Burglars' IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds

Yan Jia[1,2,3†], Luyi Xing[3], Yuhang Mao[1,2], Dongfang Zhao[3],
XiaoFeng Wang[3], Shangru Zhao[1,2], and Yuqing Zhang[2,1*]

[1]*School of Cyber Engineering, Xidian University, China*
[2]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences ,China*
[3]*Indiana University Bloomington, USA*

***Abstract*—With the increasing popularity of the Internet of Things (IoT), many IoT cloud platforms have emerged to help the IoT manufacturers connect their devices to their users. Serving the device-user communication is *general messaging protocol* deployed on the platforms. Less clear, however, is whether such protocols, which are not designed to work in the adversarial environment of IoT, introduce new risks. In this paper, we report the first systematic study on the protection of major IoT clouds (e.g., AWS, Microsoft, IBM) put in place for the arguably most popular messaging protocol – MQTT. We found that these platforms' security additions to the protocol are all vulnerable, allowing the adversary to gain control of the device, launch a large-scale denial-of-service attack, steal the victim's secrets data and fake the victim's device status for deception. We successfully performed end-to-end attacks on these popular IoT clouds and further conducted a measurement study, which demonstrates that the security impacts of our attacks are real, severe and broad. We reported our findings to related parties, which all acknowledged the importance. We further propose new design principles and an enhanced access model MOUCON. We implemented our protection on a popular open-source MQTT server. Our evaluation shows its high effectiveness and negligible performance overhead.**

I. Introduction

The popularity of Internet of Things (IoT) devices and the demands for their convenient deployment and control give rise to a new type of services, dubbed IoT cloud platforms. Such platforms are widely deployed by IoT device manufacturers to enable users to remotely control various devices such as smart locks, switches, thermostats, etc., from everywhere at anytime (Fig. 1). Today, many leading cloud service providers and IoT device manufacturers provide such cloud services. Examples include AWS IoT Core [1], Microsoft's Azure IoT Hub [2], and Samsung's SmartThings [3]. At the center of these services is the mechanism that mediates the communication (e.g., control commands and messages) between IoT devices and users. Such communication is built on existing general messaging protocols, in particular *MQTT* (Message Queuing and Telemetry Transport). MQTT is an OASIS standard [4] and designed to support the communication with weak computing devices over unreliable and low-bandwidth channels, and thus is well suited for IoT-user interactions. Hence, it has been widely adopted by mainstream IoT clouds [5], including those offered by AWS, Microsoft, IBM, Alibaba, Tuya, Google, etc.

Security risks in MQTT. Unfortunately, MQTT is *not* designed to operate in an adversarial environment, and therefore cannot protect itself against potential threats to the IoT systems it serves. More specifically, the protocol has almost no built-in mechanisms for authentication and authorization, forcing the cloud platform providers to develop their own safeguards. Given the complexity in customizing the general-purpose protocol to work in diverse IoT application scenarios, effective protection of its communication is challenging. In the absence of a proper security analysis, there is little confidence that the user-device interactions have been effectively secured. Given the critical role the protocol plays, any security weakness, once exploited, could have serious consequences, such as loss of device control, disclosure of sensitive user data, etc. However, with the importance of the problem, little has been done so far to understand whether MQTT and the communication mechanism it supports have been adequately protected.

Attacks. In our research, we performed the first systematic study on the security threats to the integration of a general messaging protocol on major IoT clouds. As a first step, we focused on MQTT, arguably the most widely-adopted protocol across IoT cloud platforms [5]. Our research shows that the cloud's security additions to MQTT are often vulnerable (Section III). Exploiting such vulnerabilities, the adversary can (1) control the victim's devices (e.g., unlocking the door and window), (2) infer sensitive user information, such as her daily routine, health condition, location, cohabitant, (3) perform a denial-of-service (DoS) attack against a vast number of devices, and (4) impersonate a target device to the cloud to manipulate the messages delivered to the user. The consequences of such attacks are significant, sometimes even devastating. For example, we found that the weak authorization on Suning Smart Living enables a remote adversary to collect all events produced by all the devices managed by the cloud,

†The majority of work was done when the first author was visiting Indiana University Bloomington.
*Corresponding author

DOI 10.1109/SP40000.2020.00051

including identifiable user information (such as emails, phone numbers) and device information.

Looking into the root causes of these flaws, we found that most IoT clouds today fail to appreciate the security implications in running a relatively simple general messaging protocol in complicated IoT environments, leaving the door wide open to new risks when the gap between the protocol and real-world device-use scenarios has not been fully covered with proper protection. Particularly, we found that the revocation process, which has not been considered in MQTT, presents a serious security challenge to its application to IoT communication. IoT devices (like a smart lock) today are increasingly shared among users, e.g., hotel dwellers, Airbnb apartment renters, home visitors (such as babysitters, etc.), according to recent studies [6]–[10]. Once a person's access right to a device has been terminated, he should not be allowed to connect to it again without proper authorization. This protection, however, is found to be hard to enforce on mainstream platforms, due to the utility-oriented design of the protocol that gives the user various ways to reach devices: our research shows that through a topic, a session and other entities, a malicious ex-user can retain full control of the devices on which his access privilege has expired, even receiving the current user's private messages like health conditions, cohabitation relations, etc. (Section III-A, III-B and III-D). Also discovered is the observation that security-sensitive states and state transitions in the original protocol have not been fully identified and secured for running the protocol in an adversarial environment. For example, MQTT includes a *ClientId* field as a unique identifier for each client, and the cloud disconnects an existing client when a new client with the same ClientId shows up. This entity is not protected in the original protocol, due to its design for operating in a benign environment. However, when the protocol is run on an IoT cloud, in the absence of authentication involving the entity, we found that the adversary can abuse the protocol state to disconnect clients through claiming their ClientIds in a large scale, which leads to a DoS attack or even MQTT session hijacks (Section III-C).

Impacts. To understand the impacts of these security flaws, we conducted a measurement study that analyzed eight leading IoT cloud platforms (including those provided by AWS, IBM, Microsoft, etc.). Our research has revealed the pervasiveness and significance of the security risks: Most of these platforms contain such critical flaws in their customized MQTT protocols, exposing thousands of device manufacturers and millions of users to the aforementioned hazards (Section IV). We reported all discovered problems to the affected parties, including AWS, Microsoft, IBM, Alibaba, Baidu, Eclipse Mosquitto, etc., and are working with them to address these issues. Most importantly, our findings have been brought to the attention of the MQTT Technical Committee (TC), which is seeking solutions through open discussion now [11]. The video demos of our attacks are posted online [12].

Given the pervasiveness and significance of the problems discovered, we envision that serious efforts need to be made to standardize authentication and authorization protection of general messaging protocols for IoT clouds. As a first step, we propose a set of secure design principles and implemented them on Mosquitto [13], a popular open-source MQTT server. Our evaluation shows that this protection incurs only a negligible overhead and effectively addressed all the problems we discovered. We are communicating with the MQTT TC about the solution and helping them enhance protection of the protocol. Also we release the code of our technique online [14].

Contributions. The contributions of the paper are outlined as follows:

- *New understanding of secure IoT communication*. We performed the first systematic study on security risks in use of the general messaging protocol for IoT device-user communication. Our research reveals the gap between the protocol designed for operating in a simple and benign environment and the complicated, adversarial IoT use scenarios, and the challenges in covering the gap with proper security means (through both extending the protocol to cover new scenarios such as revocation and protecting existing security-sensitive states and transitions). The lesson learned from the study will contribute to better design and implementation of other protocols to work in adversarial environments.
- *Measurement*. We demonstrate the pervasiveness of the security risks across leading IoT cloud platforms and a popular open-source MQTT server, and identify the serious consequences of the attacks on the vulnerabilities discovered.
- *Secure design principles and implementation*. We proposed new security principles in customizing general messaging protocols for IoT clouds and implemented end-to-end protection. We show that our approach is both effective and efficient, and can be conveniently integrated into today's IoT platforms. Through our communication with the MQTT TC and IoT cloud platforms, this first step could lead to better protection of user-device interactions in the real world.

II. Background

A. Cloud-based IoT Communication

Architecture. A cloud-based IoT system typically includes three components: the *cloud* (also referred to as *cloud platform* in this paper), the *IoT device*, and the user's *management console* (mobile apps in particular) to control the device, as illustrated in Fig. 1. Central to the system is the cloud that manages the communication between the device and the app, through which the app sends control messages (commands) to the device (e.g., to lock a smart door) and gets information back from the device (e.g., temperature from a thermostat, the "on" or "off" status of a lock). To protect such interactions, the cloud authenticates the device and the app (representing a user), and decides whether the user should be allowed to command the device or receive messages from it. To this end, the cloud provider offers SDKs that implement certain messaging protocols (e.g., MQTT), which are integrated by the IoT manufacturers into their devices and mobile apps for

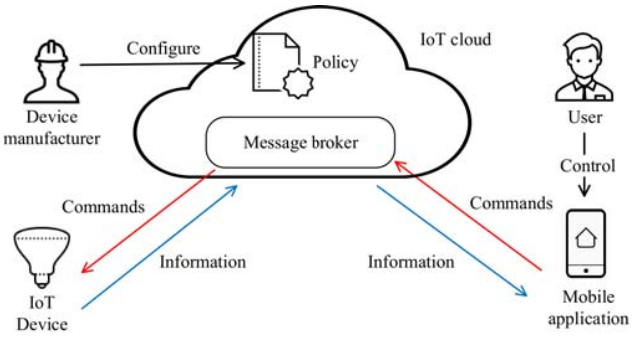

Fig. 1. Architecture of cloud-based IoT communication

communication through the cloud. This paradigm has been adopted by most IoT cloud providers (e.g., AWS, Microsoft, IBM, Tuya, Alibaba) and device manufacturers [15], [16].

MQTT and its IoT application. A publish-subscribe messaging protocol allows the sender to deliver messages to a class [17], based upon the topic of the messages or its content, subscribed by a group of receivers. A prominent example is MQTT[1], which is an application-layer protocol (based on the OSI model [19]), and runs over TCP/IP or other ordered, lossless, bi-directional connections like WebSocket [20]. MQTT is known for its light-weight design, which works on resource-constrained devices in low-bandwidth or unreliable networks, and thus is well suited for supporting the IoT ecosystem.

