Qrkey: Simply and Securely Controlling Swarm Robots

Alexandre Abadie, Mališa Vučinić, Diego Badillo, Said Alvarado-Marin, Filip Maksimovic, Thomas Watteyne

Qrkey: Simply and Securely Controlling Swarm Robots

Alexandre Abadie*, Mališa Vučinić*, Diego Badillo*†,
Said Alvarado-Marin*, Filip Maksimovic*, Thomas Watteyne*

* Inria, France
† Universidad Técnica Federico Santa María, Chile
e-mail: first.last@{inria.fr*,sansano.usm.cl†}

***Abstract*—Swarm robotics is a research field on robotic collaboration where large number of robots are deployed to complete collective real-world tasks. Progress made in the field now tends to be democratized through swarm deployments such that users with different background can learn, program, conduct research or just play with them. One of the main challenges for swarm operators though is to keep the infrastructure at reasonable complexity, maintenance level, and price, to allow occasional users to interact with many robots smoothly. In this paper, we present Qrkey, an open-source library which provides an engaging, accessible and trustworthy user experience to control and track robots in a swarm. After a detailed presentation of the primitives and the protocol proposed by Qrkey, we discuss the security concerns raised by this solution. Finally, we conclude by giving Qrkey limitations and possible future work.**

***Index Terms*—swarm robotics, user experience, MQTT, security**

I. Introduction

In the context of robotic swarms for education, research, and remote control, significant overhead from complex networking infrastructure prevents operators and users from deploying robots and from easily manipulating existing deployments. For example, how can a swarm operator easily allow visitors to control swarm robots with their own smartphone as they are coming by a swarm deployment? Another example: how can a teacher show up in a classroom with a couple of robots, setup the swarm quickly and let the students control them with their phones? What teams operating a robot swarm typically have to do is *(1)* either have users use a dedicated interface (e.g. a tablet) that runs all the right software and is pre-configured to connect to the robots, or *(2)* setup complex, centralized and costly infrastructure (e.g. a dedicated WiFi access point) that users have to connect to with their phone.

Without loss of generality, we call "researcher" the person who is operating the swarm, and "visitor" the occasional visitor. We call "interact with the swarm" any action by which the visitor controls one of more robots, or simply looks at a display of their internal state.

This paper introduces Qrkey, an open-source library that solves this lack of simple user experience. It spares the researcher from having to install heavy infrastructure such as complex network configurations and WiFi access points. It allows visitors to use their own phone to interact with the robots, without installing dedicated software, and without them having to change their connection settings.

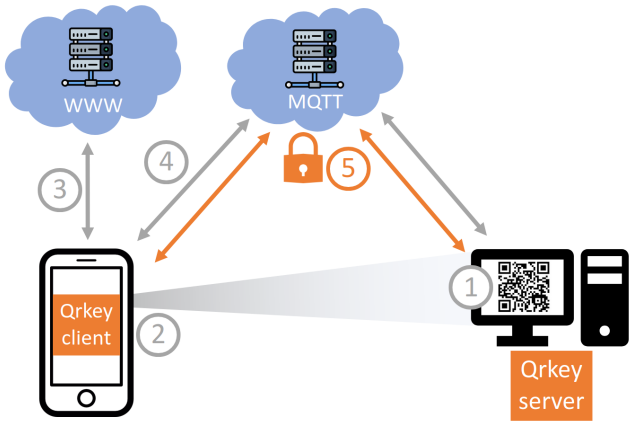

Fig. 1. The Qrkey user experience. ① The Qrkey server displays a QR code on a screen. ② A user flashes that QR code using their smartphone. ③ The smartphone fetches a static page containing the Qrkey client library. ④ The Qrkey client library connects to a public (non-secure) MQTT broker. ⑤ The Qrkey client and server establish a secure session using the Qrkey protocol. **This is achieved without requiring any dedicated infrastructure**.

