Scalable Identity and Key Management for Publish-Subscribe Protocols in the Internet-of-Things

Prashant Anantharaman
pa@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

Kartik Palani
palani2@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Sean Smith
sws@cs.dartmouth.edu
Dartmouth College
Hanover, NH, USA

ABSTRACT

Publish-Subscribe protocols such as the Message Queuing Telemetry Transport (MQTT) protocol are considered scalable, lightweight, and one-size-fits-all solutions for the Internet-of-Things (IoT) networking. MQTT has been widely adopted in the Industrial IoT to automate distributed power grid equipment such as smart meters and sensors. Such protocols are being adopted rapidly, without much attention being paid to security. Although these protocols support client-side TLS certificates, operators often do not enable these features, fearing performance and availability issues. Moreover, managing these certificates would be yet another challenging problem.

We present MaQaTooT, a key-management and communication scheme based on Macaroons for the IoT and Smart Grid applications. MaQaTooT offers a technique to authenticate devices throughout their lifecycle, while sustaining the lightweight nature of MQTT, and also keeping the communication confidential and maintaining its integrity. Furthermore, it allows us to revoke keys reliably. To validate our key-management scheme, we built a prototype client for the Firefly RK3288 ARM Development Board and a key-management server for a GNU/Linux machine. We demonstrate that its performance on the prototype client fits the 4 ms latency limit of Industrial IoT protocols. We also verified our session-key establishment protocol using Proverif to ensure that the protocol never leaks the shared secrets.

CCS CONCEPTS

• **Security and privacy → Security protocols**.

KEYWORDS

Macaroons, Key Management, Security, MQTT

ACM Reference Format:
Prashant Anantharaman, Kartik Palani, and Sean Smith. 2019. Scalable Identity and Key Management for Publish-Subscribe Protocols in the Internet-of-Things. In *9th International Conference on the Internet of Things (IoT 2019), October 22–25, 2019, Bilbao, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3365871.3365883

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IoT 2019, October 22–25, 2019, Bilbao, Spain

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7207-7/19/10…$15.00
https://doi.org/10.1145/3365871.3365883

1 INTRODUCTION

The pervasive nature of the Internet-of-Things (IoT) has provided attackers with a new set of devices to exploit and misuse. Developers have deployed several of their IoT devices, such as web cameras, refrigerators, and oil and gas pipelines on the Internet with public IP addresses and barely any authentication. These devices speak to their administrators, and among each other, using a messaging pattern known as publish-subscribe protocols.

As the name suggests, publish-subscribe protocols comprise two message types — publish, when a device writes a message to a channel, and subscribe, when a device subscribes to a channel and wants to receive all the messages sent to that channel. Two extensively used publish-subscribe protocols in the Industrial IoT are the Message Queuing Telemetry Transport (MQTT) protocol and the GOOSE IEC61850 protocol, used in Power Grid substations to control relays in real time.

Based on adoption data [10], we decided to further analyze implementations of the MQTT protocol. On Shodan, a port-scanning website, we analyzed the public IP addresses responding to MQTT messages [11]. Over 65% of these servers were returning a “Connection Accepted” message, implying that they do not need a client-side TLS certificate or a password.

Bringing security guarantees to publish-subscribe protocols such as MQTT presents specific challenges. First, updating all the clients as well as the servers is a massive undertaking. The purpose of the server or the broker is to receive messages from the clients, maintain lists of subscribed clients, and forward the messages to the subscribers. Any security scheme proposed for publish-subscribe protocols must not require any updates to the servers.

Second, publish-subscribe protocols do not present the senders’ identifiers along with messages. A receiver does not know which device sent the messages. All messages are encrypted and sometimes signed using TLS between the clients and the server, but the server decrypts the message to check the channel it has to be sent to and then forwards the messages. Any device connected to the server can publish messages to channels. As mentioned earlier, anyone can post messages to channels on over 65% of the servers running MQTT.

Third, in case of a server compromise, any adversary can read any messages as the server decrypts the packets before they are

This material is based upon work supported by the Department of Energy’s Office of Cybersecurity, Energy Security, and Emergency Response and the Department of Homeland Security’s Security Science & Technology Directorate under Award Number DE-OE0000780.

