a published message.

With AS discovery option, an ACE-MQTT client may request the AS location from the broker by sending a CONNECT packet with no Authentication Data. The broker responds to this CONNECT message with an authorization error but uses the new "user properties" field in the CONNACK packet to inform the client about AS properties. This optional feature may simplify the initial configuration of the client and allow a non-static AS location.

In retrospection, the choice of the base broker library did not give us enough control over the TLS session handling. The broker library depends on the default Java cryptographic API for TLS support, which doesn't allow extensive fine tuning,e.g. not allowing us to use TLS exporters. Also, there was not much flexibility on the TLS session algorithm choice and configuration; for example we wanted to try lightweight TLS cipher suites that do not use certificates, such as TLS-PSK, instead of using self signed certificates but these weren't supported. Had the broker library given us control over the transport layer parameters, we could have switched to the BouncyCastle library which adds all these missing features to Java. Thus it is important that a future implementation chooses a base broker library that gives full control of the TLS session to the ACE-MQTT broker.

4.1.3 ACE-MQTT client. The client is a Java extension that acts as a wrapper over the HiveMQ MQTT client.[8] Our extension provides a simple public API to configure the client and choose between different options, such as authentication type. The implementation extends the HiveMQ client with the following capabilities:

[7]https://connect2id.com/products/nimbus-jose-jwt
[8]The source of our implementation is at https://github.com/michaelg9/HiveACEclient.

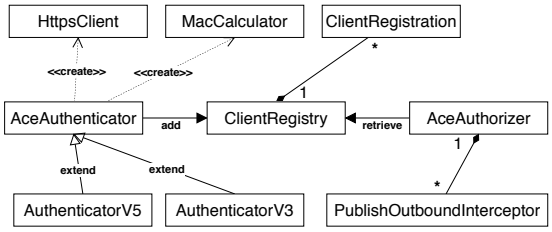

Figure 3: Main components of ACE broker extension used for authentication, authorization, access token tracking.

(1) *TLS and HTTPS support* for securing the communication with the ACE-MQTT broker and the AS;
(2) *Client bootstrapping* supports 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;
(3) *Proof of Possession (PoP)* based on a MAC or digital Signature proof of AT ownership;
(4) *Authentication* using simple and challenge-based flows;
(5) *Re-authentication* on AT expiry for version 5 clients.

The architecture of our client extension is illustrated in Figure 4.

In our implementation, a version 3.1.1 client can choose between simple authentication, which is the default, or no authentication, if the client is not used as an ACE client. On the other hand, a version 5 client can select between simple, challenge-based, and no authentication. Any additional actions or settings required to make the client comply with the ACE-MQTT protocol, such as AS discovery, client registration, AT request and transport protocol settings are performed automatically, without user interaction.

Client bootstrapping. A client instance requires initial configuration before it can execute. The configuration is made up of key-value pairs, with the keys shown as required and optional parameters in Table 1. E.g., the location of the broker needs to be known. In our set-up, the AS and the broker have self-signed TLS certificates; thus, the clients should be pre-configured with a trust store.

The configuration file is parsed and client initialization is performed in steps. First, the AS IP address is looked up, and if found,

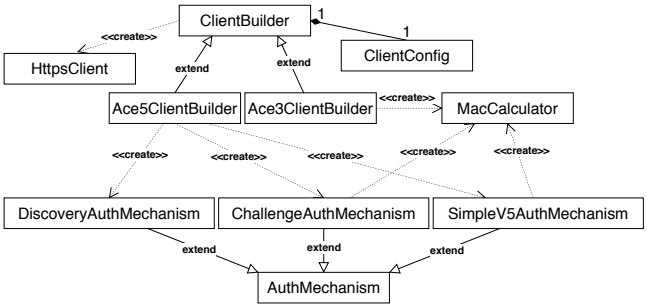

Figure 4: Main components of ACE client showing base ClientBuilder class (top) and its dependencies (HttpsClient, ClientConfig), the version 3 and 5 client builder sub-classes (Ace3-/Ace5-ClientBuilder), the three v5 authentication mechanisms (Discovery-/SimpleV5-/Challenge-AuthMechanism) and MacCalculator used to calculate the PoP.

Parameter	Required
Broker IP address	Yes
Broker port	Default if missing
AS IP address	Yes, unless discovery is possible
AS port	Yes
SSL/TLS key & trust store	Yes, if SSL/TLS is used
Transport protocol	No, 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 1: Client initial configuration parameters

then the client registration is checked, i.e., if the configuration includes a client id and secret. If that is not the case, then the client is registered with the AS registration endpoint. The retrieved client id and secret are saved in the configuration file to avoid undergoing the registration phase next time. Then, the MQTT session is configured according to the transport protocol parameter; the default value is 'TLS', according to the ACE-MQTT protocol, however, the user may set it to 'TCP' instead, if the client does not support TLS. Finally, if the AS IP address is not found, the client proceeds to discover this if possible.