At the center of MQTT-based IoT communication is *MQTT message broker* (or *broker* for short in this paper), as shown in Fig. 1. The *broker* hosts MQTT *topic*s at its server endpoint with each topic structured like a hierarchical file path, such as `/doorlock/[UUID-8JH...S9P]/status`. With the broker as the connection pivot, MQTT leverages a publish–subscribe pattern [17] for communication: the MQTT client (e.g., an IoT device or a management app) publishes a message to a specific *topic* hosted by the broker, then the *broker* routes the message to the other clients that have subscribed to the topic. A client can subscribe to a specific level in a topic's hierarchy or use a wildcard (#) to connect to multiple levels.

In the MQTT communication, the client sends three basic types of messages to the broker, `CONNECT`, `PUBLISH` and `SUBSCRIBE`, as illustrated in Fig. 2. First, the MQTT client, e.g., a smart air conditioner or an app, sends a `CONNECT` message to the broker for establishing an MQTT session (if the broker accepts the connection). The session and the client are uniquely identified by a *ClientId* field (embedded in CONNECT message), which is similar to a web session cookie. In the established session, an IoT device subscribes to its *associated topic* (e.g, `/DeviceId/cmd`) by sending

[1]In this paper, we mainly focus on MQTT version 3.1.1, which is widely adopted by all leading platforms we studied (e.g., AWS, Microsoft, IBM, Alibaba, etc., see measurement Section IV). Although the latest version 5.0 [18] has already been released in 2019, it has seen a limited deployment so far, as observed in our measurement study. In the meantime, most of our problems found in our research are also present in the new version.

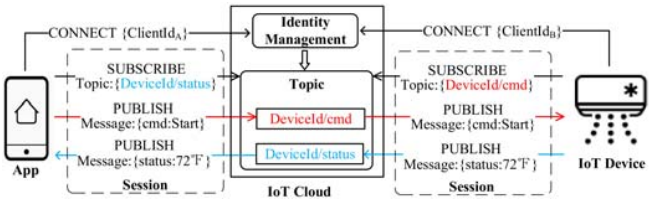

Fig. 2. Use MQTT protocol for IoT communication

a *SUBSCRIBE* message (including its *topic*) to the broker. The broker maintains the subscription status for each session and delivers the MQTT message published to a topic to its subscribers. Through this channel, an app can work on its user's behalf to operate on a device, by publishing commends to the topic the device subscribes (e.g., start or stop). Also, the device can periodically update its state information to a topic, such as the current temperature, which is received by all the apps subscribing to the topic.

In this process, the whole MQTT communication relies on four entities: identity (*ClientId*), message, topic and session. Hence, whether these entities have been well protected is critical to the protocol's secure application to the IoT environment.

B. Protection of MQTT on IoT Clouds

As a general messaging protocol, MQTT is not designed to work in an adversarial environment: for example, it lacks build-in authentication and authorization. To protect the communication involving sensitive IoT devices (e.g., door locks, cardiac devices, security cameras, fire detectors), particularly private information such devices collect, an IoT cloud often has its custom security mechanisms in place to authenticate MQTT clients and authorize the parties eligible for publishing or subscribing to an MQTT topic. Here, we summarize such protection deployed on leading IoT clouds based on our study of eight leading IoT clouds.

Client authentication. MQTT connections go through WebSocket and TLS [21], which are authenticated using a variety of mechanisms deployed on different IoT clouds. For example, AWS IoT supports Amazon username/password, single sign-on (SSO) [22] through Google/Facebook, and Amazon Cognito [23] (a login scheme across various AWS cloud services). As another example, in a TLS connection, the client presents a cryptographic certificate to authenticate to the cloud with the TLS client authentication mode [24] – a typical approach to authenticate devices, which may carry built-in certificates.

Client authorization. The IoT cloud platform aims to ensure that each user can only send commands to and receive messages from the devices the person is allowed to use. For this purpose, the cloud enforces a set of security policies. Examples include the *topics* and *messages* a client is allowed to access and the actions (e.g., publish or subscribe) it can take.

C. Threat Model

We assume that the adversary can open user accounts with IoT device manufacturers and IoT clouds and is also capable of collecting and analyzing network traffic between the IoT cloud, the IoT device and the app under his control. On the

other hand, he cannot eavesdrop on the communication of other users' devices and apps.

In some of our attacks, we consider the device-sharing situation that becomes pervasive today. Hotels, Airbnb, apartments and other vacation rental homes are increasingly equipped with IoT devices and their guests are routinely granted temporary access to the devices [7]–[10]. A recent study [6] further shows that most people are willing to share their devices (e.g., smart lock), for example, with family visitors or babysitter, etc. Under these circumstances, an IoT access control model is expected to handle revocation in a secure and reliable way, which is completely outside the original MQTT protocol. Note that except the study on revocation, we do not assume that the attacker is granted temporary access to target device(s) in other attacks.

III. Security Analysis of MQTT IoT Customization

In this section, we report a security analysis on the MQTT-based communication mechanisms operated by leading IoT cloud platforms. Our study systematically inspected individual protocol entities, including identity, message, topic and session, in an attempt to understand whether their related security-sensitive protocol states have been properly guarded or whether they could be abused to circumvent the IoT protection MQTT does not support. The study shows that such security gaps do exist and oftentimes, these entities have not been properly authenticated or authorized to cover the gaps. To understand the security implications of the findings, we further implemented end-to-end attacks on all the problems discovered, with their demos posted online [12]. It is important to note that all such experiments were conducted in an ethical way: we always aimed at our own devices, never putting the cloud services, platforms and other users in danger. Also importantly, we reported all our findings to the manufacturers and IoT cloud providers, sharing our PoC attacks with them and helping them improve their security protection. This effort has been well received and acknowledged, and we received 6,700 USD in total from those vendors' bug bounty programs.

A. Unauthorized MQTT Messages

As mentioned earlier, MQTT was *not* designed to work in the diverse scenarios of IoT communication. A prominent example is device sharing and revocation, in which a party (hotel dweller, Airbnb tenant, babysitter, etc.) is only trusted with *temporary* access to an IoT device and not allowed to get information from or interfere with the activities of prior and future users of the same device. This potentially adversarial situation is completely outside the MQTT protocol. Without in-depth understanding of the problem, the current IoT cloud platform just cannot handle it in a secure way. Following we present the new security risks we discovered in the scenario due to the insecure management of MQTT messages, including *Will message* and *Retained message*, which leads to unauthorized control of IoT devices.

Unauthorized Will Message. Based on the MQTT specification, the client can register with the broker a special *Will Message* (in a `CONNECT` type MQTT message, see Section II) for a topic. Once the client is accidentally disconnected (i.e., not sending a `DISCONNECT` message to the broker), the broker will publish the *Will Message* to all subscribed clients of the topic, allowing them to take corresponding actions. Like other MQTT messages, *Will Message* can include either control commands or just text.

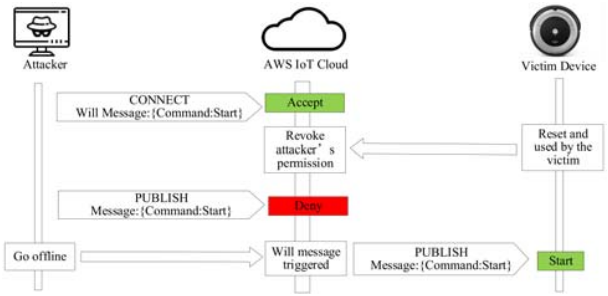

Fig. 3. Will Message Attack on iRobot Device

However, this exception handling feature was not meant to work in the adversarial environment, when the access right on a device is transferred from one party to another. In this case, a malicious ex-user can strategically register a *Will Message* to trigger it later when he no longer has the access privilege, to stealthily issue commands when the device is serving other users: for example, a babysitter or a repairman given access to a smart lock once could open the door for others later. Specifically, we found that even after the attacker's privilege on a device expires, so he is prevented from direct communicating with the device by the broker based upon the owner-specified policy, on the protocol level, the attacker's *Will Message* will still be issued *as soon as his client is disconnected from the broker*. In this way, he can decide when to unlock the door in the above example by choosing the right time to go offline. Note that, the attacker can launch the attack multiple times leveraging multiple clients to register multiple *Will Message*s (e.g., 10 times in our PoC attack below).

The problem can be addressed by removing the *Will Message* registration when one's access right is revoked. This, however, is complicated since the IoT cloud platform can no longer just operate on top of MQTT, by controlling whether a user should communicate with a device based upon her platform ID, but needs to get into the protocol to expand its state machine with the capability to handle revocation (e.g., finding out each state related to the party like registering a *Will Message* and cleaning up each of them). This has never been thought of, up to our knowledge, and does not exist on all leading IoT clouds we studied (see Section IV).

Attack through *Will Message*. We implemented a PoC attack exploiting *Will Message* on the AWS IoT cloud using our iRobot Roomba 690 (Fig. 3). Specifically, we wrote a script to register a *Will Message*, including a command (`Command:Start`) to start our device with the broker. Then, when a victim user (like the subsequent guest of the same hotel room) reset the device and used the robot and our client's privilege was completely revoked (e.g., it was denied of pub-

lishing messages to the device), we show that the commands in the *Will Message* were still delivered and invoked the robot when our script went offline. Such an attack can be amplified by exploiting the problem multiple times: we also tried 10 "malicious" clients to register 10 *Will Message*s at a time, and each client was able to independently launch an attack.

The same problem has been confirmed on the IoT clouds of AWS, IBM, Baidu, Tuya Smart, etc. Given the absence of proper security checks in the clouds and missing of security advisory for device manufactures, the problem potentially affects all devices on these clouds, such as door/window locks, video doorbells, cardiac devices, security cameras, fire detectors, etc., which are highly security-, privacy- and even safety-sensitive.

Discussion. Through our conversation with the MQTT Technical Committee and inspecting the MQTT specification, we found that the problem comes from the contract-like property of *Will Message*: a registered *Will Message* on a topic is entitled to deliver to all clients that subscribed to the topic; whether the message violates the security requirement of subscribing clients is not a concern of MQTT. Such a contract assumes a much more trusted environment MQTT is designed for, not the adversarial environment of the IoT communication (e.g., the previous device user may attack the subsequent one). The problem we found indicates that IoT vendors should understand *this gap* and extend the protocol to address its security implications. For the first time, our finding shows that the importance of covering this gap has been largely underestimated and the security risks are highly realistic. Also we found that the problem is not limited to *Will Message* but also other MQTT features, as elaborated later.

Unauthorized retained message. When an MQTT client publishes a message to a topic, and no client is subscribed to the topic, the message is simply discarded by the broker. However, this simple treatment also disrupts the publisher's communication with its subscribing clients when the clients are all temporarily offline. To address the problem, the MQTT client can register a *Retained Message* with a topic (by setting the `retained` flag in a regular MQTT message), which allows the broker to keep the last *Retained Message* on the topic, and publish it immediately to any future subscribers to the topic.