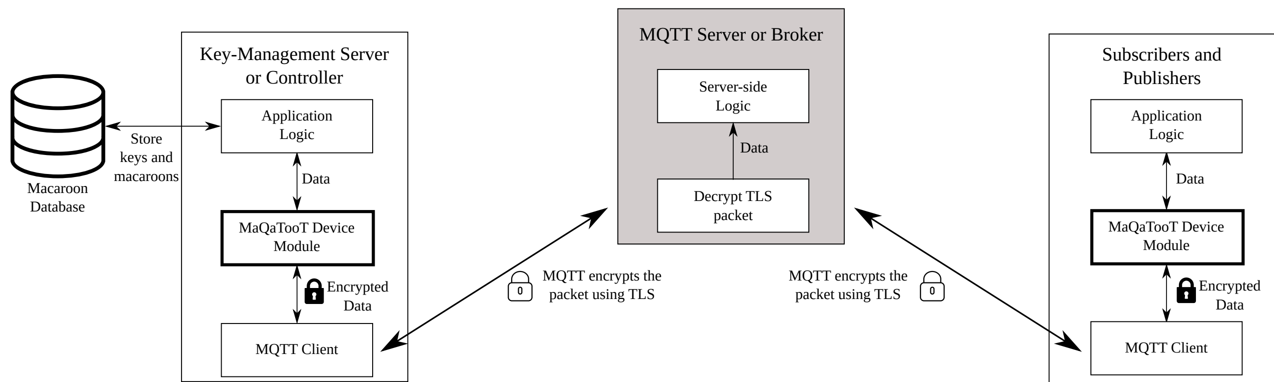

Figure 1: An overview of publish-subscribe system components. These systems comprise controllers that aggregate all the data. We reuse these controllers to introduce a key-management server. The MQTT server or broker forwards all the messages to the correct subscribers. The shaded component has access to encrypted data only. The components with the thick borders are introduced by MaQaTooT.

forwarded. A server compromise leads to serious privacy leakages. In the case of Industrial IoT, a server compromise leads to adversaries learning how the components function. The adversary could later use this information to send messages to these devices that could damage them.

Finally, since Industrial IoT devices communicate real-time sensor data and need to receive commands in real-time, the latency limitations are fundamental to any scheme designed to secure them. For example, the GOOSE protocol specifies a maximum latency of 4 ms. We cannot use a key-management system that violates these latency requirements.

Our key-management scheme, MaQaTooT, addresses the above concerns with existing publish-subscribe protocols and proposed security solutions for them. MaQaTooT involves embedding devices with multiple keys known as Macaroons [3]. One of these macaroons is long-term and specifies all the channels the device needs to access. The channel-specific macaroons are short-term and expire often. The messages also include a Hashed Message Authentication Code (HMAC) to ensure that the message was not altered in transit.

Our system appends the senders' identifier to each message, so that the devices know the sender. The channel-specific keys and device-specific keys help devices to communicate with each other without servers and adversaries without access to the channel-specific keys. MaQaTooT protects against data leakages in active server compromises. We use J-PAKE [6] to establish the session keys based on the shared keys, and we use symmetric-key cryptography to ensure that we stay within the latency requirements. Devices ignore any messages that are not encrypted using the correct keys, thereby providing properties of authentication, confidentiality, and integrity.

Our Contributions: MaQaTooT brings forth several contributions to the Internet-of-Things and Smart Grid Security communities:

- We build a key-management scheme for publish-subscribe protocols that is compliant with the availability requirements of smart grid and IoT protocols.
- Our tool requires minimal effort to incorporate as a separate layer on client as well as administrative applications.
- We verify that the session-key establishment protocol does not reveal the shared secrets between key-management servers and devices using the Proverif cryptographic protocol verifier.

Organization: We have organized the rest of the paper as follows. Section 2 describes publish-subscribe IoT protocols and reviews macaroons. In Section 3, we propose an architecture for our system MaQaTooT and describe our communication protocols. In Section 4, we analyze the effectiveness of our system with respect to latency requirements and programmer effort, and we also prove that our protocol does not leak the shared secrets. We review related work in Section 5 and present pitfalls in prior work. Finally, Section 6 presents conclusions.

2 BACKGROUND

2.1 Publish-Subscribe Protocols

Unlike traditional server-client protocols, in the publish-subscribe paradigm of messaging the information provider (publisher) is decoupled from the consumers of that information (subscriber). All publishers and subscribers are essentially clients, and the server only handles authentication and message forwarding.