Qrkey consists of two parts: the Qrkey client (which runs on the visitor's device) and the Qrkey server (running on the robots, or on an element controlling them directly). The Qrkey client and server communicate using the Qrkey protocol, which is built on top of Message Queuing Telemetry Transport (MQTT), a popular and standardized protocol widely used in IoT applications. An MQTT administrator starts and maintains a broker instance on a computer on the Internet. Users then connect to that broker and can publish/subscribe to topics. MQTT has the option of using the Transport Layer Security (TLS) protocol to secure the communication between each of the clients and the broker. Messages are protected (encrypted and authenticated) by the TLS protocol, but, the broker has visibility of message contents as it terminates the TLS connection. The standardized way of securing MQTT requires a trusted administrator to manage the MQTT broker, in particular, to provide credentials to each user. The goal of Qrkey is to operate with a public MQTT broker (many

are available free-of-charge, such as HiveMQ[1]) so that no dedicated infrastructure is required, and so that the user experience is smoother without requiring credentials. Use of a public broker creates a security vulnerability because any user of the public broker can subscribe to any topic and receive any message published by other users. In order to secure message contents and to mitigate this concern, Qrkey implements a shared-key payload encryption scheme.

The contributions of this paper are three-fold.

- We describe the design of the Qrkey protocol. We detail how we use recent standards-based solutions for its security features. We provide a detailed security assessment.
- We provide a full implementation of the Qrkey client and Qrkey server as two open-source libraries which are ready to be integrated in swarm robotics environments.
- We show what a user can expect in terms of performance, and describe how we use Qrkey to control the DotBot swarm.

II. Related Work

A. Platforms for Research & Education

Over the past decade, a number of platforms have emerged on which to conduct research, and to teach.

The Robotarium [1] is a permanent testbed operated and hosted by Georgia Tech. It consists of a large table with several dozen robots called "GRITSBots" that users can remotely program and control. The robots themselves are WiFi-connected with an Xtensa ESP8266 microcontroller, and all of the hardware and software is available open-source. In order to program and control the robots, a user can upload their code onto the Robotarium web interface and a simulator is used to verify that the resulting user experiment will not result in significant robot collisions. This web interface is the only interface for remote robot control.

IoT-lab [2] is a large testbed for IoT networking research. IoT-lab consists of 2728 low-power wireless devices deployed around 6 sites. Each device consists of a low-power wireless device attached to a Raspberry Pi-like single board computer. Users upload their firmware on the web interface of the platform, after which the manager software flashes the user-specified devices. This is mainly used by researchers conducting experiments on wireless protocols using IEEE 802.15.4, Bluetooth Low-Energy and LoRA physical layers and hardware. While approx. 30 robots were deployed at the site in Lille in Northern France, support has been dropped in 2022 as it was technically too complex to automate recharging and localization.

The SmartSantander testbed [3] is a real-world experimental deployment of thousands of environmental sensors, many of which are wirelessly connected (with IEEE802.15.4 radios), in Santander, Spain. Users program the wireless nodes over-the-air.

The OpenTestbed [4] is a minimalistic testbed environment that allows users to deploy their own instance. By using simple Raspberry Pi boards connected over WiFi to the Internet, the 81-node instance at Inria Paris has cost less than 9,000 EUR to install, including the cost of the low-power wireless boards. The protocol used by the OpenTestbed to coordinate between devices builds on top of MQTT. That protocol is the foundation of the Qrkey protocol.

B. MQTT Security

In literature, the researchers have proposed different ways of securing MQTT end-to-end between the publishers and the subscribers by payload encryption.

Park et Nam [5] propose a security architecture that supports access control and end-to-end security between subscribers and publishers. They introduce the concept of a *topic certificate* that together with a device certificate, provisioned into each device, is used to establish mutual authentication and for session key exchange. Their solution, however, requires complex identity and certificate management, making it contradictory to our minimal setup requirement.

Rizzardi *et al.* [6] have proposed end-to-end security mechanisms based on attribute-based encryption, that can be decrypted only by clients satisfying the access policy. This again necessitates maintaining a complex infrastructure.

