# Towards improving the Privacy in the MQTT Protocol

Marten Fischer, Daniel Kümper, Ralf Tönjes
University of Applied Sciences Osnabrück, Germany
E-mail: {m.fischer, d.kuemper, r.toenjes}@hs-osnabrueck.de

***Abstract*—The Internet of Things (IoT) is growing rapidly as more and more household appliances, sensors and actuators are connected to the internet and communicate with each other. Because these IoT devices usually have only limited processing and communication capabilities, an efficient communication protocol is required to reduce the protocol overhead. The Message Queue Telemetry Transport (MQTT) protocol provides such properties. MQTT is a publish/subscribe protocol, where each client can subscribe to a message topic in order to receive all messages published under that topic. The payload of a message (i.e., the content) can be encrypted to hide private information. However, to forward messages, the topic of a message needs to be read by the broker and thus cannot be encrypted and might reveal private information. This paper presents methods to avoid this problem. It features and evaluates a one-time password approach to provide a fully obfuscated communication method for MQTT topics. Thereby, the user tracking and the generation of profiles is prevented.**

## I. Introduction and Motivation

The Message Queue Telemetry Transport (MQTT) protocol was developed by Andy Standford-Clark and Arien Nipper. They wanted a lightweight protocol for machine to machine communication of embedded devices, sensors or actuators. MQTT is a protocol, where clients can subscribe at a central broker in order to receive a specific type of message identified by the message's topic [3]. The protocol is designed to be used on resource-constrained devices with little computational power and slow network connectivity. For example, the protocol overhead when publishing a message can be, in best cases, only 2 bytes. MQTT provides a simple Quality of Service (QoS) mechanism for the reliable delivery of messages, if necessary. The MQTT standard defines 3 QoS levels: 0 - the message will be delivered at most once ("fire and forget"); 1 - the message will be delivered at least once; 2 - the message will be delivered exactly once. A very useful feature when using the protocol in an IoT environment, where devices might lose the connectivity to the broker more often, is the "Last Will" feature. That is, a client can deposit a message at the broker, which will be sent out if the connection between the client and the broker is interrupted. This way other clients can get notified about the status of a device, even when it is not connected to the broker.

Messages are forwarded based on the message topic they are published with. Thus, a message topic identifies the message type. In this paper, we will further use multiple message topics to designate a single message type. A message topic is composed of different topic levels delimited by a slash ("/") and represented as a string. This allows to define message topics in a hierarchical way and enables the usage of wildcards for a subscribing client. The MQTT standard defines two wildcards, the hash sign # and the plus sign +. The plus sign matches exactly a single topic level whereas the hash sign includes multiple topic levels. Wildcards in message topics are only allowed for subscriptions. Recently, the MQTT protocol has become quite popular. Because of that, publicly available brokers are provided by a number of organisations. In [2] a list of publicly available brokers is provided. Since MQTT clients establish the connection to the broker, the use of a publicly reachable broker (i.e. a broker with a public IP) also makes it more easy to exchange messages from a private network to clients outside of the network. Hence, devices behind a network address translated (NATed) network pose no problem and therefore no re-configuration of the network is necessary.

**Problem:** The message topics need to be readable for the broker in order to be capable to decide to whom the messages are forwarded to. This allows the broker to analyse the messages even without access to their content, extract privacy-related information, and create user profiles. For example, a message published under the topic *livingroom/switch/off* might allow to deduce sleeping habits. Just knowing a switch was toggled is private data.

However, we argue that the message topic does not need to be meaningful in the sense that it contains real words which identify its type. As long as all clients use the same topic(s) for a specific message type, the topic can be obfuscated. To reduce the ability of the broker to deduce any private information, the used message topics need to appear random for any outsider. This raises the question, how two or more clients can agree on an obfuscation scheme without any third party to know about it. This is the motivation of this paper, which will present five strategies to achieve above-mentioned obfuscation. The most promising approach was implemented and experimentally validated to prove the applicability in real-world applications. Although the MQTT protocol was designed with resource-constrained devices in mind, the strategies here will require additional computational power and memory. In our evaluation, we show that the presented strategy delays the message exchange only insignificantly compared to the network latency. The strategies shall allow the usage of a publicly available MQTT broker. Therefore, all of them will be compatible with the MQTT standard. The focus on this paper

978-1-7281-2171-0/19/$31.00 

is solely on the message topics, not on the payload. Methods to protect the payload of MQTT messages have been developed in the past already (see section II).