In the most straightforward setting, described in Figure 1, there exists a publisher, a server (or a broker) and subscribers. The subscriber makes one or more subscriptions at the server, and the server keeps track of how these packets need to be forwarded when a device publishes a message.

Publishers send data to the server, and the server then forwards them to the subscribers. In Industrial IoT networks comprising sensors and actuators, sensors publish data to their respective channels, whereas an administrator or a device with administrative privileges publishes commands to be handled by the actuators.

The Generic Object Oriented Substation Events protocol (GOOSE) and MQTT are the most prominent publish-subscribe protocols

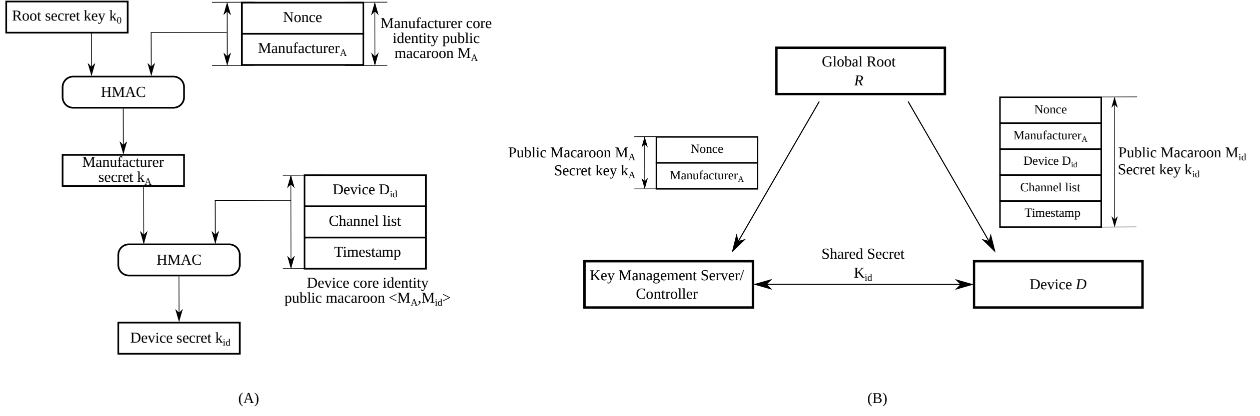

Figure 2: (A) Macaroons can be chained in a way similar to certificates. The key-management server can append the caveats granted to devices and compute the secret key k_{id}. (B) A Global Root key is used to sign the first caveat in the Macaroons. The Macaroon <M_A, K_A> is granted to the Key-Management Server. Device D presents <M_{id}> to the key-management server. The key-management server verifies the Macaroon, computes k_{id} and performs a handshake. This figure is adapted from Figure 1 in our earlier paper [2].

used in the Industrial IoT. The protocol is used in the command and control of Power Grid substations, and it supports operations such as analog and digital statuses, circuit statuses, breaker control, and transformer temperatures. All GOOSE communications use Ethernet. GOOSE suffers from similar issues as MQTT — authentication is not enforced, senders' addresses aren't included, and it does not include encryption [7].

We analyzed the MQTT deployments on the Internet on Shodan and found that over 65% of these implementations did not use authentication by default. Although the MQTT specification requires TLS communications, the server decrypts all the packets it is forwarding. The publish-subscribe model without any authentication also means that any device can subscribe to any channel, and any device can publish to any channel. MaQaTooT aims to address all these issues.

2.2 Macaroons

IoT devices are resource constrained. To satisfy the latency requirements of IoT protocols, we use macaroons instead of using traditional elliptic-curve cryptography [3]. Macaroons are flexible authentication credentials, which consist of two portions. The first is a public portion which comprise a nonce and data elements, together known as caveats. Macaroons also include a private part — a secret that is computed by iteratively running the HMAC algorithm on each of the caveats.

MACAROON_GENERATE
(device D_{id}, manufacturer M, key k_1, caveats C)

1. M: k_2 := HMAC256 $(N_M)_{k_1}$
2. M: k_3 := HMAC256 $(D_{id})_{k_2}$
3. M: k_4 := HMAC256 $(M)_{k_3}$
4. M: k_{i+1} := HMAC256 $(C_i)_{k_i}$ for i = 4,5...n
5. M: k_{id} := HMAC256 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel$ timestamp $\parallel k_{id}$

Figure 3: Macaroon Generation Procedure.