The work similar to ours is that of Sadio *et al.* [7], where a shared symmetric key is used for payload encryption and authentication. The authors, however, do not elaborate on how the shared key can be provisioned nor what are the security properties or the adversarial model.

While MQTT has all the features to ensure secure communication, in our use case, using these security features would require the researcher to manage the MQTT broker, in particular to enter credentials of each client. Instead, we would like to use public MQTT brokers, on which MQTT security features are not available. The Qrkey protocol therefore implements its own security features, in essence resulting in protecting (encrypting and authenticating) the payload that transits over a non-secure MQTT broker.

III. Qrkey

A. Requirements

While the requirements have already been discussed in Section I, we here provide a more formal list.

1) Simple User-experience.
2) System segregation. Controlling devices in one swarm should not impact another. This impacts the choices made when using MQTT.
3) No dedicated infrastructure. Of course, an MQTT broker is infrastructure, but since it's a public broker operated by a third party, we consider it doesn't represent dedicated infrastructure.
4) End-to-end security. We consider a honest-but-curious trust model for message delivery. We consider the group members in a deployment as trusted. Messages must only be readable by the members of the deployment (group-level confidentiality). An attacker outside of the group must not be able to tamper with the message.

[1] https://broker.hivemq.com

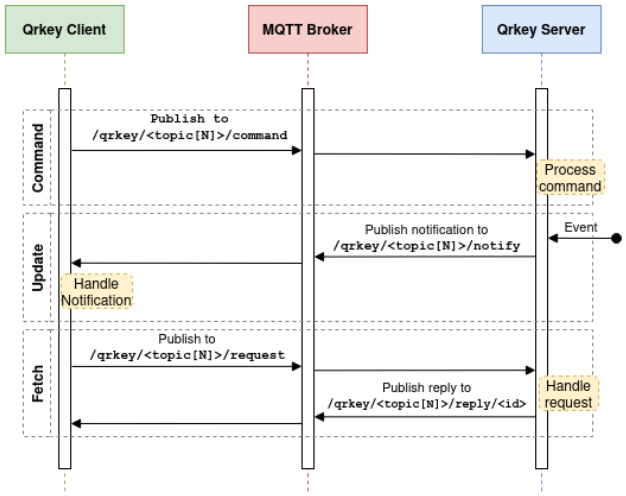

Fig. 2. The Qrkey protocol allows for secure request/reply, command and notification primitives between the Qrkey client and Qrkey server, over an otherwise unsecured public MQTT broker.

Group members must have a way of verifying whether the received message is fresh.

B. User Experience

Fig. 1 illustrates the user experience Qrkey allows, and which consists of five steps:

① A robot, or a gateway device that controls all robots, runs a Python program which includes the Qrkey server library. The device has means of displaying a QR code on a screen. In the case of a Raspberry Pi, this just consists in connecting a regular computer screen to it. The QR code encodes the URL of a static webpage, for example hosted by GitHub pages [2], such as `https://dotbots.github.io/PyDotBot?pin=1234`. Here the string "1234" is the pin used by the protocol (detailed below).

② Visitors who want to interact with the robots uses their phone to scan the QR code.

③ Their phone's browser opens the static page, which loads the Qrkey client library.

④ The Qrkey client library connects to a public MQTT broker.

⑤ Through the MQTT broker, to which the robot has previously connected, the Qrkey client and Qrkey server establish a secure session through the Qrkey protocol.

C. Primitives

We implement the following communication primitives over MQTT, as shown in Fig. 2:

- **Command**. A Qrkey client sends a command to the Qrkey server.
- **Update**. The Qrkey server broadcasts a notification to all Qrkey clients.

[2] The popular source code hosting service can also host static webpages, free of charge.