Just like *Will Message*, this feature was not designed to work in the adversarial IoT environment and again we found that it can be exploited by a malicious ex-user to stealthily command a device he no longer has access to. For example, the malicious ex-user of a device in an Airbnb room can publish a *Retained Message*, which includes arbitrary control commands (e.g., `open the door at 3 am`), to the associated topic of a smart lock when he still has an access right. Later when he checks out and therefore loses the privilege, he can wait for the device to get back online. When this happens, the lock will subscribe to the old topic again and receive the `unlock` command. Once executed, the door will be open at 3am and burglars can get in.

PoC Attack. We performed PoC attacks and confirmed that the IoT cloud of Baidu (one of the top Public Cloud Services Providers in China [25] and with the fourth largest website in the world [26]) and the Eclipse Mosquitto (a famous open-source MQTT broker [13], which was forked more than 950 times [27] and deployed in popular open-source IoT platforms [28], [29]) were subject to the *Retained Message* attack. On both platforms, we first ran a "malicious" client (representing the ex-user) to register a *Retained Message* onto a topic, and then revoked its regular permission. Later, when a "victim" client (representing the device when it is used by the different user) subscribed to the same topic, both platforms forwarded the attack messages to the victim, which can lead to unauthorized control on a device.

Responsible disclosure. We reported the problems to affected parties including AWS IoT, IBM, Baidu, Tuya Smart, Eclipse Mosquitto, etc., which all acknowledged their importance. AWS had an online meeting with us to discuss possible solutions to mitigate the risks to IoT users.

B. Faults in Managing MQTT Sessions

As introduced earlier, the MQTT communication is through the established session between the client and the broker server, and each session is associated with an MQTT client. Therefore, when a client has a state change (e.g., his/her access to a device is revoked), the states of its established session should be updated, which is particularly important to security sensitive ones such as subscription state (which topics are subscribed to) and the lifecycle state (whether the session should last or be terminated) in particular. However, this expected security property is often not in place on real-world IoT platforms, as elaborated below.

Non-updated session subscription state. MQTT specification suggests that the server authorizes particular *actions* of the client [18]. With this guidance in place, IoT platforms typically enforce a security policy to govern the client's operations. For example, when a device is reset to completely remove all privileges of an ex-user on the device, in any established session, the user's client becomes no longer permitted to take any proactive action, such as to `SUBSCRIBE` to the device's topic. However, when we inspect the session's state management, we found that the MQTT specification provides no guidance in updating session states in response to the client's privilege change. Likely due to such lack of guidance, session management in real-world IoT systems, particularly when it comes to a session's subscription state, was found to have privacy-critical defects. Specifically, as long as a client establishes a session that ever subscribed to a topic (e.g., the topic of a smart speaker in a hotel room), even when the user is no longer permitted to subscribe to the topic (e.g., after checking out), we found that the broker continuously delivers messages to the client through the established session. That is, the subscription state of the session lasts even after the subscriber lost his privilege, which effectively enables the malicious ex-user of a device to continuously receive all

messages generated by the device for the current user (victim), such as personal buying history and habits, health conditions and data (e.g., heartbeats), etc. Such an unsound session state management is confirmed on major IoT clouds (e.g., IBM, Tuya, Alibaba, Baidu, see responsible disclosure below).

Non-updated session lifecycle state. The MQTT client in the IoT environment can represent two different roles, the *device* (when the client is authenticated through a device's credential) and the *user* (when the client is authenticated through a user's credential). The two roles are managed differently by IoT clouds from the security perspective: the device is treated as the resource to access, and the user is regarded as the principle to authenticate and authorize. Such a difference is found to have security implications in the scenarios of IoT device sharing and revocation. Particularly, when a device is reset by a new user (for removing ex-users' access), permissions of the ex-user (and his/her client) for accessing the device are revoked (i.e., publish/receive messages through the device's topic). In contrast, there is no concept of revoking the permission of a device for accessing its topic. Hence, a possible attack is: the ex-user (attacker) obtains the device credential when he/she is permitted to use the device (obtaining device credentials is oftentimes trivial, such as through traffic analysis or reverse engineering, as demonstrated by a recent study [30] [31]), then even after the new user removes ex-users, the attacker can always leverage the device credential to impersonate the device and publish fake messages to the device's topic. Such an attack, however, is found to be mitigated by leading IoT platforms. For example, Tuya's IoT cloud ensures (leveraging its device SDK provided to manufacturers) that device credentials under its cloud are forced to expire if the owner changes (e.g., when the new user resets the device). Hence, with an expired device credential, the ex-user can no longer impersonate the device.

However, in our study, when we look at the problem from the perspective of MQTT session management, we found a new weakness on leading IoT platforms that allowed the attack to proceed. Particularly, through the obtained device credential, as long as the ex-user (attacker) establishes a session before the credential expires and keeps the session online, he/she can *always* publish fake messages to the device's topic through the session, on behalf of the device. Note that, the attack can continue even after the credential expires and thus can no longer be used to authenticate new clients or establish new sessions. This has serious security and even safety implications: for example, a burglar or criminal can leverage the fake message to show to the user that the door is locked or the gas valve is fastened though it is not; the fake message can trigger other sensitive devices (e.g., unlock a door) of the victim through a trigger-action IoT platform, such as IFTTT [32].

In this attack, the fault unwittingly made by the platform is: when the device is reset for cleaning all its existing users and more fundamentally, existing states, the lifecycle state of the established MQTT session is not updated. To be secure, the session's state should be cleaned up, e.g., by completely terminating the established session. However, MQTT specification [4] does not consider the security necessity of session state update in cases of user privilege change, likely due to its assumed much simpler and less adversarial usage environment, compared to IoT. The problem was confirmed and acknowledged by IBM, Tuya, Alibaba, Baidu, etc. (see Section IV), and an end-to-end attack was implemented below to show the feasibility of a real attack.

More fundamentally, the two attack scenarios above indicate that secure MQTT session state management, i.e., sound mechanisms that govern session state updates in response to IoT users' privilege changes, is understudied and suffers from new security risks in IoT environment. Such risks are highly underestimated by established MQTT development guide and real-world IoT vendors.

Attack. Exploiting the above weaknesses, we implemented PoC attacks using our MiKO smart socket which is mediated on the cloud of Tuya Smart [33] (hosting over 100 million smart devices). First, to exploit the session subscription state, acting as the malicious user, we established a session, which was able to continuously receive messages from the device even after a second user reset the device for removing all previous users. Second, to exploit the insecure session lifecycle management, through reverse engineering the device traffic, we were able to get the device credentials and established a "malicious" session. Later, the device was reset by a second user (victim) for removing ex-users' access, but the malicious session was able to continuously publish fake device messages (e.g., device status of "on") to the victim's app that subscribed to the device's topic. Through such an attack, a real burglar or criminal may break into a home, in the mean time publish fake device updates (impersonating home-safety devices) to the victim/police and show that the home is still safe.

Responsible disclosure. We reported our findings to affected vendors including Tuya, Alibaba, Baidu, IBM, etc. who all acknowledged the problem.

C. Unauthenticated MQTT Identities

As mentioned earlier (Section II), IoT cloud platforms authenticate their MQTT clients using their own platform-layer identities (e.g., Amazon accounts on AWS IoT cloud). In the meantime, each client is also identified by its own protocol-level identity *ClientId*. The relations between these two identities can be complicated: one account can have multiple devices, each with their own *ClientId*, while one device could be shared between multiple accounts. Such relations, if not managed well, could expose MQTT communication to attacks, as discovered in our research.

ClientId hijacking. The MQTT protocol requires the broker to disconnect the online client on observing a new client with the same *ClientId*. In an adversarial environment, one would expect that related MQTT protocol states and transitions (e.g., moving a client to the offline state under a detected conflicting *ClientId*) are protected by proper authentication involving *ClientId* and the token is kept secret. However, our

research shows that such protection is actually not in place over major IoT cloud platforms. As a result, the attacker can leverage his/her own authenticated *platform* identity to connect to the IoT cloud with an arbitrary *ClientId*, including the one belonging to a target device, so as to force the cloud to drop off the target. In our PoC attack, we show that the threat is realistic, and can be done in large scale, across most IoT clouds we studied (Section IV), such as AWS, IBM, Baidu.

The attack on *ClientId* actually goes beyond DoS. Specifically, the MQTT protocol allows the broker and the client to restore the prior session if the client connects with its previous *ClientId* (with `Clean Session` flag checked in CONNECT message [4]), which allows the client to quickly recover prior states (e.g., its subscriptions and the pending messages it is supposed to receive), so as to avoid configuration hassles. This recovery mechanism, however, can be abused to work against a target client, once its *ClientId* is disclosed to the attacker. Note that the attacker here can carry a completely legitimate *platform identity*. In the absence of the security policies that connect these two identities and regulate one's access rights to the related objects, a malicious cloud user can leverage the target's *ClientId* to resume its session and steal its messages (e.g., health condition, location, personal habits, etc.). In our research, we successfully executed PoC attacks on IBM Watson IoT and Baidu Cloud IoT, in which the attacker could receive the victim's messages though he never subscribed to the victim's topic. We found this is a realistic problem due to the improper identity management on real-world IoT clouds, which do not authenticate *ClientId* as long as the MQTT client proves its platform identity.

ClientId identification. The aforementioned attack is predicated upon the attacker's knowledge about the target's *ClientId*, which turns out to be completely realistic. Specifically, our research shows that apparently, both device manufacturers and IoT clouds are still unaware of the importance of keeping *ClientId* safe, allowing us to acquire a victim's *ClientId* in at least two ways, as elaborated below.

• *Making a guess*. A *ClientId* is a sequence whose format and semantics are determined by device manufacturers. It is used by a client to establish a connection with the *broker*, which either does not check its legitimacy at all or just inspects its basic format (e.g., only letters and numbers are allowed). When two clients connect with the same *ClientId*, the first connection will be disrupted from the *broker* as required by MQTT. Exploiting this property, we developed a simple attack to *test* the existence of a *ClientId*, by attempting to connect to the *broker* using the token: if it is already used by a client (connected app or device), the connections will fail repeatedly, since the client is competing with the attacker. Otherwise, the connection always goes through.