The rest of the paper is organised as follows: Section II gives a brief overview of secure communication using the MQTT protocol. Section III will introduce different strategies to obfuscate message topics, which at the same time still allow the broker to forward the messages appropriately. In section IV we present the results of our evaluation and section V concludes this paper.

## II. Related Work

The authors of [9] propose a Secure MQTT (SMQTT) protocol based on the Attribute-Based Encryption (ABE) scheme. They use a lightweight elliptic curve to address resource limitations of IoT devices. Also, they employ a hybrid encryption, where the payload itself is encrypted using the Advanced Encryption System (AES) and the AES keys are encrypted using ABE. However, only the payload is encrypted. The message topics are still readable by the message broker. In [7], the authors highlight the need to secure the data generated by IoT devices in order to protect the privacy of citizens/users. For this, they developed a solution to enforce security policy rules called SecKit. The enforcement is integrated into the MQTT protocol layer. The solution must be integrated at the broker and clients need to trust this broker. In contrast, this paper describes methods which are employed on the client side only, hence any (even untrusted) public MQTT broker can be used. The authors of [10] used Access Control Lists to control the message flow between clients in a smart home context. For this, they used a feature provided by Mosquitto, a popular open source MQTT broker implementation. The transferred data is secured by SSL, but not end-to-end. That means the broker is still able to read message topics and even the content. Calbaretta [4] and Shin [8] employ the augmented PAKE (Password-only Authentication and Key Exchange) algorithm to secure the communication between MQTT parties. The algorithm is used to exchange session keys between the broker and the clients. Like [10], this solution focuses on securing the communication between the broker and the client but still allows the broker to read message topics. Kraijak et al. describe a scenario, where a malicious user publishes messages under a topic with faulty data, which was then consumed by others [6]. However, they do not provide a solution but rather see the security and permission policy as a point for future discussion.

## III. Privacy enhancing methods for MQTT

This section will present five methods to enhance privacy when clients publish messages using the MQTT protocol over a public broker. All methods do not require any modifications to the MQTT standard and therefore can be used with any publicly deployed MQTT broker.

### A. Redefinition in the Payload

The first method to obfuscate the topic of a message is to redefine a new topic every time such a message is published.

| Message 1 | |
|---|---|
| Topic | room/light/switch |
| Payload | {t: ”jkdsf/ertef/bvfdg”, p: [payload] } |
| **Message 2** | |
| Topic | jkdsf/ertef/bvfdg |
| Payload | {t: ”knshd/vbcfdd/bvouqw”, p: [payload] } |

TABLE I: Two consecutive Messages with Obfuscated Topic

The new topic could be stated in an encrypted payload, thus a broker will not have knowledge about the new, secret topic a particular message is exchanged.

Table I illustrates an example where first message is published under the topic *room/light/switch*. The payload contains a JSON object with the element *p* containing the actual payload and an element *t* stating the topic under which the next message of this type will be published, in this case *jkdsf/ertef/bvfdg*. The new topic can be generated using a pseudo-random generator. The receiving client(s) only need to subscribe to the new topic. The second message is published under the topic as stated in message 1 and contains again a new topic, which will be used for the subsequent message of this type. To ensure that each message, including the new topic, is received an appropriate QoS level must be used.

This method has a couple of disadvantages. Any client, who wants to receive messages of a certain type, but missed the first one, is not able to get knowledge about the randomly chosen topics that followed. Furthermore, a broker could guess that a subscription to the topic *jkdsf/ertef/bvfdg* by a client, who just received a message with the topic *room/light/switch* means, that both message topics indicate the same message type (assuming the broker has knowledge about this method of obfuscation). Furthermore, as the real payload is embedded into a JSON object, any binary content needs to be encoded. Using, for example, the Base64 code, the size of the content will be increased by 33%. An additional increase in message size results from the fact, that the new topic needs to be transferred each time a message of this type is published.