TABLE I
QRKEY TOPICS USED BY THE QRKEY PROTOCOL

`/qrkey/<topic>/command`
`/qrkey/<topic>/notify`
`/qrkey/<topic>/request`
`/qrkey/<topic>/reply/<client id>`

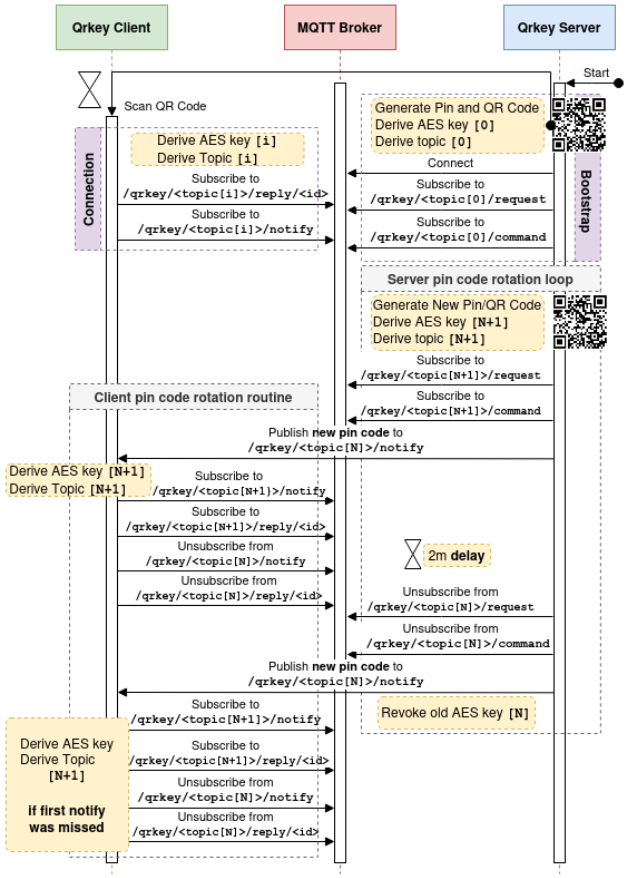

Fig. 3. The Qrkey protocol, including the connection phase between Qrkey client and Qrkey server, and pin code rotation.

- **Fetch**. A Qrkey client requests information from the Qrkey server, which answers.

Table I explicitly lists the MQTT topic structure used for the type of primitive.

The *Fetch* primitive is implemented as follows. The Qrkey client publishes a message on topic `request`; to which the Qrkey server is subscribed to. The payload of a request is a JSON message with a *request* field corresponding to the type of request, and a *reply* field corresponding to the unique identifier of the MQTT client that published the request. When the Qrkey server receives the request, it uses the *reply* field to determine the topic to publish the reply to. This strategy guarantees a point-to-point request scheme between a Qrkey client and the Qrkey server.

D. The Qrkey Protocol

Fig. 3 illustrates the Qrkey protocol. At startup, the Qrkey server generates an L-digit pin code. The length of the pin code *L* is set to 12, which results in a 39.86 bit entropy which is just above the recommended randomness for high security passwords by [8]. The pin code entropy is computed using $\log_2(R^L)$, where *R=10*, the number of possible values for a digit, and *L=12*, the length of the pin code. This pin code is meant to be shared and is encoded into the QR code.

The pin code is used by both the Qrkey client and the Qrkey server to feed a Key Derivation Function (KDF), producing two pieces of information:

1) **a unique topic**. A base64 string representing an 128-bit key used for the root topic in the form `/qrkey/<base64 string>` specific to one running Qrkey deployment at a time,
2) **an AES symmetric key**. A 256-bit key used for encryption/decryption of the payload of all MQTT messages exchanged over the MQTT broker.

The payload of all packets is encrypted using the JOSE JSON Web Encryption (JWE) standard protocol [9], with the following parameters: *algorithm* is a direct use of the 256-bit symmetric key, *encryption method* is AES GCM. The underlying JWE library takes care of selecting a unique nonce that is used for encryption and included in the message. To protect against replay attacks, all payloads exchanged contain an authenticated `timestamp` field that is checked upon reception.