Figure 2-A shows the functioning of Macaroons. There are two ways to construct a macaroon. First, the manufacturer can use the Global Root signing key to build HMACs with all the caveats one by one iteratively. The final HMAC is considered to be the shared secret.

The second way to construct a Macaroon is when the key-generation authority does not have the root key, but has some intermediate key. As shown in Figure 2-A, we can compute HMACs on the rest of the caveats in the public portion of the Macaroon iteratively, and grant the final key generated, k_{id}, to the device as its secret. Figure 3 shows the procedure to generate a Macaroon. We use k_0 as the root secret and use it to sign the *Nonce*. The obtained HMAC is used to sign the following caveats iteratively.

In Figure 3, ($N_M \parallel D_{id} \parallel M \parallel C_i \parallel$ timestamp) is the public portion of the Macaroon M_{pub}, whereas k_{id} is kept private [3]. We compute the HMAC of the Nonce N_M first, and then compute the HMAC iteratively on the Device ID D_{id}, the Manufacturer ID M, and the Caveats C_i, and finally append the timestamp to specify the expiry of the Macaroon.

Revoking keys. Macaroons make use of epoch counters in the timestamp. Each Macaroon includes the validity caveat, specifying when the Macaroon expires. Macaroons can also expire by blacklisting the public portion and the corresponding secret keys. The key-management server checks against the blacklist before authenticating devices.

3 PROPOSED ARCHITECTURE

In our design of MaQaTooT, we address the following questions:

- How can we build a key-management scheme that is secure against attacks such as replay attacks and man-in-the-middle attacks?
- Can we revoke keys reliably?
- Can we prevent the server from leaking data when it is compromised?

3.1 Overview

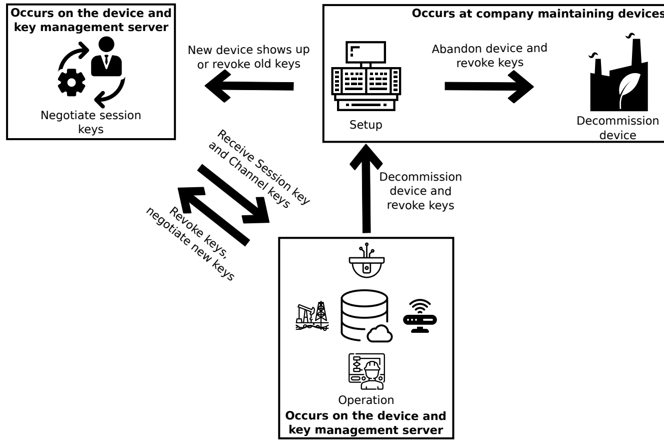

Figure 4: Lifecycle of IoT devices using MaQaTooT.

The challenges with Industrial IoT devices are unique because they are hard to reach physically. Companies often need to "roll trucks" to replace a malfunctioning device. As shown in Figure 2, we propose embedding a symmetric key k_{id} in the IoT devices during setup. Only the device and the key-management server are aware of this key. They use this symmetric key to establish a session key. The device can use this key to communicate throughout its lifecycle and the key-management server can also safely revoke these keys and grant a new one.

Scaling any architecture to all devices in an Industrial IoT network is a challenge. We propose using two identities associated with each device [2].

- Core-Identity Macaroon.
  Each device receives a Macaroon during setup, to identify the device uniquely. The device uses this core-identity Macaroon later to authenticate and receive the other short-lived Macaroons.
- Association-Attribute Macaroons.
  Whenever the channel-specific keys expire, the device initiates a negotiation using its core-identity macaroons to establish the keys for all the channels the device needs to access. These are short-lived and expire on a regular basis.

Figure 4 shows the entire lifecycle of devices using MaQaTooT. Devices are embedded with their core-identity macaroons when they are set up. The device and the key-management server negotiate a session key using J-PAKE, and the key-management server grants the channel-specific keys and macaroons for the device to use. Once the keys expire, the device negotiates the key again with the key-management server. When the device is past its life, the key-management server blacklists the device and revokes all the keys granted.

Each device has a device-specific channel on the MQTT server, where the device uses the session key established using J-PAKE. On this channel, the key-management server then communicates the association-attribute macaroons and keys, which are short-lived.