### B. Synchronous Random Topics

In this method, two or more clients have an identical Pseudo-Random Function (PRF), which needs to be synchronised using the same seed. In case only two clients want to exchange information, the seed can be exchanged for example using the Diffie-Hellman key exchange (DHKE) method. Otherwise, any form of encryption can be used, such as symmetric encryption (e.g., AES) with a pre-shared key. For different message types a separate PRF is required. The subscribing client uses the random values to generate a new topic and subscribe to them at the broker. The publishing client uses the PRF to generate the same topic for a message type (as the PRFs are synchronized they produce the same output) and then publishes the message. After the receiving client has received the message, he generates the next topic for a message of the specific type and subscribes to this topic at the broker. This method does not increase the size of a message, assuming the obfuscated topic length is identical to the original one, but additional messages need to be exchanged to synchronise the

PRFs and for the recurring subscriptions. Of course, a broker could detect such behaviour. A subscription directly after the reception of a message, and thus classify different topics to an identical message type. In order to make it harder for the broker, the receiving client could perform the subscription after a certain delay in case it can estimate the time between two successive messages of a certain type.

This method works best if only two clients want to exchange data, as they can use the DHKE method. Otherwise, some encryption information such as symmetric keys or public keys need to be exchanged beforehand, which can be problematic in the IoT environment. Another drawback is that the broker can detect the behaviour described in this method if the subscribing client re-subscribes to a new topic every time it receives a message and thus is able to classify all topics to the message type.

### C. Mass Random

This method extends the method presented in section III-B. Again the clients share an identical PRF and exchange the seed $s$ for this function securely. In addition, the constants $c_t$ are chosen for each message type $t$ intended to be exchanged between the clients. The constants are shared along with the seed $s$ with all participating clients. Now, the clients create sets of obfuscated topics $S_t$ such that $|S_t| = c_t$ for each message type $t$ using the synchronised PRF. If the number of message types is small, additionally a set $S'$ of dummy topics can be added. Here, each level in the random topics should not differ in length, so that no classification by length leads to a specific message type. This can be achieved using a hash algorithm $F$ and dummy placeholders for topic levels to always the topics with the same number of levels. Of course, the algorithm to create a placeholder and $F$ must be known by all participating clients.

The elements of all sets $S_t$ and $S'$ are now merged into one set $S = S' \cup S_t \forall t$, rearranged randomly and subscribed to at the broker. After subscription, the set $S'$ can be discarded. The sets $S_t$ are required to map an obfuscated topic to the original topic. The publishing client generates $S_t$ and picks a random element from it. This obfuscated topic is then used to publish the message. The recipient can look-up the obfuscated topic in the $S_t$ sets to find the original topic. In other words, the sets $S_t$ contain synonymic topics for the topic $t$. The sizes of the sets ($c_t$) should be chosen appropriately depending on the frequency a message of type $t$ is published, to reduce the risk of using a synonymical topic too often.

This method eliminates the problem of re-subscribing every time a message of a specific type has been received. The subscription to multiple message types in a random way, the use of placeholders so all topics look similar and using dummy topics make it hard for the broker to get any information during subscription. At runtime, the broker might classify obfuscated topics to message types over time by analysing the frequency of their usage or typical times they are used. As the strategy in III-B, the participating clients are required to synchronise their PRFs beforehand.

### D. One-Time Password (OTP)

This method is inspired by one-time password techniques, to create topics seeming random. Here, we assume two or more clients can communicate securely, e.g. using a pre-shared key to encrypt messages. Also, each client uses the same hash function $F : * \rightarrow \{a-z, A-Z, 0-9\}^*$. At first, the sender of messages prepares a list of hashes $H$ for each topic $t \in T$ intended to be used to publish messages. The first entry in the list is the hash value of a random string $s$, each following entry in the list is the hash value of the predecessor. We denote a hash list for topic $t$ as $H_t$ and an element in such list at position $i$ with $H_t^i$. The length of a list $n_t = |H_t|$ can be chosen freely, depending on the available memory space. The hash list is constructed as follows: $H_t = \{F(s), F(H_t^1), F(H_t^2), ..., F(H_t^{n_t-2}), F(H_t^{n_t-1})\}$. Secondly, the sender chooses a random string $p$, which will be used as a prefix for the topic in each sent message. Using predefined topics, the sender publishes a message $m$ to all eligible clients containing the prefix $p$ and a mapping $M : t \rightarrow H_t^{n_t} \forall t \in T$, that is $m = \{p, M\}$. The payload of message $m$ is encrypted using the pre-shared key.