Key to the attack is efficient search for the *ClientId*s already assigned. We found that this is feasible since *ClientId* has *never* meant to be a secret. Actually, the MQTT specification only requires that the *ClientId* is unique [4]. Thus, popular IoT cloud platforms publicly propose to the customers (device manufacturers) the *ClientId* constructions should be easy to manage. For example, IBM (the creator of MQTT) advocates the use of a device's 48-bit MAC address as its *ClientId* to ensure its uniqueness [34]. Even the device manufacturers are unaware of the security implications of *ClientId*: they tend to choose such insecure formats as serial number [35], [36], which is often assigned incrementally [37]. As an example, the format of an Apple product's serial number is largely known, with its main part specifying the attributes like manufacturing factory, date, product model, etc., and only three digits for differentiating the devices with the same attributes [38].

In our research, we found that the format information can help us easily discover assigned sequences. For example, for MAC address, its first 24 bits are related to a specific manufacturer, while the remaining 24 bits can be enumerated to identify the sequences already in use [30]. Also when the sequence involving a serial number, we can start from a known *ClientId* to search its neighborhood, which likely hits other tokens using by the devices connected to the IoT platform.

• *Attacking a shared device*. Once a device has been accessed by a user (e.g., through hotel stay, apartment rent, etc.), its *ClientId* is forever exposed to the person (through looking at its MAC, serial number and sniffing network traffic, etc.). If the user is malicious, he can force the device to go offline from time to time (e.g., a revenge from a disgruntled Airbnb tenant) or even restore the current user's MQTT session to steal her sensitive messages (e.g., health condition and statistics, private habits, etc.). Note that this can be done *remotely* through the target device's cloud platform.

Attack. We implemented PoC attacks on the AWS IoT cloud, using our iRobot Roomba 690, a vacuuming robot, to show that a large-scale DoS attack is realistic. Our attack identifies a set of likely real-world *ClientId*s for iRobot devices, which contain serial numbers, through searching for the sequences in the neighborhood of our device's token. Our research shows that the sequences are accurate enough for a large-scale attack.

Specifically, from our own device, we found that iRobot actually used its 16-digit serial number (e.g., `3147C60043211234`) as its *ClientId*, which allowed us to continuously increase or decrease our device's number to generate potential *ClientId*s. Evaluating these tokens on the AWS cloud could be disruptive to legitimate users. So we resorted to a Web service discovered from our analysis of the traffic produced by the mobile app. The service is run on https://disc-prod.iot.irobotapi.com/robot/discover, which helps an app find out where its device's *broker* locates across different AWS servers around the world, given a *ClientId*. By querying the service with the 200K IDs, we were able to confirm in 6 hours that over 10K of them are the *ClientId*s for the devices already deployed. Note that our experiment utilized only a single thread to the queries and more threads and a longer time can potentially help us find hundreds of thousands of real *ClientId*s in a day.

To verify that indeed the attack can happen on a real-world cloud platform, we built a simple script to connect to

iRobot's broker on AWS IoT (https://a2uowfjvhio0fa.iot.us-east-1.amazonaws.com), which authenticated to the *broker* using our platform credential (the legitimate iRobot account we registered), but made the connection with the *ClientId* of a target device (another iRobot device of us). The attack immediately caused the target to go offline. Further we tested the scalability of the attack, successfully running our own client (through a single platform identity) to make 2000 concurrent connections with our own AWS IoT broker. This indicates that with a little more resources (scripts and platform identities), a DoS attack affecting tens of thousands or more devices is completely feasible.

The DoS weakness we discovered turns out to be general, which has been confirmed on other leading IoT clouds including IBM, Tuya Smart, etc. In addition to iRobot, we launched PoC attacks against our devices of Tuya Smart and Suning Smart Living. Later in Section III-D, we further show that leveraging another flaw, the attacker can easily obtain the *ClientId*s of devices from more than a hundred manufacturers [39] on the Suning IoT cloud.

Discussion. We believe that fundamentally, the security weakness comes from the insecure practice for managing *ClientId*s. The MQTT specification [4] does not treat *ClientId* as a secret in the first place and even suggests it could be as short as a single byte. More seriously, this problem has never been identified and addressed when applying the protocol to the potentially adversarial IoT communication. As evidence, even though MQTT specifies *ClientId* as an identity that represents a client, no IoT cloud has seriously considered the token as an identity like user ID, which is protected using various authentication schemes. Further, in MQTT, *ClientId* also acts like a web session cookie for restoring sessions. However, little has been done by both IoT platforms and manufacturers to make it a secret, as discovered in our research. Most confusing to the IoT cloud platforms could be the co-existence of platform identities, which indeed have been involved in various authentication and authorization processes (e.g., two-factor authentication, AWS cognito, SSO, etc.). However such protection just ensures that *only authenticated platform users can establish MQTT connections*, not that *only authorized users can claim ClientIds for the connections*. As a result, any authenticated user on the platform can use any *ClientId*.

Responsible disclosure. We reported our findings to all affected vendors such as iRobot, Microsoft, IBM, Tuya, Baidu and Suning, which all acknowledged the problem. Particularly, Microsoft awarded us through their bug bounty program.

D. Authorization Mystery of MQTT Topics

Due to the lack of guidance and the unique adversarial environment of IoT, the necessity to protect messages (Section III-A), sessions (Section III-B) and *ClientId* (Section III-C) are unclear to IoT vendors. However, even for MQTT entities that are clearly known to protect, such as MQTT topics, we found their protection is inadequate: IoT clouds take insecure shortcuts in protection, again due to the lack of guidance for securely adopting MQTT, especially in the adversarial environment of IoT; unsound description of protected MQTT topics may also grant a malicious user excessive access. In practice, such problems are found to be highly damaging to both security and privacy.

Insecure shortcut in protecting MQTT topics. As introduced earlier (Section II), IoT clouds add security control missing in MQTT. In particular, IoT clouds control what MQTT topics (associated with a specific device) a user is permitted to publish or subscribe to. Real-world authorization on IoT clouds that has to manage millions of users and devices from thousands of manufacturers, can be highly complicated, especially considering the commonplace user privilege changes. For ease of MQTT adoption, we found IoT clouds take shortcuts in building authorization. For example, Suning's (a Fortune Global 500 company [40]) IoT cloud serving more than 100 brands [39], [41], permits the user to subscribe to any MQTT topic he/she knows, based on an implicit assumption that MQTT topics are confidential.

However, such an assumption does not hold in the adversarial environment of IoT. As introduced in Section II-C, IoT adversary is oftentimes able to use the target device at least once (e.g., in hotel rooms and even private homes), and thus easily knows its topic (e.g., through traffic analysis). Also, we found device manufacturers tend to use an established unique identifier of the device as its MQTT topic, such as the device serial number or MAC address, which are subject to brute-force enumeration (Section III-C). Consequently, presenting the topic of the target device, the adversary is able to subscribe to all its messages without user consents. Such messages can include highly sensitive or private information depending on the devices, such as health statistics and conditions, purchasing preference and history, personable habits, household relations, etc.

Such privacy risks are introduced in the adversary environment of IoT, which may not even exist in the original usage scenarios of MQTT; for example, industrial proprietary devices transmit telemetry over the satellite [42]. Again, our finding shows that the *security gap* between the original and more trusted MQTT environment and the IoT environment is previously inadequately assessed and largely underestimated. Further, in the absence of guidance and standard for securely adopting MQTT in IoT environment, vendor-specific and unsound protection (like the Suning's above) tends to happen in the real world, considering the great number of IoT platforms in the market [43].

Attack. We implemented a PoC attack using our HONYAR Smart Plug IHC8340AL, with which we reverse-engineered the traffic between the Suning Smart Living mobile app and the cloud, and found its MQTT topics. Presenting the topics, we created a script and successfully subscribed to all subsequent messages of the device even after the device was reset and then used with another user account (i.e., representing the next person to use the device who is the victim).

Expressive syntax of MQTT. A device may have multi-

ple associated topics (e.g., `/deviceID/cmd` for delivering commands, and `/deviceID/status` for updating status). For ease of use, the permitted user of a device can use a wildcard character # or + to subscribe to multiple topics of the device or even multiple devices. In our study, we found that such expressive syntax of MQTT leaves tremendous space for IoT clouds to unwittingly put user privacy under high risks. As an example, the popular IoT cloud Suning Smart Living failed to properly authorize subscriptions of MQTT topics with wildcards. Specifically, any user of the platform is able to subscribe to the universal # topic of the IoT cloud, which by its definition in MQTT, means all MQTT topics on the broker (effectively all devices under the IoT cloud). Such a flaw can leak various highly critical privacy information of *all* IoT users on the cloud platform through their IoT devices, such as personally identifiable information (PII), health condition, household relation, personal habits, etc. (see our PoC experiment below).

Surprisingly, such a problem tends to be general because the flexible MQTT syntax makes it subtle in practice to soundly describe and interpret the resources to protect. Unsound resource description and interpretation in security policies practically allows the adversary to access unauthorized resources. In the above Suning example, likely Suning intends to allow the user only to access permitted topic described such as `deviceId/cmd` in its security policy; however, the requested wildcard-topic # can technically match the permitted one in the policy, leading to the policy bypassing. Such a *class* of problem was also confirmed on AWS IoT. Specifically, device manufacturers configure security policies for their users and devices, which are then enforced by AWS. We found that even a policy explicitly denies a user to access a topic described such as `deviceId/cmd`, AWS failed to interpret the protected target soundly and enables unauthorized access: a malicious user is able to subscribe to `deviceId/#` and receive messages from the protected topic. Apparently, AWS failed to soundly interpret the resource description that includes a wildcard and correlates it with the topic that should be protected.

Not only IoT clouds failed to soundly deal with the flexible resource-description syntax, device manufacturers also made similar mistakes. In particular, to easily permit the user to access multiple topics of her device, we found iRobot configures its security policy on AWS IoT with a (trailing) wildcard like `/[deviceModel]/[deviceId]/+`. Such a topic description in the policy is overly inclusive and even allows a malicious iRobot user to create an arbitrary topic that matches the path prefix, such as `/[deviceModel]/[deviceId]/ATTACK` (note that the `deviceId` and `deviceModel` are the Id and model of the malicious user's own device in the attack). This practically provides a hidden C&C channel: the malicious user can publish messages to this hidden topic, and all bots can subscribe to the topic and receive control commands, bearing the innocuous MQTT traffic to evade detection. Such a problem was inferred through our reverse-engineering of the traffic between our iRobot mobile app and AWS IoT, and was confirmed with iRobot, although we did not implement end-to-end attack experiments for ethical reasons. Also interestingly, all the communication through the topic will be charged to iRobot by AWS, because the broker that hosts the topic was deployed by iRobot. More importantly, such a practical, hidden, and money-saving C&C channel potentially has significant real-world impacts considering the recent devastating botnets such as Mirai [44].