Lessons learned: In hindsight, the choice of client library would be best implement in C. As revealed by our experiments, the memory footprint of the Java Virtual Machine is non negligible and can prove problematic in highly constrained devices. In addition to that, the cryptographic API natively available in Java is not as extensive as in the OpenSSL API provided in C environments. Thus, it does not support a feature similar to the OpenSSL TLS exporter, which is required in order to obtain a nonce established during the TLS handshake for the MQTT simple authentication phase. To circumvent this problem, we used a different source to generate nonces, as we detailed in the next subsection.

Next, we describe how the clients and broker mutually authenticate to achieve the security guarantees of the ACE-MQTT protocol.

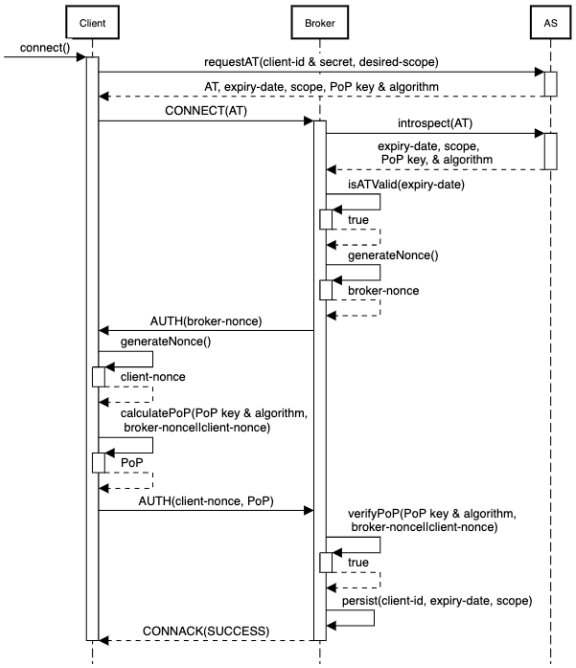

Figure 5: Challenge-based Authentication

4.2 ACE-MQTT PoP-based Authentication

In this section, we divide the two-way authentication phase between an ACE-MQTT client and the broker into steps and describe the implementation details of each. How the PoP is performed depends on the type of authenticator. The MQTT-TLS profile describes two mechanisms: (1) pre-computed PoP based on the challenge value returned using the TLS exporter, which we call simple authenticator, and (2) a challenge-based scheme, initiated by the broker. Version 3 clients can only support the simple authenticator, while version 5 clients support both mechanisms.

Client-side implementation. Due to lack of support for TLS-exporter in the development environment, we have implemented the authenticators differently. In the simple implementation, the PoP is computed based on the contents of the CONNECT message, which may not have enough randomness, and hence, may not be as secure. However, it allowed us to have a performance comparison between the two different mechanisms. Depending on whether symmetric or asymmetric keys are used, the PoP is a Message Authentication Code (MAC) or a digital signature.

The challenge-based authentication (shown in Figure 5) uses the broker and client-generated nonces. The challenge-response phase is initiated by the Broker sending an 8-byte challenge to the client. The authenticator then generates a secure cryptographic random nonce and concatenates the two nonces to calculate the PoP. The client nonce and PoP are returned to the broker as an AUTH packet.

Broker-side implementation. When the broker receives the CONNECT packet, it first checks whether the packet is formed correctly. Then it checks the validity of the AT and PoP. For this, it may introspect the client AT using the AS introspection endpoint. Next, the broker must validate the PoP. In the case of a simple authenticator, the broker generates the expected PoP using the PoP key and

algorithm in the token. Then, it compares this with the provided PoP and authenticates the client if the two match.

In the case of the challenge-based authenticator, on receiving the CONNECT packet, the broker generates a cryptographic nonce. It responds with an AUTH packet containing the nonce with reason code "CONTINUE_AUTHENTICATION". The client performs the PoP and returns it in an AUTH packet. To validate the PoP, the same procedure as in the simple authenticator is then followed.

If the PoP is valid, the authenticator accepts the request. First, it caches the introspection response in an in-memory map with the client id as the key. Then it parses the scope of the AT into a topic permissions list and applies it to the configuration of the client. Both of these are used during authorization, as we explain next.