The receiving clients subscribe to each topic beginning with the prefix $p$ using the wildcard #, i.e. "p/#". The sender publishes messages for topic $t$ constructing an obfuscated topic as $p/H_t^{n_t-1}$. Each time a message is published the last element in $H_t$ is deleted and $n_t$ is decremented by one. On the receiving side, the client removes the prefix (including the delimiter "/") to get $\tau$. Then, he uses the hash function $F$ to compute the hash value $h = F(\tau)$. Using the mappings $M$ received before in message $m$, the client can then determine which topic maps $h$. If a mapping is found, it is replaced by $t \rightarrow \tau$. If no mapping is found, the message is ignored.

Using predefined topics, a request mechanism for message $m$ by the receiving clients can be implemented. This way, eligible clients can still participate, even if they join in later. When the length of a hash list is decreased to 1, the sender needs to construct a new list and notify the receiving clients, by sending an updated mapping.

In case the sender has very limited memory space, storing the random string $s$, prefix $p$, and number $n_t$ is sufficient to generate the obfuscated topic $p/H_t^{n_t-1}$ in order to publish a message. This requires more energy and may not be suitable for battery-powered devices, as with each publication $n_t$ hash computations need to be performed. It also may not suitable for some real-time applications.

The length of the prefix should be quite small. In case two independent parties chose the same prefix, the broker will forward "wrong" messages, but since the receiving client can not de-obfuscate them, these messages will be ignored. At the same time, these "collisions" make it more complicated for the broker to identify communication partners. To cause such collisions, instead of a random prefix, the first level of an often used topic could be selected. The selection should consider how frequent a topic is used. For example, we observed about 1.4 million messages in 48 hours, published on a public broker

who's topic started with "iot/". The attempt to de-obfuscate a message topic requires time and energy, so the prefix iot/ is obviously no good choice.

Since each message is published under a topic based on the random string $s$, the real topic is never revealed to the broker. However, when using a standardised hash function a data analyst might recognise the scheme and is able to link corresponding messages together. Therefore, fully randomized obfuscation is needed.

### E. Advanced One-Time Password (AOTP)

This section describes an extension to the OTP approach, which was introduced in section III-D. The approach is also using a hash list for each topic $t$, which is being published. As before $H_t$ denotes such a list for topic $t$, $H_t^i$ denotes a hash value at position $i$ within that list, and $n_t$ is the length of the list $H_t$. In contrast to the approach before, we define the hash function as $F : * \rightarrow \mathbb{Z}$. The function $str$ returns the string representation of a hash value (e.g. using the hexadecimal format), whereas the function $num$ returns the hash value of such string representation. Furthermore, a random polynomial $P$ is chosen as well as a random string $s$ and the prefix $p$. The hash list is now constructed as follows: the first element is again the hash value of $s$, i.e., $F(s)$. Each following element is calculated as $F(H_t^{i-1} \cdot P(i)) | 2 < i \leq n_t$. Now, the sender sends an encrypted message $m$ to all eligible clients as $m = \{p, M : t \rightarrow H_t^{n_t} \forall t \in T, P, n_t \forall t \in T\}$ where $T$ is the set of topics the sender intends to publish messages on. A message is published using the topic $p/str(H_t^{n_t-1})$. After publishing a message the number $n_t$ is decremented by one and $H_t^{n_t}$ is deleted.

Upon reception of a new message, a receiver removes the prefix $p$ and applies function $num$ to get $\tau$ and calculates $h = F(\tau) \cdot P(n_t)$ for each topic mapping stored in $M$, until a matching mapping is found. The mapping is then replaced with $t \rightarrow \tau$ and $n_t$ is decremented by one.

The use of a random polynomial $P$ makes it harder for a data analyst to link two hash values, as it acts as a sort of salting mechanism. This approach could be simplified, by using a constant random number multiplied with each preceding hash list element. On the other hand, a different polynomial $P$ for each message type $t$ is also possible, in order to make it even more difficult to analyse the connection between obfuscated message topics.

The prefix $p$ makes it possible to limit the number of messages that are received, to limit unnecessary de-obfuscation attempts. As explained in section III-D, sharing the same prefix with different parties prevents the broker from identifying communication partners. As before, a reasonable tradeoff must be found, to reduce the de-obfuscation attempts.