For the first time, our findings show that to soundly describe and interpret MQTT topics are highly error-prone in practice, and in the adversarial environment of IoT, the mistakes can lead to devastating security, privacy and even direct financial breaches. Again, our study shows that to securely build IoT communication from the less protected MQTT protocol is much more challenging than previously expected, without the guidance of a standard that soundly considers IoT risks and subtle MQTT resources.

PoC experiment. We performed PoC experiment to validate the above problem on Suning's IoT cloud. The MQTT communication was TLS-encrypted on port 1885 of Suning Smart Living cloud's endpoint. Through a simple script, we authenticated our Suning account on the cloud and subscribed to topic # (the universal topic of the platform). Through the subscription, we received a vast number of privacy-critical messages from smart locks, cameras, home-security monitors, etc. Through a 3-week message collection (with IRB approval), we found the potential adversary was able to infer Suning users' household/cohabitation relations, behavioral habits, and even personally identifiable information (see details in Section IV-B).

Also surprisingly, the leaked messages include *ClientId*s of all devices under the cloud. Leveraging the *ClientId*-based DoS attack introduced in Section III-C, the potential adversary can easily kick arbitrary devices of arbitrary users offline; considering the user and the device type (e.g., a health or security monitor) can be identified from the message, the potential security and even safety risks are significant.

Responsible disclosure. We reported our findings to Suning, AWS, and iRobot who all acknowledged the problems and their severe security, privacy and financial implications.

IV. Measurement

To understand the scope and magnitude of the design defects found in our research, we conducted a measurement study on eight leading IoT clouds. The study brings to light the pervasiveness of these problems and their impacts, including the cost for executing these attacks (such as gathering device messages). Further we show that misleading guidance provided by the IoT clouds could contribute to the problematic security designs and implementations on the IoT manufacturer side.

A. Scope and Magnitude

Our measurement focused on leading IoT clouds that mediate devices of many high-profile manufacturers, e.g., AWS

IoT Core [1], IBM Watson IoT [45], Alibaba Cloud IoT Platform [46], Microsoft Azure IoT Hub [2], etc. (see Table I), which all rely on MQTT (version 3.1.1, at the time of our study) for their IoT communication. To assess each IoT cloud platform, we registered their accounts, leveraged public development documentation and SDKs, and built demo devices and apps for exercising its MQTT-based IoT communication and verifying its security protection against each problem we discovered – identity management (*ClientId* hijacking, see Section III-C), message authorization (*Will Message* and *Retained Message*, see Section III-A), session management (Section III-B) and topic authorization (Section III-D). Note that, two of our studied cloud platforms do not publish their SDKs (only available to paid device manufacturers), i.e., Tuya and Suning; hence, we purchased a few IoT devices mediated under the two to support our testing. Our measurement results are summarized in Table I and elaborated as follows.

• *Identity management*. The majority of tested cloud platforms suffered from the *ClientId* hijacking attack, including AWS, IBM, Microsoft, Tuya, etc. (Table I), which demonstrates the problem is general. As discussed in Section III-C, very likely the problem is due to the misguidance of MQTT specification and the lack of secure and standardized practices in managing MQTT *ClientId* in the IoT systems. In such unguided development environment, each platform implements its own version of IoT communication, and interestingly, some platforms' implementation renders the problem harder to exploit. In particular, each device under IBM IoT cloud has to use the *ClientId* pre-assigned under its factory setting, in the format `d:orgId:deviceType:deviceId`), in which the `deviceId` field must match the recorded ID of the device on the cloud. Hence, the attacker cannot exploit his/her device's MQTT connection to claim a victim's *ClientId*. In contrast, on the same cloud, the user client follows another format `a:orgId:appId`, in which the `appId` field can be arbitrarily set by the user. Hence, the attacker can exploit his/her user connection to claim a victim user's *ClientId* – our *ClientId* hijacking attack. Since the two types of MQTT clients (*device* and *user*) on IBM cloud have different security implications, we assessed them separately in our study (see the two sub-columns under the IBM column in Table I). Although IBM's device connection is hard to exploit, this is likely attributed to its *ClientId* deployment choice for devices – not an intentional security design that restricts the *ClientId* per MQTT client. Otherwise, IBM may have applied the same restriction to protect its user client.

Also interestingly, we observed that Alibaba (the fourth largest cloud provider in the world) did not follow MQTT specification: two MQTT clients on its IoT platform could claim the same *ClientId* without conflicts which rendered our exploit ineffective. Also on our mentionable list is that AWS IoT allows device manufacturers to specify customized security policy that is highly flexible [47] and supports to restrict what *ClientId* a client can claim. However, such a restriction was never suggested to device manufacturers and the exemplary policy recommended on AWS developer documentation is even subject to our attack. Appendix Section B reports our further study of insecure AWS IoT security policies created in the wild: the majority created by developers on Github was insecure, which further highlights the hard challenges in eliminating the threat and building a secure IoT ecosystem.

Our study shows that *ClientId* management is a general problem, and more importantly, each cloud adds their customized yet ineffective security protection to MQTT communication – such an ecosystem is highly insecure, alarming and in urgent need of standardized security guideline and practices.

• *Message authorization*. All IoT cloud platforms, e.g., AWS, Microsoft, Tuya, suffered from the problems except a few, of which the implementation did not support the two types of utility-oriented messages (*Will Message* and *Retained Message*) at the time of our study.

• *Session management*. Most IoT clouds suffered from the insecure session management (subscription and lifecycle states, see Section III-B), found in our study. Interestingly, to securely govern subscription state changes, AWS added their own permission policy, which is missed to consider in MQTT specification and by other IoT platforms. Still, AWS was subject to the session lifecycle attack. Again, this observation indicates the urgent need of consistent and standardized security guideline for protecting the many IoT platforms in their MQTT communication.

• *Topic authorization*. Leading IoT platforms AWS and Suning were subject to this class of attack. Others that we studied are not, because their current implementation has not aggressively taken advantage of the wildcard in MQTT topics, a convenient feature that enables the user to easily subscribe to multiple topics of her device or multiple devices of hers at a time. Given today's usability-oriented trend in system design [48], we envision that such a feature is appealing to these platforms. Once it is adopted, the security pitfalls can be non-trivial to avoid in practice, evidenced by the mistakes made by AWS, Suning, as well as device manufacturers (e.g., iRobot, see Section III-C).

Conclusion. We reported all security issues we found to related vendors which all acknowledged the problems. Our measurement shows that IoT clouds each builds their own MQTT communication and ad-hoc security protections, which are not effective and secure. This highlights the challenges in practice in securely adopting MQTT, a general messaging protocol designed in a more trusted environment, into the adversarial and complicated IoT environment, in the absence of a systematic analysis, well-thought design principles and security guidelines.

B. Privacy Implications of Leaked MQTT Messages

The weaknesses in Section III-B and III-D enable an attacker to stealthily gather MQTT messages published by victims' IoT devices . To understand whether message leakage in the real world indeed incurs serious consequences and

TABLE I
SUMMARY OF MEASUREMENT RESULTS

Security Weaknesses		Alibaba	AWS	Baidu	Google	IBM[1]		Microsoft	Suning	Tuya
ClientId Management		✓	✗	✗	✓	✓	✗	✗	✗	✗
Message Authorization	Will Message	N/A	✗	✗	N/A	N/A	✗	✗	N/A	✗
	Retained Message	N/A	N/A	✗	N/A	N/A	N/A	N/A	N/A	N/A
Topic Authorization		✓	✗	✓	✓	✓	✓	✓	✗	✓
Session Management	Subscription state	✗	✓	✗	N/A	N/A	✗	✗	✗	✗
	Lifecycle state	✓	✗	✗	✓	✓	✗	✗	✗	✗

✗means the weakness was successfully exploited on the platform. ✓means we were not able to exploit the weakness on the platform.
N/A means the platform did not fully support the MQTT feature; or its security policy was too coarse-grained for us to test the fine-grained aspect, e.g., the platform did not support to revoke a client's capability to subscribe, so we could not adequately test its management of "subscription state".
[1] The left and right columns under IBM show the results of testing using the *device* client and *user* client respectively.

privacy implications, we performed an experiment carefully approved by the IRB office of our university.

Specifically, due to the improper wildcard handling in MQTT *topic* subscriptions on Suning's cloud platform (Section III-D), any real-world attacker can easily subscribe (through wildcard /#) to messages from all devices under the cloud. In our experiment, we ran a simple script to subscribe to Suning's broker endpoint (a URI easily obtained through reverse engineering the traffic of our own device) and collected MQTT messages for three weeks (see IRB approval below).

In total, we successfully gathered 800 million real-world MQTT messages from the cloud. From such easily gathered messages, our findings are highly astonishing. Particularly, the messages published by individual IoT device include rich information, including device IDs and types (e.g., door lock, air conditioner, camera, etc.), device status ("on", "off", "heating", "washing"), timestamp, device location ("home", "office", "living room", etc.) and information captured by the device (e.g. indoor temperature, air quality, whether a person is passing by a camera, etc.). Some of the gathered messages include users' PII (e.g., email, phone number, nicknames, and names, etc., which were configured by the user on his/her device). Also interestingly, from the device ID and user ID, the potential attacker is able to infer the number of active users and devices during the three week period, a likely business secret of the platform which we decide not disclose in the paper.

More importantly, when the information is combined for a longitudinal analysis (three weeks in our study), a potential attacker likely could infer the private habits, routine behaviors, cohabitant relation, etc., of identifiable people (through their email addresses, phone numbers). For example, Fig. 4 shows the three-week status of a door lock: the user usually stays at home on weekdays since the door frequently opens from 9:40 to 19:10, and tends to leave home by Fridays since the door was not operated at all on two weekends during our study. Also, Fig. 5 in Appendix shows three-week status of an air conditioner that also leaks information about when the user is usually not at home. Hence, we envision such routine information (from door locks, air conditioner, fridge, bulb, coffer maker, oven) of a victim could help a potential burglar figure out the best time to break into a home. Further, we found the message from home locks like `"[Person Name set by user] opened the door via fingerprint"`, which possibly leaks the user's private cohabitants relation (e.g., who lives with a movie star).