4.3 Client Authorization

Client authorization was built on top of the existing HiveMQ broker authorization library to satisfy the ACE-MQTT authorization requirements. Client authorization 1) ensures clients can only access topics defined in their scope and only for the allowed action, i.e., publish or subscribe, 2) checks AT expiration for each publish or subscribe request, and 3) provides error reporting when the AT is expired or when the request is not authorized.

Scope permissions are implemented as part of the Broker's ACL (Access Control List) containing a rule for each permission defined by the scope in the AT for that particular client. Each permission represents the action (publish or subscribe) the client is allowed on a topic filter. Hence, when the client publishes, the topic name in the packet needs to match with the topic filter of at least one rule that permits publishing. On the other hand, when a client sends a subscribe request, each topic filter in the request is compared to all the rules in the ACL, until a match is found, i.e., there is a rule with an equal or broader topic filter. Then the broker authorizes the subscription request only for the topic filters that were authorized.

Finally, ATs must be validated when the broker has messages to forward to subscribers. If a token cannot be validated and a request must be refused, the broker may either drop the connection (version 3 clients) or send a server-side disconnect message or a negative acknowledgement (version 5 clients).

Lessons learned: There were no major issues in the implementation of this phase. However, the authorization of subscribe messages requires extra attention when comparing topic filters between them. We had to ensure that each request topic filter was a subset of an authorized topic filter. This can prove too strict under some circumstances, e.g., when the request filter is broader than the authorization one, but the corresponding MQTT topics match. Another trade-off in this phase is between performance and freshness of ATs. If a client AT is revoked before it expires, the broker would not know that in the current implementation, because it uses caching. If we choose not to cache, then the broker must introspect each AT for each client action, which can become too expensive in a production environment. In this case we prioritized performance, with the assumption that ATs are not revoked on the AS.

5 PERFORMANCE EVALUATION

Our ACE-MQTT development is based on an enterprise MQTT broker and client solution, HiveMQ, and therefore, does not consider very constrained IoT set-ups. Nevertheless, in this section, we evaluate our implementation based on its resource utilization to assess the additional overhead that comes with security. We compare its performance with different authentication mechanisms against that of MQTT systems with plain insecure clients that do not use TLS or any authentication. The metrics we consider are memory, CPU, and network load, power demand and overall energy consumption.

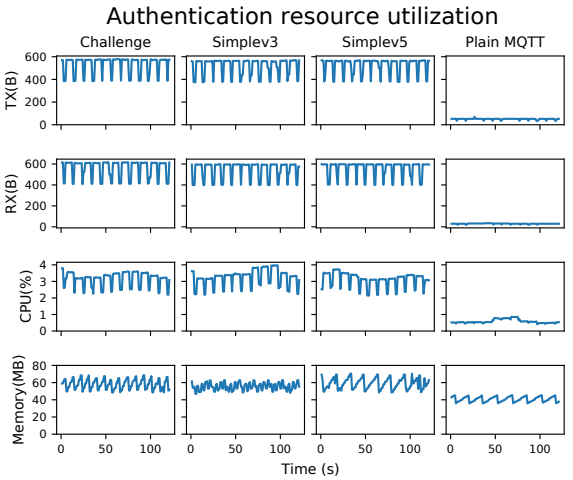

Figure 6: Resource utilization of ACE-MQTT with different authentication methods vs plain MQTT connection set-up.

5.1 Resource Utilization

To measure the computing resources used by ACE-MQTT and insecure MQTT clients, we ran simultaneously a secure ACE-MQTT and a plain MQTT client inside containers, using a Docker compose setup. We measure the CPU, memory, and network utilization per second, for each container. Specifically, we use the Google Container Advisor to query the container statistics and a Prometheus time series database to store the readings [9]. The only notable difference is that memory is measured using JMX[10] to observe the usage of the Java process and not that of the whole JVM. One key benefit of working with containers is that the setup allows running multiple different clients simultaneously, in synch at the same phase of operation, and obtain a meaningful comparison between them.

We test the different authentication methods against a plain MQTT client connecting with no authentication and report the results obtained over a 2-min window in Figure 6. As expected, plain MQTT bears the smallest demand. All ACE authentication methods incur high overhead during the authentication phase, and 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.