## IV. Evaluation

For the purpose of evaluation, we implemented the AOTP method (see subsection III-E). The implementation was done in the programming language Python using the MQTT library Paho [1] by the Eclipse Foundation. The implementation consists of two parts: the first part generates the hash lists and publishes the message $m$. Afterwards, new messages are published under a randomly chosen message topic out of a pool of predefined topics. The second part of the implementation receives these messages and attempts to de-obfuscate the topics. As broker a public instance of the Eclipse Mosquitto at *iot.eclipse.org* was chosen, supporting protocol version 3.1. *Note: the authors do not believe, that the Eclipse Foundation as operator of this public broker is collecting data to create and store user profiles.* As hardware platform, a Raspberry Pi 2 Model B was used.

In our evaluation, we investigated how much of a delay is introduced using the AOTP obfuscation strategy. We define delay as the time between a message is published by one client and usable (i.e. the topic is de-obfuscated) at a receiving client. In a first step, we measured the network latency caused when using the public broker and compared it with the latency of a local instance of the same broker software (Mosquitto). Afterwards, the delay while obfuscating the message topics was measured. That means, in total 4 different combinations were measured: AOTP with a remote and a local broker and un-obfuscated (plain) message topics with a remote and a local broker instance. To avoid measurement errors due to different workloads on the broker's side, each combination was repeated in a 10 minute interval over the course of 4 days. In each iteration, the sending part published 10 messages, each with a random topic out of a pool of 39 topics. In total, each combination was executed more than 3800 times. The payload of each message was the timestamp of publication. The receiving part was executed on the same Raspberry Pi, so no synchronisation of different clocks was required. Upon receipt, the receiver calculated the delay as the difference between the timestamp in the received message and a current timestamp.

In figure 1 we compare the total delay (communication time + de-obfuscation time) when using the public broker provided by the Eclipse Foundation and a locally running broker instance. As expected, the local instance is much fast than the remote broker. 75% of the message topics are usable in a tenth of the time or less. This shows that the network overhead is more significant than the overhead due to the de-obfuscation when using a Raspberry Pi.

Figure 2 compares the delay between obfuscated and un-obfuscated (plain) message topics when using the local broker instance. Without using the AOTP obfuscation the delay remains below 5 milliseconds. The figure shows, that a delay of 3.7 milliseconds is introduced by the broker and due to the communication using the loopback interface. When using the AOTP obfuscation, the smallest delays start at about 5 milliseconds. Subtracting the average delay introduced by the broker and due to the communication using the loopback interface, the fastest de-obfuscation requires about 1 millisecond. The delay increase gradually. In 50% the delay is between 10 and 15 milliseconds. Compared with figure 1, the de-obfuscation requires about 10% of the total time to transfer a message using a remote MQTT broker. For

completeness, we also measured the de-obfuscation time using an ESP8266 microcontroller. Here, on average, the delay was 120 milliseconds. Roughly speaking, the delay is doubled on such a device utilising the AOTP strategy in combination with a remote broker. In practice, it will be more likely that such a microcontroller is used to collect data sets and publish them through the broker. Hence, the de-obfuscation time is less relevant on such devices. In general, it can be expected that the de-obfuscation will take place on an unconstrained device in most cases.