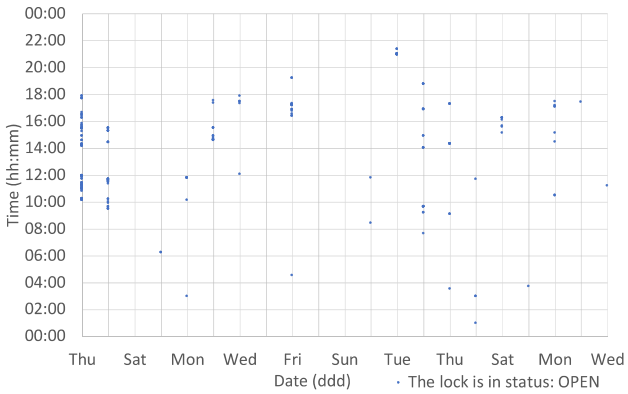

Fig. 4. "Open" Status of a Smart Lock Mediated Under Suning's Cloud

Consider the millions of consumers and medical IoT devices on the market and their sensitive functionalities, such as those for testing and monitoring pregnancy/blood pressure, depression, blood glucose, heartbeat, Parkinson disease, and user location, the potential privacy leakage demonstrated in our study deserves serious attention and calls for urgent protection.

IRB approval. To avoid the potential privacy infringement on real-world users, the design of our experiment was approved by our university's IRB. In particular, all data we collected was encrypted in transit (through TLS-enabled port 1885 of Suning broker server); any personally identifiable information (PII) in the message, e.g., email, phone number, was hashed through SHA-512 at rest.

V. MITIGATION

From our security analysis, we conclude that today's IoT clouds fail to bridge the gap between the usability-oriented design of MQTT and the security demands of complicated user-device interactions. Mitigating the risks introduced by the flaws requires enhancing protection of entities within the protocol to ensure that an IoT device can only be accessed by the party during the period that she/he is authorized. However, our findings show that such protection, though urgently needed in IoT environment, is out of the current security model of MQTT protocol, e.g., identities, messages and session states are generally unprotected or not suggested to protect.

Hence, key to the protection is to add the missing security model and design principles that govern the critical protocol entities. Hence, in this section, we present design principles for protecting the protocol entities and an enhanced access control model. We also report our implementation of the protection on Mosquitto and evaluation of its efficacy.

A. Managing Protocol Identities and Sessions

A key design principle in the adoption of a messaging protocol to the complicated and adversarial IoT systems is, *protocol-layer identity (e.g., ClientId), if any, should be authenticated; additionally, if the identity is used as a security token (e.g., session token), its confidentiality should be guaranteed*.

Our study shows that IoT cloud platforms generally soundly authenticate the MQTT connection through platform-layer identities (such as Amazon accounts, see Section III-C). Hence, a lightweight and effective protection for a typical IoT platform is, restrict an MQTT client's *ClientId* to literally match the client's platform identity p_user_id (or alternatively the message digest of the client's platform-layer authentication token, e.g., an OAuth [49] token or certificate). For example, a user's *ClientId* should start with her platform identity p_user_id. Such a design also respects the concurrency property of IoT applications: a user can have multiple clients of different *ClientId*s, and each is restricted through its prefix. This design, however, may not work under certain circumstances, e.g., a device client may have hard-coded *ClientId*. To address such cases, a more general approach is to maintain a mapping between the platform identity and its allowed.Any attempts to claim an unauthorized *ClientId* should be denied. We will leave the full design and evaluation of such a mapping to our future research.

Additionally, sessions in a messaging protocol should be guarded following the principle: *in the presence of an adversarial environment where subjects (e.g., a user) are expected to have privilege changes, session states, including protocol-agnostic states (e.g., lifecycle states) and protocol-specific states (e.g., subscription states), should accordingly keep updated in response*.

B. Message-Oriented Access Control Model

Key to securing a messaging protocol on IoT systems is to protect its message communication: *the system should govern the subjects' rights to send/receive messages, and additionally manage security implications in receiving a message with respect to the recipients' security requirement*.

As found in our study, the former is generally known to enforce, but the latter, especially protecting message recipients in the face of device sharing and privilege revocation (Section II-C), is completely out of the current consideration of MQTT. To bridge this gap, we propose `Message-Oriented Usage Control Model` (MOUCON), an enhanced access model for IoT communication based on UCON [50]. The core idea of MOUCON model is simple, and builds on familiar concepts: it takes the message as the resource (object), and checks access rights of the subject with respect to attributes of the subjects and objects. Its definition is as follows:

Subject (S). The set of the clients in the communication, such as devices and users. A subject is defined and represented by its attributes.

Subject Attributes (ATT(S)). A subject's attributes are specified as $ATT(S) = \{id, URI_w, URI_r\}$, and include identity information (id), the set of URIs [51] (e.g., topics) permitted to send messages to (URI_w), and URIs that are permitted to receive messages from (URI_r).

Object (O). The set of messages that subjects hold rights on.

Object Attributes (ATT(O)). An object's attributes are specified as $ATT(O) = \{content, URI, source\}$, and includes $content$ which is the application-layer information (e.g., message content), URI which represents the channel of the message (e.g. which MQTT topic the messages is published to or from), $source$ which represents the source of the object, i.e., the subject that created the message.

Rights (R). Rights are privileges that a subject can hold and exercise on an object. There are two general classes of rights, *Read (R)* (e.g., receive a message) and *Write (W)* (e.g., publish a message).

Authorizations. Authorization functions evaluate $ATT(S)$, $ATT(O)$, and requested rights together with a set of authorization rules for access decisions. For example, an authorization rule against message receiving by checking the attributes of both the message sender (source) and recipient is as follows:

$$allowed(s, o, R) \Rightarrow \\ (o.URI \in s.URI_r) \wedge (o.URI \in o.source.URI_w) \quad (1)$$

In equation (1), $allowed(s, o, R)$ indicates that subject s is allowed right R to object o (i.e., a recipient s is allowed to receive a message o). The decision is made by checking the condition at the right. In IoT scenarios, this rule ensures that, from message recipient's perspective, if a message's source ($o.source$, e.g., a user client that sent the message) has lost rights for accessing the topic where this message is from (by checking if $o.URI$ is not in $o.source.URI_w$), this message (e.g., *Will Message*) should be rejected. In addition, this rule also checks whether the recipient is allowed to receive the message by checking if the message's topic ($o.URI$) is in the allowed topic set $s.URI_r$. Such a rule bridges the gap for considering security implications of message recipients and can defeat our message-related attacks (Section III-A and Section III-B).

Discussion. MOUCON also supports other security policies. For example, we can define a rule to authorize a message sender based on its attribute URI_w, which specifies the topics it is allowed to send messages to. Further, a more complete solution should take into account how to properly update the sender and the recipient's privileges during an IoT system's runtime: for example, how a client's Read/Write privileges can be dynamically revoked, how to authorize such revocation requests by the cloud, etc. Down the road, we also plan to

extend the model to safeguard other messaging protocols such as CoAP when they operate in an IoT environment, e.g, by extending the attribute sets we define and corresponding authorization rules to enhance the protection of these protocols.

C. Implementation and Evaluation

We implemented the above protection on Mosquitto (version 1.5.4, which originally was not equipped with our protection). Specifically, we modified relevant data structures relating to its messages (struct `mosquitto_msg_store`) to add security-related attributes (e.g., a message's source), and added authorization functions to its broker (e.g., check message source before delivery). Also, we added the proposed *ClientId* restriction in the broker's existing access control function [52] used for establishing sessions.

We deployed our secured version of Mosquitto in lab environment and evaluated its effectiveness and performance overhead. Specifically, our secured Mosquitto and the original unprotected Mosquitto were respectively deployed on a workstation powered by 3.40GHz Intel i7-6700 CPU, 15.6G Memory and 475.3GB Disk. We also ran scripts on laptops (2.2 GHz Intel Core i7, 16 GB Memory, 251GB SSD and 2.3 GHz Intel Core i5, 16 GB Memory, 251GB SSD) that acted as MQTT clients. To evaluate the effectiveness of our protection, we launched all attacks in Section III: our secured Mosquitto defeated all attacks while the unprotected server did not capture any attack. To evaluate the performance overhead introduced by our added protection, against each server, we ran concurrent clients to publish messages and the same number of subscribers to receive messages. For each server, we recorded the average delay (in ms) between message publishing and receiving, and average CPU and memory usage in the period. We repeated the experiment for different numbers of concurrent publishing clients, ranging from 1000 to 8000. The results (detailed in Table II) show that our added protection incurs negligible messaging delay (at most 0.63%) and memory overhead (at most 0.16%). Although CPU usage is higher (around 10%), this is under normal expectation: adding critical security checks usually comes at a cost especially compared to a non-protected implementation.

VI. Discussion and Future Work

Lessons learnt. The most important lesson learnt from our study is the caution one should take when applying a utility-oriented, common-purpose protocol to the domains that may involve malicious parties. In such a case, both the scenarios the protocol does not cover and individual states of the protocol need to be evaluated to identify the gap between what the protocol can protect and what needs to be protected. More specifically in the use of general messaging protocols for IoT device-user communication, our study highlights that not only should ClientId and its related states be safeguarded with proper authentication and authorization, but also the whole revocation process, which is security critical, needs to be added to the protocol with protection in place. Note that although our research focuses on MQTT, due to its wide deployment, other messaging protocols such as Firehose [53], CoAP [54], AMQP [55], JoyLink [56] and Alink [57] may also have similar problems. Indeed, we discovered a similar revocation issue in the Firehose WebSocket on Samsung's Artik IoT cloud [53], with Samsung's acknowledgement: the cloud fails to terminate a client's session for receiving messages even after its privilege expires.

Mitigating such authentication/authorization flaws requires a joint effort from both the protocol designer and the IoT manufacturer. As mentioned earlier, though the MQTT TC starts enhancing the protocol against the threats we discovered, manufacturers also need to do their part. For this purpose, uniform interfaces that enable them to securely connect their devices and users to the cloud through different messaging protocols are valuable. Our approach makes a first step toward this end and is released online [14].