To prevent an attacker from brute forcing the pin code and actively interfering with the ongoing session, shared pin codes are changed every 15 minutes. We use an overlap period of 2 minutes during which the keys derived from the N^{th} and subsequent $(N+1)^{th}$ codes are both valid. At the beginning of the rotation, the server subscribes to the new $(N+1)^{th}$ based topics (for commands and requests) and notifies connected clients on the old N^{th} based topics. When a new Qrkey client joins, it uses the $(N+1)^{th}$ pin to derive $(N+1)^{th}$ topics and AES key. At the end of the overlap period, the Qrkey server unsubscribes from the N^{th} topics, invalidates the N^{th} AES key, and once more sends the $(N+1)^{th}$ pin to all clients still using the N^{th} topics and AES key.

E. Security Discussion

The security of Qrkey relies on a shared symmetric secret and on payload encryption. It is therefore not suitable for more powerful adversarial models other than the honest-but-curious message broker.

In particular, the solution is susceptible to injection or replay attacks inside the group. Any compromised or misbehaving member of the group could inject messages on different topics without this being noticeable by the group members.

There is no explicit authentication among the MQTT clients. This means that an attacker can subscribe to any topic and be handed a copy of the (encrypted) message by the broker. The confidentiality of the message is ensured by symmetric encryption, with the symmetric key being derived from the pin code. Security against offline brute force attacks is therefore dependant on the pin code entropy.

In order to actively interfere with the group, the attacker would need to guess the pin code and derive the symmetric key. Due to the pin code rotation, the attacker has a time limit for doing this. However, even after the pin code becomes obsolete, if the attacker succeeds in guessing the pin code, it is able to decrypt the message from a previous rotation.

Only the payload of MQTT packets is protected. This means that an active attacker can launch a series of attacks using the control MQTT packets, such as disconnect a client, or unsubscribe it from a given topic.

IV. Implementation

The Qrkey implementation is published under an open-source BSD licence[3]. The Qrkey server implementation, written in Python, is provided as a Python package on PyPI [4]. The Qrkey client implementation, written in Javascript, is provided as a Javascript NPM package [5]. This package implements ReactJS hooks to connect and communicate with an MQTT broker, as well as cryptographic primitives and the pin code rotation routines.

V. Expected Performance of Qrkey

We qualify the performance by two metrics. First, the reliability is the portion of messages received over the total number of messages transmitted. Second, the latency is the duration between a message being transmitted, and it being received. Because we are using an MQTT broker, which typically runs on the Internet, far away from the swarm, the major contributing factor to reliability and latency is the performance of the Internet connection between the Qrkey client and the broker, and between the broker and the Qrkey server.

To quantify reliability and latency, we conduct an experiment using the public HiveMQ broker. At the time of writing, the dashboard of the broker[6] indicates 61,532 clients connected, 284,260 active subscriptions, and 27,646 messages queued over the last 2 min. These large numbers indicate that the broker will *not* somehow overload when sending a handful of messages per second.

To measure the reliability and latency while using HiveMQ, we write a Python script which publishes timestamped 20 character-long packets every 500 ms, and measure the time that it takes for receiving the same packet after having subscribed to the topic it is published on. Each packet contains a unique sequence number, allowing us to count lost packets, i.e. reliability. Results are shown in Fig. 4, for an experiment lasting 15 h. 108,471 messages were published. None were lost, translating to 100% reliability. On average, the latency

[3] As an online addition to this paper, the implementation of Qrkey is published under an open-source BSD license at https://github.com/dotbots/qrkey.

[4] https://pypi.org/project/qrkey

[5] https://www.npmjs.com/package/qrkey

[6] https://www.mqtt-dashboard.com/

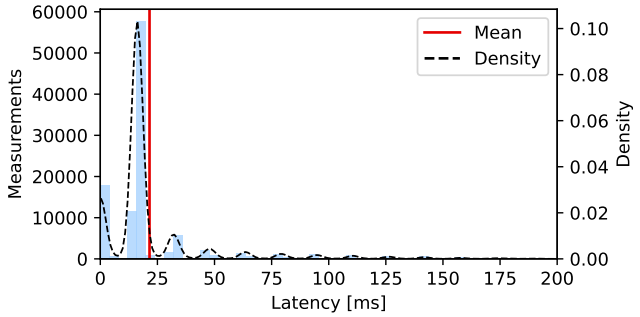

Fig. 4. Latency that can be expected when using Qrkey on the HiveMQ public MQTT broker.