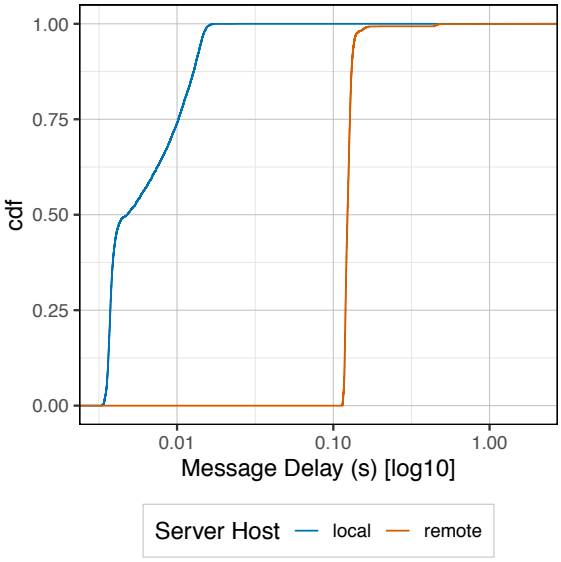


Fig. 1: Comparison of localhost and remote message delays

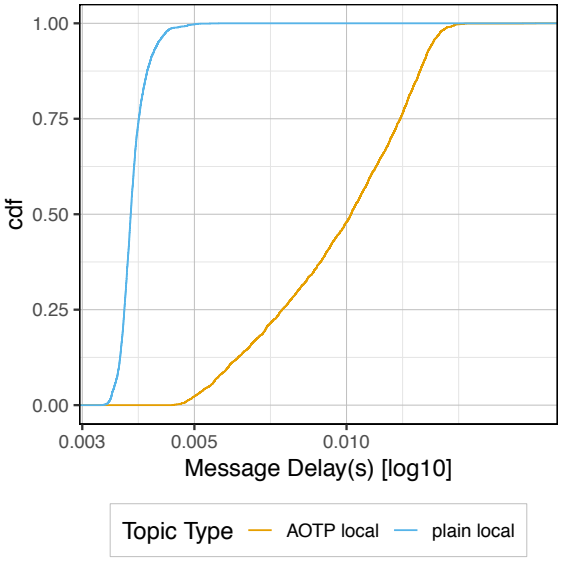


Fig. 2: Comparison between AOTP and un-obfuscated topic

In figure 3 the pure de-obfuscation time of the 39 different topics is depicted. As indicated by figure 2, the de-obfuscation time varies roughly between 1 and 13 milliseconds (no broker delay). Furthermore, the figure shows that the de-obfuscation time depends on the topic itself. The reason for this is the ”try-and-error” approach when de-obfuscating a topic. For a better understanding, figure 4 depicts the flow chart of the de-obfuscation algorithm. The algorithm iterates through the possible topics $M_i$ within the mapping $M$. In each iteration, the algorithm needs to compute the hash $h$ and compares it with the stored value in $M$. This means, in the worst case the receiver needs to calculate 39 hashes for all 39 topics. In the best case, only one computation is necessary. In general, if only a small number of topics are exchanged this effect is minor. With an increasing number of possible message topics, the maximal delay also increases.

In a real-world scenario, where one topic is more likely to be published than another, the algorithm can be optimised. Depending on the frequency the message topics are used, the mapping can be reordered so that the most frequent topics will be tested first. In case the sender knows the frequency beforehand, an ordered list of message topics could be published inside message $m$. Otherwise, the receiver could create such an ordered list on-the-fly by observing the message topic frequencies.