Automated discovery of the flaws. Based upon our understanding of the security pitfalls in applying generic messaging protocols to support IoT management, we believe that more systematic and automatic security analysis and vulnerability discovery are feasible. Using MQTT as an example, a possible approach is to semi-automatically construct state transitions of its operations, particularly those introduced by the protocol's IoT customization, and utilize model checking and/or guided manual analysis to detect security flaws. More specifically, the finite-state machine (FSM) of MQTT could be recovered using Natural Language Processing (NLP) through comparing MQTT specifications against the IoT cloud documents, as we did in our prior work [58] (on analysis of payment services). On the FSM, we could manually identify desired security properties, such as session state updates for access privilege revocation, and specify them using Linear Temporal Logic (LTL) [59]. The presence of the properties on the FSM could be evaluated using a model checker, e.g., SAL [60]. A problem reported by the checker could serve as a counter-example for finding security flaws on the MQTT-IoT system, by running the example against the system. In reality, however, NLP might not fully capture all details of an IoT protocol, due to its limitations and/or incomplete or inaccurate specifications in the protocol documentation. Also, a complicated FSM might render formal verification hard to succeed. In both cases, the vague, incomplete description or the complicated protocol logic could guide our analysis to focus on related implementation for finding security weaknesses. Note that this step could also be automated to some level, for example, through a guided fuzz test on protocol entities (e.g., clientID, topic, Will Message, etc.). Down the road, we envision that a line of research will happen in this direction, to automate discovery of complicated security flaws in IoT systems, particularly those built atop existing messaging protocols.

MQTT 5. The new version of MQTT specification, *MQTT5*, became an OASIS standard in March 2019 [18]. As far as we know, no public IoT cloud supports MQTT5 during the time of our research. Also, the new standard maintains all features in the previous version, so all of our attacks are expected to work

TABLE II
PERFORMANCE EVALUATION BY COMPARING OUR SECURED MOSQUITTO WITH THE UNPROTECTED MOSQUITTO IMPLEMENTATION

Clients Num	1000			2000			4000			6000			8000		
	Without Protection	With Protection	Overhead (%)	Without Protection	With Protection	Overhead (%)	Without Protection	With Protection	Overhead (%)	Without Protection	With Protection	Overhead (%)	Without Protection	With Protection	Overhead (%)
Delay (s)	1.432	1.441	0.63	1.450	1.456	0.40	1.456	1.462	0.41	1.458	1.459	0.06	1.466	1.471	0.34
CPU (%)	19.1	22.2	5.52	23.2	25.6	10.34	24.4	26.9	10.25	27.6	29.6	7.34	29.5	32.2	9.15
MEM (KB)	6725	6734	0.13	6736	6740	0.05	6752	6756	0.05	6872	6880	0.12	6883	6963	0.16

on IoT clouds using MQTT5 without extra protection. Also, the new specification does not provide protocol-layer solutions to address the security risks we discovered. Even though its non-normative sections mention that the client needs to be authorized to use a ClientId and an implementation should provide access controls to restrict the client's capability to publish or subscribe to particular topics, it offers no recommendation on how such protection should be enforced in the IoT scenarios we studied. We reported our findings to the OASIS MQTT TC, which acknowledged that our findings affect both version 5 and 3.1.1, and the issues are under open discussion now [11].

VII. RELATED WORK

Security studies on MQTT. Industry reports showed that the general MQTT protocol lacks basic authentication and authorization. Without any protection such as those already added by IoT clouds today, [61] showed that an attacker was able to connect to arbitrary MQTT servers without authentication and subscribe to any topics. Another industry report [62] scanned exposed MQTT server endpoints on the Internet and implementation bugs (in handling character encoding) in MQTT libraries and broker implementation. In contrast to these works that studied completely unprotected MQTT implementation, we focus on the protection today's mainstream IoT clouds put in place in adopting MQTT and their insufficiency in eliminating new risks unique to IoT environments.

Academic studies proposed to enhance the security model of MQTT due to its lack of basic security consideration in the protocol design. [63] proposed new mechanisms for distributing temporary keys to MQTT clients; [64] proposed to enhance security protection through considering clients' context information, such as IP address; [65] proposed dynamic control model that equips MQTT by monitoring mutable attributes for making access decisions; [66] proposed to introduce OAuth into MQTT environment as a security addition. In contrast to the previous works, our study focused on real-world IoT platforms, the weaknesses of their added security protection in MQTT deployment, and analyzed practical security challenges and pitfalls IoT vendors are facing today.

IoT platform security. The security of IoT platforms has been extensively studied, such as in [30], [67]–[72]. [70] conducted the first security analysis of Samsung's SmartThings platform and discovered its coarse-grained access control design. [30] studied the interactions between IoT devices, mobile apps, and clouds and focused on weaknesses in state transition diagrams of the three entities. Works have also been proposed to protect IoT systems. [73]–[75] studied how to restrict the capabilities of the cloud, mobile application and device; [76] tracks and protect sensitive IoT information; [77] proposed a provenance-based framework to aggregate device activities for detecting errors and malicious activities. In sharp contrast, our study of IoT focuses on the unique aspects of the underlying messaging protocol, i.e., MQTT, and the insufficient security protection added by IoT clouds in adopting MQTT to the complicated and adversarial device-user interactions.

VIII. CONCLUSION

We performed the first systematic study on security risks in the use of the general messaging protocol for IoT device-user communication. Our research reveals the gap between the protocol designed for operating in a simple and benign environment and the complicated, adversarial IoT scenarios, and the challenges in covering the gap with proper security means. From the findings, we generalized new design principles and proposed an enhanced access model. Our protection is implemented and its high effectiveness and efficiency are evaluated. Our new findings and protection will lead to better protection of user-device interactions in the real world.

ACKNOWLEDGMENTS

We would like to thank our shepherd Nick Feamster, the anonymous reviewers and MQTT TC for their insightful comments. Yan Jia, Yuhang Mao, Shangru Zhao and Yuqing Zhang were supported by National Natural Science Foundation of China (No.U1836210, No.61572460) and in part by China Scholarship Council. The IU authors were supported in part by Indiana University FRSP-SF, and the NSF CNS-1618493, 1801432 and 1838083.

REFERENCES

[1] A. W. S. (AWS), "Aws iot core," https://aws.amazon.com/iot-core/, accessed: 2019-01.
[2] "Azure iot hub," https://azure.microsoft.com/en-us/services/iot-hub/, accessed: 2019-01.
[3] "Samsung smartthings," https://www.smartthings.com/, accessed: 2019-01.
[4] OASIS, "Mqtt version 3.1.1 plus errata 01," http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, 2015, accessed: 2019-01.
[5] "Key trends from the iot developer survey 2018," https://blog.benjamin-cabe.com/2018/04/17/key-trends-iot-developer-survey-2018, accessed: 2019-07.
[6] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur, "Rethinking access control and authentication for the home internet of things (iot)," in 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD, 2018, pp. 255–272.
[7] "Hilton announces connected room, the first mobile-centric hotel room, to begin rollout in 2018," https://newsroom.hilton.com/corporate/news/hilton-announces-connected-room-the-first-mobilecentric-hotel-room-to-begin-rollout-in-2018, 2017, accessed: 2019-01.
[8] "Smart hosting: The dos and don'ts of the ultimate airbnb smart home," https://www.the-ambient.com/guides/host-smart-airbnb-home-tech-217, 2018, accessed: 2019-01.

[9] "The best smart lock for airbnb hosts," https://www.remotelock.com/smart-locks-airbnb-hosts, accessed: 2019-01.

[10] "Smar home technology can increase your earning potential," https://learnairbnb.com/smart-home-technology/, 2018, accessed: 2019-01.

[11] "Revocation of authority to publish and subscribe," https://issues.oasis-open.org/projects/MQTT/issues/MQTT-536?filter=allopenissues, accessed: 2019-010.

[12] "Demos of iot vulnerabilities exploits," https://sites.google.com/view/attackdemos/, accessed: 2019-01.

[13] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," Journal of Open Source Software, vol. 2, no. 13, 2017.

[14] "Mitigationdemo," https://github.com/user-online/mosquitto-defense/tree/master, accessed: 2019-010.

[15] AWS, "Iot customer success stories," https://aws.amazon.com/solutions/case-studies/iot/, accessed: 2019-01.

[16] T. Inc., "Cooperation we have already provided one-stop iot services for 93,000+ manufacturers and brands," https://en.tuya.com/cooperation, accessed: 2019-01.

[17] "Publish–subscribe pattern," https://en.wikipedia.org/wiki/Publish-subscribe_pattern, accessed: 2019-07.

[18] OASIS, "Mqtt version 5.0 - candidate oasis standard 01," https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html, 2018, accessed: 2019-01.

[19] "Osi model," https://en.wikipedia.org/wiki/OSI_model, accessed: 2019-05.

[20] IETF, "The websocket protocol," https://tools.ietf.org/html/rfc6455, 2011, accessed: 2019-01.

[21] "Transport layer security," https://en.wikipedia.org/wiki/Transport_Layer_Security, accessed: 2019-07.

[22] "Single sign-on," https://en.wikipedia.org/wiki/Single_sign-on, accessed: 2019-07.

[23] "Amazon cognito," https://aws.amazon.com/cognito/, accessed: 2019-07.

[24] "Client-authenticated tls handshake," https://en.wikipedia.org/wiki/Transport_Layer_Security#Client-authenticated_TLS_handshake, accessed: 2019-01.

[25] "Baidu rises to top 5 in idc ranking of public cloud services providers in china," https://www.bloomberg.com/press-releases/2019-05-08/baidu-rises-to-top-5-in-idc-ranking-of-public-cloud-services-providers-in-china, accessed: 2019-07.

[26] "The top 500 sites on the web," https://www.alexa.com/topsites, accessed: 2019-07.

[27] "Eclipse mosquitto - an open source mqtt broker," https://github.com/eclipse/mosquitto/, accessed: 2019-07.

[28] "Home assistant mqtt brokers," https://www.home-assistant.io/docs/mqtt/broker#embedded-broker/, accessed: 2019-06.

[29] "Mqtt arrives in the modern openhab 2.x architecture," https://www.openhab.org/blog/2018-12-16-mqtt-arrives-in-the-modern-openhab-2-x-architecture.html, accessed: 2019-06.

[30] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1133–1150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zhou

[31] J. Chen, C. Zuo, W. Diao, S. Dong, Q. Zhao, M. Sun, Z. Lin, Y. Zhang, and K. Zhang, "Your iots are (not) mine: On the remote binding between iot devices and users," in 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2019, pp. 222–233.

[32] IFTTT, "Ifttt," https://ifttt.com/, accessed: 2019-01.

[33] "Tuya smart," https://en.tuya.com/company, accessed: 2019-01.

[34] "The mqtt client identifier," https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/tt60310_.htm, accessed: 2019-06.

[35] "Mqtt security fundamentals: Advanced authentication mechanisms," https://www.hivemq.com/blog/mqtt-security-fundamentals-advanced-authentication-mechanisms/, accessed: 2019-06.

[36] "Device integration using mqtt," https://cumulocity.com/guides/device-sdk/mqtt/, accessed: 2019-06.