SHORT_LIVED(device D_{id},
key K_{id}, caveats C, channels S, Controller A)

1. $D_{id} \rightarrow A$: M_{id} || timestamp || HMAC256(M_{id} || timestamp)$_{k_{id}}$
2. A: k_{id} := CALC_KEY(M_{id})
3. A: k_2 := HMAC256 $(N_{new})_{k_id}$
4. A: k_3 := HMAC256 $(D_{id})_{k_i}$
5. A: k_4 := HMAC256 $(S_j)_{k_i}$ $\forall j \in S$
6. A: k_{i+1} := HMAC256 $(C_i)_{k_i}$ for i = 4,5,6...n
7. A: k_{s_j} := HMAC256 (timestamp)$_{k_n}$
8. $A \rightarrow D_{id}$: M_{id} || N_{new} || D_{id} || C_i || timestamp || k_{s_j}

Figure 5: Procedure to grant short-lived macaroons. The device presents its core-identity macaroon and receives its association-attribute macaroons for channels. The communication between the controller or key-management server and the device is encrypted using the session key.

3.2 Components

The architecture of MaQaTooT is shown in Figure 2-B. MaQaTooT consists of two components:

MaQaTooT Device Module. The MQTT implementation on the device needs to communicate with the key-management server and receive the association-attribute macaroons. The device sends the public portion of the Macaroon to the key-management server to authenticate itself. The key-management server can generate the key based on the public part of the Macaroon sent by the device and the secret key provided by the Global Root.

Key-Management Server. The key-management server serves multiple purposes. First, it provides short-lived association attributes to all the devices trying to establish a connection. Since the core-identity macaroons granted to the devices are an "attenuation" of the macaroons given to the key-management server, the key-management server can compute the secret and establish a channel.

Second, the key-management server can also revoke keys when devices are compromised. If an adversary gains access to the keys of a device and its subsequent channels, the key-management server has to revoke the keys for all those channels and alert all the connected devices to use the new keys. This functionality of the key-management server is central to the integrity of all the devices connected to the server.

3.3 Protocols

- Session-Key Agreement.
  Since we have a pre-shared key in the form of the Macaroon public k_2, we need to make use of a protocol that can generate session keys from a pre-shared key. We use J-PAKE to generate the session key for a fixed amount of time [6]. The session key has to be renegotiated at the end of this pre-agreed amount of time.
- Granting of Short-lived Macaroons.
  Figure 5 shows how short lived Macaroons are generated by the key-management server A. A device D_{id} sends its identifier Macaroon M_{id} along with the timestamp.
- Sending messages using Macaroons.
  Figure 6 shows the procedure to send normal messages using MaQaTooT. The device looks up the channel-specific key,

uses it to compute the HMAC, and then appends the HMAC to the data. Both are encrypted using AES256 and sent to the broker. The J-PAKE algorithm is used to establish session keys between the key-management server and the device, for the device-specific channel only.

OPERATION(device D, broker B, receiverGroup G)

1. D: $k_c := getChannelKey(G)$
2. D: m_{data} := data | timestamp
3. D: m_{mac} := HMAC256(m_{data})
4. $D \rightarrow B$: $m_{final} := enc\ (m_{data} \mid m_{mac})_{k_c}$
5. $B \rightarrow G$: m_{final}

Figure 6: Procedure to send messages using macaroons. The messages are encrypted using core-identity macaroon secret key for the device channel and channel-specific keys for other channels.

- Key Revocation.
  When the key-management server realizes that an adversary has compromised a device, the server calls the API to revoke the keys it granted to the device. Revoking a key involves blacklisting the previous key and granting a new key to be used in subsequent communications. If the key-management server withdraws the core-identity macaroon, then the newly generated Macaroon needs to be installed in the device manually. When the association attributes for a channel need to be revoked, the server looks up all the devices that are associated with the channel, and revokes the keys for all devices with access to the channel.

3.4 Threat Model

An adversary can fully control devices and servers. From a compromised server, an adversary can block any messages going through the server, or even forward messages to incorrect channels. The adversary cannot, however, decrypt any of the messages sent to the server by any of the clients, since we do not store any of the keys on the server. We assume that the adversary cannot break the cryptographic protocols we use.