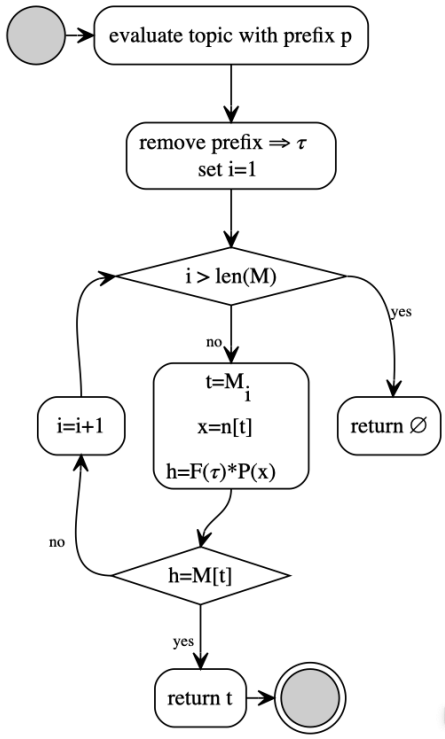


Fig. 4: Flow chart of de-obfuscation algorithm of the AOTP strategy

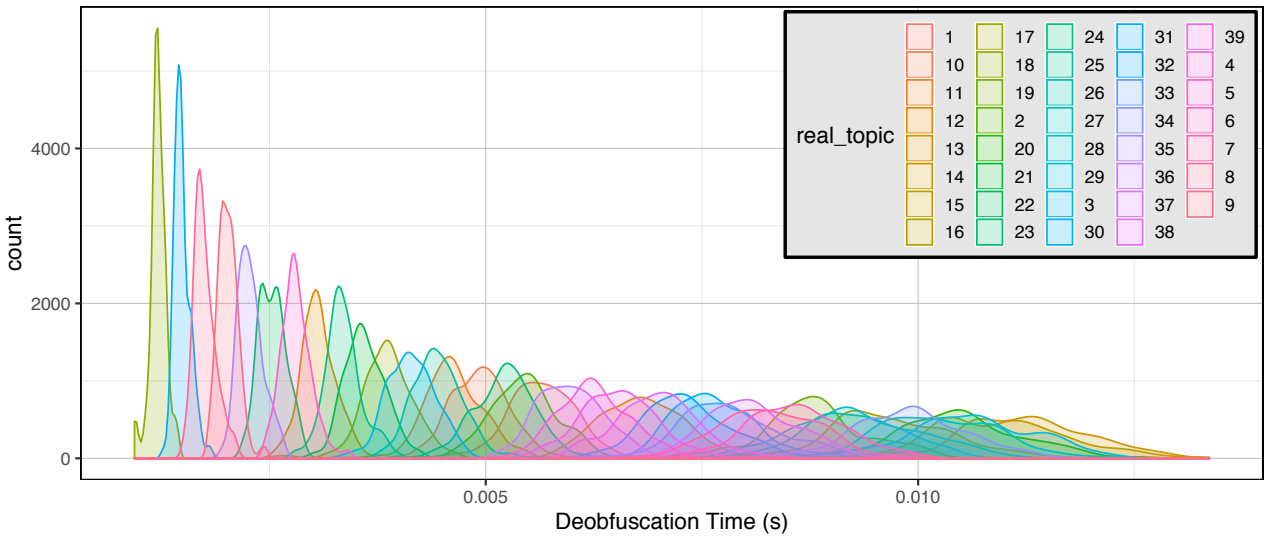


Fig. 3: Density of different topics

## V. Conclusion

Using public MQTT brokers has the advantage that no modifications in a NATed network nor a separate server are needed. To be able to forward a message the message's topic needs to be interpretable by the broker. In this paper we presented different strategies to obfuscate the message topics when publishing messages. The obfuscation protects the privacy when using public brokers. A strategy using iterative hash lists for each topic published named Advanced One-Time Password was implemented and evaluated. The evaluation showed, that the time overhead introduced to de-obfuscate a message topic at a receiving client is only a fraction of the delay caused by the network connection. Even on a resource-constrained device such as an ESP8266 the de-obfuscation required as much time as the message transport did, making the approach suitable for most real-world scenarios.

The de-obfuscation time varies at a high degree, depending where a topic is stored inside the mapping $M$. Analysing the frequency of the different message topics allows to optimise the order in which de-obfuscation attempts are executed.

The source code of the evaluation is published at Github.com [5].

## Acknowledgement

This work is part of the research project AgraSEC founded by the "Europäischen Fonds für regionale Entwicklung (EFRE)" (project number 85003218) and IoTCrawler founded by the European Union program H2020 (grant number 779852).

## References

[1] Paho MQTT library: https://www.eclipse.org/paho/.

[2] Public MQTT Broker https://github.com/mqtt/mqtt.github.io, January 2019.

[3] Andrew Banks and Rahul Gupta. Mqtt version 3.1. 1. *OASIS standard*, 29, 2014.

[4] Marco Calabretta, Riccardo Pecori, and Luca Velti. A token-based protocol for securing mqtt communications. pages 1–6, 09 2018.

[5] Marten Fischer. Improving the Privacy in the MQTT Protocol. github repository, https://github.com/mafi-mcfly/aotp-mqtt, February 2019. accessed: 2019-02-19.

[6] S. Kraijak and P. Tuwanut. A survey on internet of things architecture, protocols, possible applications, security, privacy, real-world implementation and future trends. In *2015 IEEE 16th International Conference on Communication Technology (ICCT)*, pages 26–31, Oct 2015.

[7] R. Neisse, G. Steri, and G. Baldini. Enforcement of security policy rules for the internet of things. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 165–172, Oct 2014.

[8] S. Shin, K. Kobara, Chia-Chuan Chuang, and Weicheng Huang. A security framework for mqtt. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 432–436, Oct 2016.

[9] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar. Secure mqtt for internet of things (iot). In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 746–751, April 2015.

[10] Y. Upadhyay, A. Borole, and D. Dileepan. Mqtt based secured home automation system. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4, March 2016.