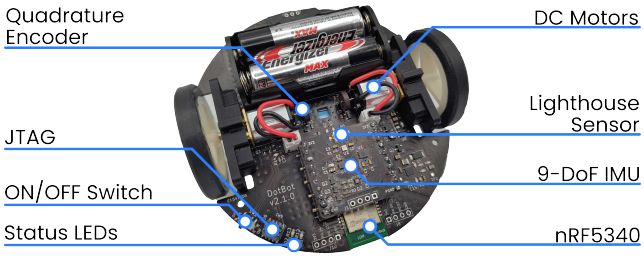

Fig. 5. The DotBot swarm robotic platform features an nRF5340 micro-controller driving two wheel motors, and interfaced to a lighthouse localization system.

is 21.55 ms. 1.194% of the packets exhibit a latency larger than 200 ms. During the worst case, 70 messages in a row exhibited a latency larger than 200 ms.

These measurements indicate that the performance of using a public broker such as HiveMQ is probably sufficient for simple interactions between visitors and robots.

VI. Using Qrkey for the DotBot

The DotBot, pictured in Fig. 5, is a simple driving robot which features an nRF5340 micro-controller with low-power wireless communication capabilities, as well as two-motor-two-wheel propulsion, and a lighthouse-based localization system. We are building up a swarm of 1,000 DotBots. The nRF5340 features two ARM Cortex-M33 cores – the application core and the network core – with 256 kB of RAM and 1 MB of flash memory. The application core runs at 128 MHz, the network core at 64 MHz. The nRF5340 also includes two quadrature decoder (QDEC) peripherals, which enable it to measure the DotBot's wheel velocity and allows for precise odometry of the robot. The integrated 2.4 GHz radio supports commonly used Internet of Things (IoT) protocols, such as: BLE (2 MBps to LR 125 kBps), IEEE 802.15.4, and Enhanced ShockBurst (ESB). We use the embedded ARM CryptoCell peripheral in the nRF5340 to implement standard cryptographic protocols, ensuring the security of radio communication and over-the-air firmware updates. All hardware and firmware is published[7] under an open-source BSD license.

[7] https://github.com/DotBots

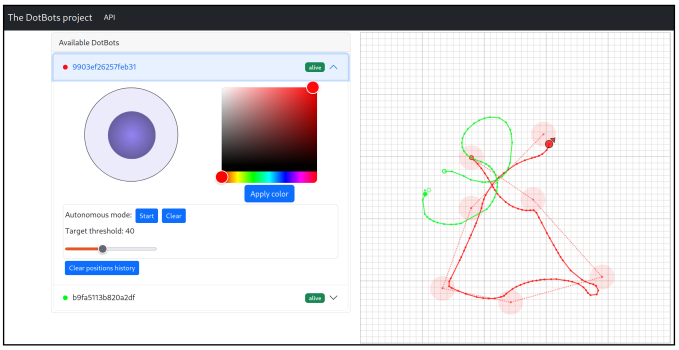

Fig. 6. Screen capture of the interface to control a swarm of DotBots, which uses Qrkey. The buttons on the left allow the visitor to move the DotBot and set the color of its LED. The map on the right displays the current position and the trajectory of each DotBot.