A compromised device presents a more significant challenge. A device has access to its channel and channels set up to communicate with other devices. An adversary cannot, however, listen to all the channels on the device or send arbitrary messages on any channel on the server.

As long as none of the devices with access to keys are compromised, MaQaTooT provides confidentiality guarantees — something that is missing in most implementations of the protocol.

3.5 Assumptions

To make the above security claims, we make a few assumptions. We assume that the software implementations on various devices deployed will not send the secret keys to adversaries and that the software is exploit-free. Buffer overflows in the application on the device or the server could lead to an adversary gaining control of a device and put our security guarantees in jeopardy.

3.6 Limitations

MaQaTooT has some limitations as well. It does not address any side-channel attacks. Systems built with MaQaTooT would still be vulnerable to several unmitigated side channels. MaQaTooT also does not hide metadata. An adversary connected to the server would know which channels are active and which are not, but would not know anything about the contents of these messages.

In case of active server compromises, MaQaTooT does not prevent an adversary from altering the code on the server and preventing messages from being forwarded. An adversary could perform a denial-of-service attack in that manner.

3.7 Architecture

Publish-subscribe applications comprise devices, controllers and brokers. For our implementation, we leave the broker completely untouched. We add code on the controller and the device as shown in Figure 2. Table 1 shows all the methods available through the API that MaQaTooT exposes.

3.8 Implementation

We implemented the device client and key-management server APIs for MaQaTooT in Python. Most developers using the MQTT protocol want to use an off-the-shelf MQTT broker and do not wish to alter it. We do not change the MQTT broker code and instead use the Mosquitto MQTT broker [9]. Our implementations make use of the API calls described in Table 1.[1]

3.9 Summary

This section presented the design and implementation of MaQaTooT. It is an innovative key-management and scalable-identity scheme for large-scale IoT networks such as the ones using publish-subscribe protocols. MaQaTooT adds features that we found to be missing in current implementations of publish-subscribe protocols such as MQTT and GOOSE. Our design does not expose any device communication on the MQTT server, even in case of a compromised server, while providing for authentication, integrity, and confidentiality of the communications. Only devices with access to the channel-specific macaroons can access these communications.

4 RESULTS AND DISCUSSION

When we evaluate our system, we look to answer the following questions:

- Can we satisfy the latency requirements for various publish-subscribe protocols used in industrial control systems?
- How well does our key-management scheme scale in terms of CPU and memory overheads?
- How much programmer effort is needed to convert existing publish-subscribe systems to use our scheme?
- How do we verify that the shared secret (the core-identity macaroon) between the devices and the key-management server is never leaked while establishing session keys?

Function	Semantics
Key-Management Server	
generate_macaroons(*device*, *manufacturer*)	This function generates a *nonce* and iteratively computes HMACs to generate the final Macaroon.
revoke_macaroon(*macaroon*)	Takes a macaroon as an argument, checks the validity of the macaroon, and then revokes it by blacklisting.
login_device(*core-identity macaroon*, *channel id*)	The key-management server receives the public portion of the macaroon and computes the secret key. The device is logged in if the macaroon is valid.
check_access(*message*, *channel*, *secret_key*)	When the key-management server receives a message, it checks the validity of the message on a channel based on the key. If the message was invalid, then a malicious device may have connected to the server, and the key-management server alerts the administrators.
Device	
login(*core-identity macaroon*)	When the device doesn't have any channel specific macaroons, the device logs in with its core-identity macaroon. The device presents the public part of the core-identity macaroon to the key-management server.
logout(*core-identity macaroon*)	When a device is being decommissioned or being shutdown for a planned outage, the device logs out by presenting its core-identity macaroon and authenticating itself. The key-management server revokes its channel-specific keys.
encrypt_communication(*association macaroon*, *channel_id*)	The device uses the session key to communicate with the key-management server on the private channel. If the device wants to communicate on the specific channels with other devices, the device uses the secret keys granted with the association-attribute macaroons.

Table 1: The MaQaTooT API includes methods for both of our major components. The API does not alter the basic functionalities of the MQTT implementation, but builds on top of them.

Algorithm	Key Creation time	Key Verification time
Elliptic Curves		
Ed25519-256 bits	25.79 ms	29.34 ms
Macaroons		
SHA-1-HMAC	662 μs	513 μs
SHA-256-HMAC	761 μs	566 μs

Table 2: Comparing the performance of Macaroons and Elliptic-curve keys.