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 7. An ACE-MQTT publisher requires identical amount of CPU cycles

[9]https://github.com/google/cadvisor; https://prometheus.io/
[10]https://www.oracle.com/java/technologies/javase/javamanagement.html

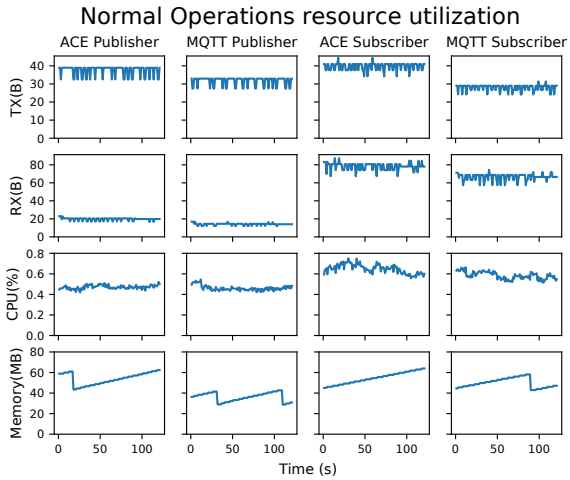

Figure 7: Resource utilization during normal operation.

and inbound network resources, but carries 15% more outbound network traffic and requires 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.

5.2 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. To reduce measurement noise, we disable all the unnecessary features and peripherals of the RPi, including the USB controller and the video interface, and operate the device via SSH.

Since there is no straightforward way to isolate the MQTT-related power consumption, 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 a 1) complete authentication phase with AT request and broker authentication, 2) publish request, 3) subscribe request, with a short sleep interval in-between. The settings are summarized in Table 2.

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. The ACE-MQTT client has an average power consumption of 1.2W per second during authentication, which is only 10% higher than the average of a plain MQTT client and corresponds to a 15% increase from the idle state. Note that the power consumption in the idle state accounts for the requirements of the operating system along with the HiveMQ implementation but without any MQTT traffic. Thus, a system

Parameter	Value
Authentication repeat interval	2 seconds
Pub repeat interval	2 seconds
Pub/Sub QoS	At least once
Pub message length	17 B
Plain client MQTT client version	5
Authentication PoP	HS256
Pub/Sub client ver.	5
Client connectivity	Wi-Fi
Environment	RPi 3 B+
TLS cipher suite	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
Client library	Executable jar
AS & broker location	Same network as client

Table 2: Energy consumption experiment settings

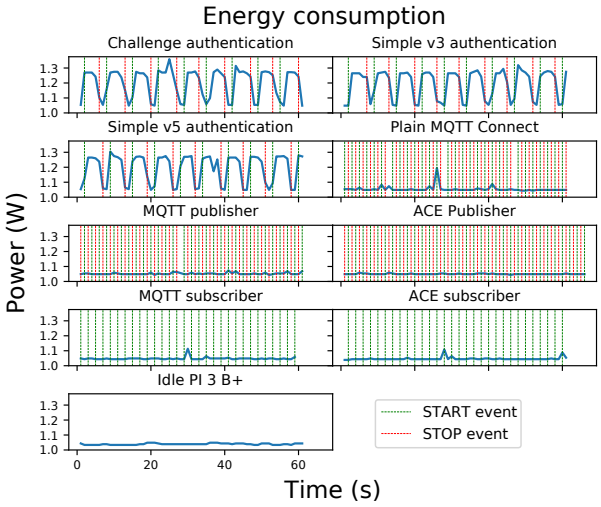

Figure 8: Average cost of authentication with the challenge-based authentication, MQTT v3 and v5 simple authentication, and plain MQTT, in terms of duration (left), power utilization (middle), and energy consumption (right).

designer only needs to worry about the capabilities of the device and the most suitable authentication scheme for their deployment.

We then compute the average duration of establishing a connection, and the associated power and energy consumption for the different authentication methods considered, across 10 individual experiments. We report the results obtained in Figure 8. As expected, since it involves the largest number of messages, the challenge-based authentication method takes the longest to complete, but the duration is comparable to that of simple authentication in MQTT version 3 and 5. The insecure MQTT variant completes the connection establishment more than 4 times faster. As the power utilization of all methods is comparable, the 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.

We examine closely the duration of simple authentication (version 5), by breaking it down into its different parts and quantifying

their time requirement. To this end, 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. More specifically, we first remove TLS session initiation 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 MQTT 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 session and the broker 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 session initiation) is approximately 4.5 seconds, with TLS overhead, AT request, and broker authentication accounting for 90%, 8%, and respectively 2% of the duration. Note that the AS was in the same LAN 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. 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 DISCUSSION

The cost of adding authentication, authorization, data integrity, and confidentiality to MQTT lies largely in the session authentication phase of ACE. It should be noted however that ATs can be re-used among different sessions as long as they are not expired, thus amortizing the overhead of requesting a token among sessions. Thus, these results present the upper bound cost of ACE-MQTT authentication. On the other hand, power, memory, and CPU consumption is also influenced by the security operations involved, i.e., PoP, generation of cryptographic nonces, and TLS encryption.