We call `PyDotBot` the software to control and track a swarm of DotBots. `PyDotBot` is written in Python and JavaScript and runs on a Raspberry Pi, connected to both an nRF52840-DK low-power wireless radio dongle, and a screen. The Qrkey server library is part of `PyDotBot`, which allows it to go from unboxing to having a swarm robot fully functional running in less than 15 min. The static web page connecting to the `PyDotBot` is hosted on GitHub pages[8]. Built on top of Qrkey, `PyDotBot` implements an MQTT interface allowing a Qrkey client to control the left and right motor drivers of the DotBot and set the color of its RGB LED. In addition, `PyDotBot` displays the position of each DotBot on a map, which is displayed on the Qrkey client. By clicking on that map, the Qrkey client can set a lot of waypoints and have the DotBot follow that route. A Qrkey client uses the *Fetch* primitive to retrieve the list of the currently available DotBots in the swarm. The Qrkey server issues notifications to the Qrkey client when a new DotBot enters or leaves the swarm, or when a DotBot updates its position. Fig. 6 is a screen capture of the interface the visitor interacts with.

In summary, `PyDotBot` implements a full interface for a visitor to interact with the swarm of DotBots, and is implemented on top of Qrkey, which allows a visitor to interact with the DotBots using their own phone, without any dedicated infrastructure.

VII. Conclusions

In this paper, we presented Qrkey, a library providing seamless communication over MQTT between robotic swarm users and a swarm deployment provider. We have described in detail the architecture, the primitives and the securing protocol implemented in Qrkey, have discussed the security concerns addressed by Qrkey and have given some performance metrics when using a public MQTT broker such as HiveMQ. Finally, we detailed an example of a Qrkey usage within the DotBot ecosystem to demonstrate, in practice, the possible swarm

[8] https://dotbots.github.io/PyDotBot

interactions offered by Qrkey based deployments as well as their ease of use.

Despite the high level of security at deployment level, e.g. between groups of users of a single swarm deployment, Qrkey still suffers security issues within the deployment group. Indeed, all users in the group are considered trusted and if a user in the group is compromised, Qrkey has no revocation mechanism for this user at the moment.

This limitation could be mitigated by adding end-to-end security at the session level between a Qrkey client and the Qrkey server such as the Messaging Layer Security (MLS) Protocol [10]. The Qrkey server would be in charge of maintaining the list of all Qrkey clients session keys and would be able to revoke any compromised client at any time, without compromising the deployment group integrity. Fully designing and implementing this is part of our future work

ACKNOWLEDGMENT

This document is issued within the frame and for the purpose of the OpenSwarm project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046. Views and opinions expressed are however those of the author(s) only and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt, "The Robotarium: A Remotely Accessible Swarm Robotics Research Testbed," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[2] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed," in *IEEE World Forum on Internet of Things (WF-IoT)*, Milan, Italy, December 2015.

[3] L. Sánchez, L. Muñoz, J. Galache, P. Sotres, J. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and D. Pfisterer, "SmartSantander: IoT Experimentation over a Smart City Testbed," *Computer Networks*, January 2013.

[4] J. Muñoz, F. Rincon, T. Chang, X. Vilajosana, B. Vermeulen, T. Walcarius, W. van de Meerssche, and T. Watteyne, "OpenTestBed: Poor Man's IoT Testbed," in *Workshop on Computer and Networking Experimental Research using Testbeds (CNERT)*, 2019, pp. 467–471.

[5] C.-S. Park and H.-M. Nam, "Security Architecture and Protocols for Secure MQTT-SN," *IEEE Access*, vol. 8, pp. 226 422–226 436, 2020.

[6] A. Rizzardi, S. Sicari, D. Miorandi, and A. Coen-Porisini, "AUPS: An Open Source Aathenticated Publish/Subscribe System for the Internet of Things," *Information Systems*, vol. 62, pp. 29–41, 2016.

[7] O. Sadio, I. Ngom, and C. Lishou, "Lightweight Security Scheme for MQTT/MQTT-SN Protocol," in *International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 119–123.

[8] D. E. Eastlake, S. Crocker, and J. I. Schiller, "Randomness Requirements for Security," RFC 4086, 2005.

[9] M. B. Jones, "JSON Web Algorithms (JWA)," RFC 7518, 2015.

[10] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol," RFC 9420, 2023.