4.1 Performance

To understand the performance overheads MaQaTooT introduces to MQTT clients, we performed our experiments on an ARM Firefly RK3288 development board with a 1.8 GHz processor. We wanted to test on a board that resembled an IoT device and its resource-constrained nature. We measured the CPU overheads, the memory used by our macaroon objects and the time delays in generating and verifying Macaroons and compared them to the elliptic-curve cryptography scheme proposed by Singh et al. [16].

The GOOSE protocol specification prescribes a maximum latency of 4 ms. Table 2 shows that although the elliptic-curve algorithm takes much longer to create and verify attribute certificates, SHA1 and SHA256-based HMACs take less than 1 ms to create and verify, conforming to the specifications GOOSE requires. We also found that our macaroon objects consumed only 64 bytes of memory in most of our tests.

4.2 Programmer Effort

To measure programmer effort, we evaluate how many additional lines of code are required by developers to instrument their client-side code. The client-side implementation authenticates itself with the core-identity macaroon and then receives the association-attribute macaroons. The key-management server or controller is usually replaced by an administrator in typical implementations of the MQTT protocol, and hence we do not measure lines of code needed to instrument them.

We selected two IoT applications on GitHub, and implemented them with MaQaTooT. Table 3 shows that MaQaTooT requires little developer effort to protect IoT devices and data, averaging an additional 35 lines of code.

Application	Lines of code before	Lines of code after adding MaQaTooT
Passive infrared sensor [14]	75	102
Ultrasonic sensor [5]	160	203

Table 3: Applications implemented with MaQaTooT. We added the MaQaTooT device API calls to various open source applications.

[1] Our API implementations can be found at https://github.com/prashantbarca/maqatoot.

4.3 Protocol Verification

We verified our shared-secret session-key-establishment protocol built on top of J-PAKE using the Proverif cryptographic-protocol-verification tool. Proverif is a protocol verifier that can perform reachability analysis and takes typed pi-calculus files as input. We can express the communication between the key-management server and the device in pi-calculus and query to check if a passive attacker can decipher the secret from the conversation. Abdalla et al. demonstrated the security of J-PAKE in 2015 [1].

We found that the shared secret for a specific MQTT channel is never leaked by our protocol. Figure 7 shows the result of our query on Proverif.

```
RESULT not event(evCocks) is true.
-- Query event(evCocks) ==> event(evRSA)
Completing...
Starting query event(evCocks) ==> event(evRSA)
RESULT event(evCocks) ==> event(evRSA) is true.
```

Figure 7: Proverif Verification result of our session-key management protocol.

5 RELATED WORK

Elliptic-curve cryptography for MQTT has been studied widely [4, 16]. Both the papers tackle the difficult problem of understanding how to perform elliptic-curve cryptography but fail to tackle the vital problem of key management and key revocation, and they analyze the performance only through simulations of CPU time and memory. Shin et al. make use of a pre-shared key technique using Diffie-Hellman Key Exchange [15]. Lesjak et al. propose using a hardware controller to perform Transport Layer Security client authentication [8]. The authors start with the assumption that the MQTT Server and the clients would have certificates already set up in them. Again, Lesjak et al. do not discuss key revocation or management in their paper. These implementations of public-key cryptography for MQTT ignore the issue of an active-server compromise and lead to all the communications being leaked [12]. Our paper adequately tackles both these issues.

Neto et al. make use of attribute-based and identity-based cryptography to enable authentication and authorization throughout the lifecycle of Internet-of-Things devices [13]. Since they rely on public key cryptography, they do not meet the latency and time constraints that are critical in the industrial control systems and power grid applications of publish-subscribe protocols.

6 CONCLUSIONS

In this paper, we presented MaQaTooT, a novel technique to manage keys using Macaroons in publish-subscribe IoT protocols. We demonstrated our technique on the MQTT protocol and showed that our method only requires a few additional lines of code. We adhere to the latency requirements of Industrial IoT protocols while providing authentication, integrity, and confidentiality guarantees, along with resilience to active-server compromises.

In the future, we would like to explore using a fully-homomorphic symmetric encryption scheme instead of the AES encryption we employ in this paper. We also want to apply our technique to power grid devices making use of the GOOSE protocol, to demonstrate our methods in the power grid domain. A possible addition to our device code could be a feature to verify that an adversary has not altered the system.