[37] "Serial number," https://en.wikipedia.org/wiki/Serial_number#Applications_of_serial_numbering, accessed: 2019-06.

[38] "Decode the meaning behind your apple serial number," https://beetstech.com/blog/decode-meaning-behind-apple-serial-number, accessed: 2019-06.

[39] "Suning showed on ces," https://news.suning.com/wtoutiao/witem2_1548553411.html, accessed: 2019-07.

[40] "Fortune global 500," https://fortune.com/global500/suning-commerce-group/, accessed: 2019-07.

[41] "Suning smart living," https://smarthome.suning.com/shop_index.html, accessed: 2019-01.

[42] "Getting to know mqtt," https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot, accessed: 2019-01.

[43] "Iot platforms vendor comparison 2018," https://iot-analytics.com/product/iot-platforms-vendor-comparison-2018, accessed: 2019-01.

[44] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[45] "Ibm iot platform," https://www.ibm.com/internet-of-things/solutions/iot-platform, accessed: 2019-01.

[46] "Alibaba cloud iot platform," https://www.alibabacloud.com/product/iot, accessed: 2019-01.

[47] "Aws iot policies," https://docs.aws.amazon.com/iot/latest/developerguide/iot-policies.html, accessed: 2019-07.

[48] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu, "Staying secure and unprepared: Understanding and mitigating the security risks of apple zeroconf," 2016.

[49] "Oauth," https://en.wikipedia.org/wiki/OAuth.

[50] J. Park and R. Sandhu, "The ucon abc usage control model," ACM Transactions on Information and System Security (TISSEC), vol. 7, no. 1, pp. 128–174, 2004.

[51] "Uniform resource identifier," https://en.wikipedia.org/wiki/Uniform_Resource_Identifier, accessed: 2019-01.

[52] "mosquitto.conf man page," https://mosquitto.org/man/mosquitto-conf-5.html, accessed: 2019-05.

[53] "Start building with artik cloud services," https://developer.artik.cloud/, accessed: 2019-01.

[54] "The constrained application protocol (coap)," https://tools.ietf.org/html/rfc7252/, accessed: 2019-01.

[55] "Advanced message queuing protocol," https://www.amqp.org/, accessed: 2019-01.

[56] H. Liu, C. Li, X. Jin, J. Li, Y. Zhang, and D. Gu, "Smart solution, poor protection: An empirical study of security and privacy issues in developing and deploying smart home devices," in Proceedings of the 2017 Workshop on Internet of Things Security and Privacy, ser. IoTS&P '17. New York, NY, USA: ACM, 2017, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/3139937.3139948

[57] "Communications over alink protocol," https://www.alibabacloud.com/help/doc-detail/90459.htm, accessed: 2019-01.

[58] Y. Chen, L. Xing, Y. Qin, X. Liao, X. Wang, K. Chen, and W. Zou, "Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis," in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 747–764. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yi

[59] "Linear temporal logic," https://en.wikipedia.org/wiki/Linear_temporal_logic.

[60] "Symbolic analysis laboratory," http://sal.csl.sri.com/.

[61] L. Lundgren, "Taking over the world through mqtt—aftermath," https://www.blackhat.com/docs/us-17/thursday/us-17-Lundgren-Taking-Over-The-World-Through-Mqtt-Aftermath.pdf, 2017, accessed: 2019-01.

[62] D. Q. Federico Maggi, Rainer Vosseler, "The fragility of industrial iot's data backbonesecurity and privacy issues in mqtt and coap protocols," https://documents.trendmicro.com/assets/white_papers/wp-the-fragility-of-industrial-IoTs-data-backbone.pdf?v1, 2018, accessed: 2019-01.

[63] A. Rizzardi, S. Sicari, D. Miorandi, and A. Coen-Porisini, "Aups: An open source authenticated publish/subscribe system for the internet of things," Information Systems, vol. 62, pp. 29–41, 2016.

[64] R. Neisse, G. Steri, and G. Baldini, "Enforcement of security policy rules for the internet of things," in Wireless and mobile computing, networking and communications (wimob), 2014 IEEE 10th international conference on. IEEE, 2014, pp. 165–172.
[65] A. La Marra, F. Martinelli, P. Mori, A. Rizos, and A. Saracino, "Introducing usage control in mqtt," in Computer Security. Springer, 2017, pp. 35–43.
[66] A. Niruntasukrat, C. Issariyapat, P. Pongpaibool, K. Meesublak, P. Aiumsupucgul, and A. Panya, "Authorization mechanism for mqtt-based internet of things," in Communications Workshops (ICC), 2016 IEEE International Conference on. IEEE, 2016, pp. 290–295.
[67] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot." in NDSS, 2019.
[68] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018, pp. 1102–1119.
[69] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," arXiv preprint arXiv:1802.08307, 2018.
[70] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 636–654.
[71] M. Ambrosin, M. Conti, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, "Sciot: A secure and scalable end-to-end management framework for iot devices," in European Symposium on Research in Computer Security. Springer, 2018, pp. 595–617.
[72] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "Homonit: Monitoring smart home apps from encrypted traffic," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018, pp. 1074–1088.
[73] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action iot platforms," in Proc. Network and Distributed Systems Symposium (NDSS), 2018, pp. 18–21.
[74] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, 2017, pp. 361–378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tian
[75] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity, "Contexlot: Towards providing contextual integrity to appified iot platforms." in NDSS, 2017.
[76] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks." in USENIX Security Symposium, 2016, pp. 531–548.
[77] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in ISOC NDSS, 2018.
[78] A. W. Services, "Iot for non-ip devices," https://aws.amazon.com/blogs/iot/iot-for-non-ip-devices-2/, accessed: 2019-01.
[79] "Github," https://github.com/, accessed: 2019-01.
[80] AWS, "Iam and aws sts condition context keys," https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_iam-condition-keys.html, accessed: 2019-01.

APPENDIX

A. Inferring User's Routine from Leaked MQTT Messages

Fig. 5 shows the three-week status of an air conditioner, in which red points indicate the time when the air conditioner is heating. From the chart, a potential attacker can infer that the air conditioner usually starts at about 6:45 AM on weekdays, 7:45 AM in weekends and stops between 10 PM and midnight, which likely indicate roughly when the user gets up in the morning and goes to sleep in the night. Also interestingly, the user very likely was at home most of the days, except the Thursday in the second week and between 12pm - 4pm from the first Saturday to the second Friday – during the periods the air conditioner did not work.

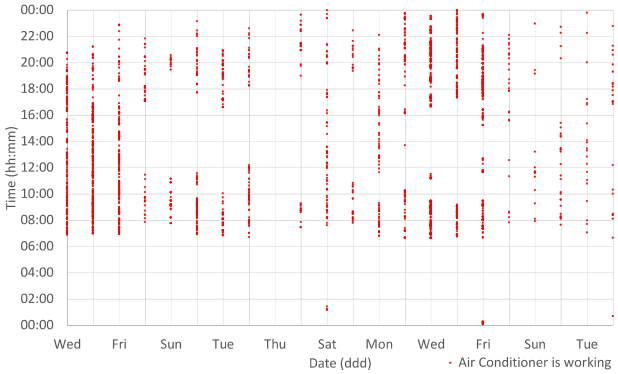

Fig. 5. Real air conditioning status of a user

B. Misleading Development Guide

On AWS IoT, device manufacturers can customize the AWS IoT policy (a security policy template, which is populated at runtime and assigned to each client) to restrict the *ClientId* of a client, so as to mitigate our ClientId-based attack (section III-C). However, the secure manner to do so is not clear to device manufacturers, since we found that the exemplary security policies recommended on AWS developer documentation and other popular IoT development guides on the Internet are even subject to our attack. Further, the majority of AWS IoT security policies we found that have been created by developers in the wild (i.e., on GitHub) is vulnerable to our attack, as elaborated below.

Through manually inspecting AWS' official developer guide and blogs, we gathered 38 exemplary/best-practice AWS IoT policies provided by AWS. Unfortunately, AWS never recommended to restrict the *ClientId* and 26 out of the 38 recommended policies we found were vulnerable. For example, Listing 1 illustrates an IoT policy from AWS official blog [78], in which the `Resource` field specifies the *ClientId* that is permitted to use when the client connects. In the template-like policy, the variable `${iot:ClientId}` will be populated at runtime to be the actually claimed *ClientId* of a client. This template feature can help a device manufacturer avoid hard coding specific *ClientId* in its IoT policy, which can be applied to a wide spectrum of clients at runtime. Such a policy, however, does not restrict what *ClientId* a client can use, and, hence, vulnerable to our attack. We reported the misleading, insecure IoT policies to AWS, who acknowledged the problem.

To understand whether the problematic policy samples provided by AWS indeed misguided IoT developers in the wild, we located 89 open-source projects on Github [79] that included an AWS IoT policy, and assessed their security qualities. Unfortunately, 85.4% (76/89) of the projects included an IoT policy that is vulnerable to our attack.

Listing 1. An Example of Vulnerable AWS IoT Policy

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
```

```
        "Action": ["iot:Connect"],
        "Resource": ["arn:aws:iot:us-east
            -1:000000000000:client/${iot:ClientId}"]
    }]
}
```

Further, although AWS IoT policy supports to specify the permitted *ClientId* of a client, in practice, how to properly define a secure policy is still unclear to device manufacturers and developers. In our study, we thoroughly inspected AWS developer guide and constructed an IoT policy that allows the manufacturer to mitigate our attack, as shown in Listing 2. Specifically, when this policy is populated at runtime, its variable `${cognito-identity.amazonaws.com:sub}` will be instantiated as the Amazon Cognito ID [80] of the client – the client's platform-layer identity. Effectively, this restricts the *ClientId* of the client to be equal to its authenticated platform-layer identity, and, thus, defeats our attack. Nevertheless, this policy has a usability problem, which very likely prevents its wide adoption. Specifically, an IoT user tends to have multiple clients (e.g., on her mobile phone, tablet, and laptop, etc.), which all would have to present the same *ClientId* – the Amazon Cognito ID of the user account – and, hence, will kick each other offline (see the conflicting *ClientId* in Section III-C). To effectively mitigate the security risks on AWS IoT, we envision a solution that is in line with Section V-A: the *ClientId* a client can use has to start with the client's platform-layer identity (with a suffix to differentiate different, legitimate clients).

Listing 2. A Secure AWS IoT Policy Example

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "iot:Connect"
        ],
        "Resource": [
            "arn:aws:iot:*:*:client/${cognito-
                identity.amazonaws.com:sub}"
        ]
    }]
}
```