Recall 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 processor that offer, e.g., specialized AES and SHA instructions. In addition, optimizing the choice and parameters of the TLS cipher suite has the potential to decrease the overhead further. Finally, a lower level language could scale down the resources needed.

Note that the client has to store sensitive information on board in ACE-MQTT, which needs to be secured. This includes the client secret, which could be used to impersonate the client if leaked. 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 one could use Zymbit,[11] which provides an encrypted filesystem and key management support, to secure Raspberry Pi (RPi) devices, as those we used in our experiments.

Finally, given the wide deployment of MQTT v3.1.1, we expect MQTT v5 servers to offer dual-stack support for both type of clients. In this case, the MQTT v3.1.1 clients have access only to limited functionality and would need to be upgraded, to benefit from advanced ACE features such as improved error reporting, and the ability to re-authenticate with new tokens during the same session. However, it is important to avoid mixed operation, e.g., MQTT v3.1.1 clients connecting to brokers that support only MQTT v5. This may cause problems even if the broker implements the simple authentication method. For instance, MQTT v3.1.1 clients are expected to ignore the fields they cannot parse in the error messages sent by the broker (e.g., PUBACK with authorisation errors), hence they may not be aware that the broker does not forward their messages to their subscribers in the case of an authorisation failure. Therefore, careful consideration needed when supporting a mix of v3.1.1 and v5 clients.

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

7 CONCLUSIONS

In this paper we presented a comprehensive implementation of the MQTT-TLS profile of ACE for MQTT v3.1.1 and v5 and results of a comparative performance evaluation. We have shown that ACE-MQTT has acceptable overhead, which is mainly incurred during the authentication phase. Since MQTT is based on persistent TCP connections, amortizing the additional resources and energy costs throughout sessions can be straightforward. Implementation improvements can also be achieved by using hardware cryptographic acceleration, lower level languages, and secure storage.

ACKNOWLEDGEMENTS

This work was partially supported by Arm Ltd. The authors acknowledge the help of Dominik Obermaier with the HiveMQ SDK.

REFERENCES

[1] Manos Antonakakis et al. 2017. Understanding the mirai botnet. In *"USENIX" Security*.
[2] Adhitya Bhawiyuga et al. 2017. Architectural design of token based authentication of "MQTT" protocol in constrained "IoT" device. In *TSSA*.
[3] C. Bormann. 2020. An Authorization Information Format (AIF) for ACE (draft-bormann-core-ace-aif-09). Internet draft (work in progress).
[4] Marco Calabretta et al. 2018. A Token-based Protocol for Securing "MQTT" Communications. In *SoftCOM*.
[5] Ronald Chiwariro and S Rajendran. 2018. Security in Publish/Subscribe Protocol for Internet of Things. *Pure and Applied Mathematics* 118, 16 (2018), 231–242.
[6] Jiska Classen et al. 2018. Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware. *ACM IMWUT* 2, 1 (March 2018).
[7] Matteo Collina et al. 2012. "Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST". In *IEEE PIMRC*.
[8] "Eclipse Foundation". 2020. "IoT Developer Survey".
[9] P. Fremantle, B. Aziz, J. Kopeckỳ, and P. Scott. 2014. Federated identity and access management for the internet of things. In *Intl Wksp Secure IoT*.
[10] Antonio La Marra et al. 2017. "Improving MQTT by Inclusion of Usage Control". In *Security, Privacy, and Anonymity in Computation, Communication, and Storage*.
[11] Haoyu Liu et al. 2019. Uncovering Security Vulnerabilities in the Belkin WeMo Home Automation Ecosystem. In *IEEE PerCom Workshops*.
[12] Suja P Mathews and Raju R Gondkar. 2019. Protocol Recommendation for Message Encryption in "MQTT". In *IconDSC*.
[13] Aimaschana Niruntasukrat et al. 2016. "Authorization mechanism for MQTT-based Internet of Things". In *IEEE ICC Workshops*.
[14] "OASIS". 2014. "MQTT Version 3.1.1".
[15] "OASIS". 2019. "MQTT Version 5.0".
[16] C. Sengul et al. 2020. "MQTT-TLS profile of ACE" (draft-ietf-ace-mqtt-tls-profile-04). Internet Draft.
[17] Meena Singh et al. 2015. Secure mqtt for internet of things "(IoT)". In *Intl Conf. Comm. Sys & Netw. Tech.*
[18] Saleh Soltan et al. 2018. BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid. In *"USENIX" Security*.