REFERENCES

[1] Michel Abdalla, Fabrice Benhamouda, and Philip MacKenzie. 2015. Security of the J-PAKE password-authenticated key exchange protocol. In *IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, USA, 571–587. https://doi.org/10.1109/SP.2015.41

[2] Prashant Anantharaman, Kartik Palani, David Nicol, and Sean W Smith. 2016. I Am Joe's Fridge: Scalable Identity in the Internet of Things. In *Proceedings of IEEE International Conference on Internet of Things (iThings)*. IEEE, Chengdu, China, 129–135. https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData.2016.47

[3] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Proceedings of Network and Distributed System Security Symposium*. San Diego, CA, USA.

[4] Abebe Abeshu Diro, Naveen Chilamkurti, and Prakash Veeraraghavan. 2017. Elliptic Curve Based Cybersecurity Schemes for Publish-Subscribe Internet of Things. In *Proceedings of International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer International Publishing, Dalian, China, 258–268.

[5] Alex Ellis. 2017. MQTT sender/receiver for PIR sensor. https://github.com/alexellis/iot-mqtt-pir. Accessed: 2019-May-10.

[6] Feng Hao and Peter Y. A. Ryan. 2011. Password Authenticated Key Exchange by Juggling. In *Security Protocols XVI*, Bruce Christianson, James A. Malcolm, Vashek Matyas, and Michael Roe (Eds.). Springer, Berlin, Heidelberg, 159–171.

[7] Nishchal Kush, Ejaz Ahmed, Mark Branagan, and Ernest Foo. 2014. Poisoned GOOSE: Exploiting the GOOSE Protocol. In *Proceedings of the Twelfth Australasian Information Security Conference - Volume 149 (AISC '14)*. Australian Computer Society, Inc., Darlinghurst, Australia, 17–22. http://dl.acm.org/citation.cfm?id=2667510.2667513

[8] C. Lesjak, D. Hein, M. Hofmann, M. Maritsch, A. Aldrian, P. Priller, T. Ebner, T. Ruprechter, and G. Pregartner. 2015. Securing Smart Maintenance Services: Hardware-security and TLS for MQTT. In *Proceedings of 13th International Conference on Industrial Informatics (INDIN)*. IEEE, Cambridge, UK, 1243–1250. https://doi.org/10.1109/INDIN.2015.7281913

[9] Roger A Light. 2017. Mosquitto: Server and Client Implementation of the MQTT Protocol. *The Journal of Open Source Software* 2, 13 (2017), 265.

[10] Sergey Lyubka. 2016. Why MQTT is getting so popular in IoT. https://mongoose-os.com/blog/why-mqtt-is-getting-so-popular-in-iot/. Accessed: 2019-May-09.

[11] John Matherly. 2015. Complete Guide to Shodan. https://www.shodan.io/. Accessed: 2019-May-10.

[12] R. Neisse, G. Steri, and G. Baldini. 2014. Enforcement of Security Policy Rules for the Internet of Things. In *Proceedings of 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, Larnaca, Cyprus, 165–172. https://doi.org/10.1109/WiMOB.2014.6962166

[13] Antonio L. Maia Neto, Artur L. F. Souza, Italo Cunha, Michele Nogueira, Ivan Oliveira Nunes, Leonardo Cotta, Nicolas Gentille, Antonio A. F. Loureiro, Diego F. Aranha, Harsh Kupwade Patil, and Leonardo B. Oliveira. 2016. AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM (SenSys '16)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2994551.2994555

[14] Mark Paluch. 2015. IoT Distancemeter. https://github.com/mp911de/iot-distancemeter. Accessed: 2019-May-10.

[15] S. Shin, K. Kobara, Chia-Chuan Chuang, and Weicheng Huang. 2016. A Security Framework for MQTT. In *Proceedings of Conference on Communications and Network Security (CNS)*. IEEE, Philadelphia, PA, 432–436. https://doi.org/10.1109/CNS.2016.7860532

[16] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar. 2015. Secure MQTT for Internet of Things (IoT). In *Proceedings of Fifth International Conference on Communication Systems and Network Technologies*. IEEE, Gwalior, India, 746–751. https://doi.org/10.1109/CSNT.2015.16