Perils and Mitigation of Security Risks of Cooperation in Mobile-as-a-Gateway IoT

Xin'an Zhou
xzhou114@ucr.edu
University of California, Riverside
Riverside, California, USA

Jiale Guan
guanjia@iu.edu
Indiana University Bloomington
Bloomington, Indiana, USA

Luyi Xing
luyixing@indiana.edu
Indiana University Bloomington
Bloomington, Indiana, USA

Zhiyun Qian
zhiyunq@cs.ucr.edu
University of California, Riverside
Riverside, California, USA

ABSTRACT

Mobile-as-a-Gateway (MaaG) is a popular feature using mobile devices as gateways to connect IoT devices to cloud services for management. MaaG IoT access control systems support remote access sharing/revocation while allowing "offline availability" for better usability. Realizing these functionalities requires secure cooperation among the cloud service, the companion app, and the IoT device. For practical considerations, we find that almost all cloud services perform access model translation (AMT) to translate expressive cloud-side access policies to simple device-side policies. During the process, ad-hoc protocols are developed to support the access policy synchronization. Unfortunately, current MaaG IoT systems fail to recognize the security risks in the process of access model translation and synchronization. We analyze ten top-of-the-line MaaG IoT devices and find that all of them have serious vulnerabilities, e.g., allowing irrevocable and permanent access for temporary users. We further propose a secure protocol design that defends against all identified attacks.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; *Software reverse engineering*; • **Networks → Network architectures**.

KEYWORDS

IoT; Access Control; Attack; Protocol; Formal Proof

ACM Reference Format:
Xin'an Zhou, Jiale Guan, Luyi Xing, and Zhiyun Qian. 2022. Perils and Mitigation of Security Risks of Cooperation in Mobile-as-a-Gateway IoT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560590

1 INTRODUCTION

IoT devices are featured varying Internet connectivity capabilities and paradigms. Many prior works [54] studied IoT devices that are not connected to a server/cloud at all and such devices are managed by local consoles, such as a mobile phone. Recent works [20, 22–25, 29, 30, 32, 38, 40, 51, 56, 58, 68, 70, 82, 85] studied IoT devices that leverage the modern IoT cloud infrastructure for convenient access management and deployment, and such devices are connected to the cloud/Internet either through built-in Wi-Fi/cellular modules, or through a local Internet-connected IoT hub, such as a Bluetooth-capable, Zigbee or Z-Wave compatible IoT hub. Access to the devices (e.g., operation commands sent from the user's mobile phone) goes through the cloud for centralized mediation and access control.

Less studied is an emerging category of IoT devices that aims to leverage the modern IoT cloud infrastructure (e.g., for centralized user management, convenient device firmware updates), but lacks persistent Internet connectivity. For reduced power consumption, manufacturing cost, and maintenance cost, such devices are not built with Wi-Fi/cellular modules (Wi-Fi is shown to consume 7x the power compared to Bluetooth [59]), nor do they require a persistent, dedicated Internet-connected IoT hub to connect to the cloud. Rather, these devices leverage users' mobile phones to act as "Internet gateways" that relay information to and from the cloud when the phones are nearby. We call such a paradigm Mobile-as-a-Gateway (MaaG) IoT (Figure 1). Despite the popularity of MaaG IoT [15, 77, 78], its security and privacy risks have yet to be fully understood, not to mention adequately mitigated.

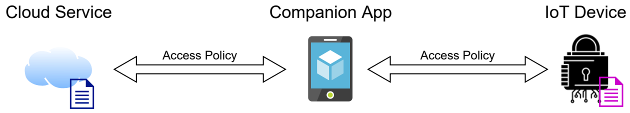

Figure 1: The MaaG IoT Architecture

New security challenges inherent with MaaG IoT. Prior works [13, 36, 50] studied Device-Gateway-Cloud (DGC) IoT, which is similar to the MaaG IoT architecture studied in this paper. In particular, [13, 36] show that it is difficult to synchronize a permission-revocation policy from the cloud to IoT devices in the presence of network partitions (e.g., due to intentional blocking of devices' access to the Internet, or internet outage). However, modern MaaG IoT in the wild has become much more complicated (than DGC) whose design comes with new fundamental security challenges. In particular, different from the previously studied cloud-based IoT

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560590

in which devices maintain persistent Internet connections with the cloud and all commands/requests to the devices are access-controlled by the IoT cloud [13, 50], in MaaG IoT, the devices and the cloud *split security responsibilities* and thus need to coordinate their security control to ensure the overall system is secure (unauthorized/unexpected users should not operate the devices). For example, popular IoT locks such as Kwikset Aura [2] manage/maintain access code (used for unlocking) that is not shared with the cloud (so a compromise of the cloud or communication channels will not compromise the home's physical security). In the meantime, the IoT devices leverage the IoT cloud for complicated user management (user roles, permission, delegation, revocation, etc., see Section 3). However, the proper security coordination among the cloud service, the companion app, and the IoT device turns out to be difficult to make right. Due to the lack of proper definition of access control models in the cloud and devices for MaaG IoT (the models often differ between the two), unclear and unsound results of their combined control efforts can arise (see below).

Also challenging is the lack of proper consistency model for the access policies between the cloud and IoT devices. MaaG IoT is inherently network-partitioned [13] and devices often are not connected to the cloud to fetch the latest access policies (e.g., revocation of a certain user). For example, one might think an "eventual consistency" model [36] (a benign user can help synchronize access policies after a malicious attacker's access is revoked) can be a reasonable model. However, we find that it is tricky and error-prone to implement such models for MaaG IoT. Further, even after the devices are connected to the cloud through the gateway, the necessary policies to synchronize between the cloud and devices are never properly defined, easily leaving real-world devices under insecure states (Section 5). The security challenges come with serious real-world security, privacy, and safety implications, but were never systematically studied.

Security analysis and real-world flaws. To understand the security implications of real-world MaaG IoT, we pick ten top-of-the-line MaaG IoT devices. After reverse engineering their custom protocols, we systematically recovered their conceptual models related to Access Model Translation and Synchronization. These lead to discovering several classes of security flaws that can have serious consequences. They include (1) allowing a temporary user retaining permanent access to the device; (2) allowing a temporary user to share her access to other unauthorized users; (3) allowing a temporary user to escalate her privilege.

Even though similar forms of attacks have been demonstrated in the past [36, 50], we wish to point out that the extent of the consequences and the root causes they rely on are substantially different. For example, [36, 50] have demonstrated that a malicious temporary user can retain access to a smart lock for as long as they can block the lock from synchronizing the access control policy with the cloud — if a benign user is able to help the lock synchronize, then the temporary user will lose access. However, in the attacks we demonstrate, a temporary user can retain access forever even after the permission is revoked at the cloud and the corresponding policy being successfully synchronized with the IoT device. The stronger results are obtained because of our superior understanding of the fundamental operating model of MaaG IoT.

Table 1: Security models of different MaaG IoT devices

MaaG IoT device	Access Management	Offline Availability
Level [8]	Remote	Guest & Admin
August [1]	Remote	Admin only
Yale [11]	Remote	Admin only
Ultraloq [10]	Local	Admin only
Kwikset Aura [2]	Remote	Guest & Admin
Honeywell [6]	Remote	Guest & Admin
Schlage [9]	Remote	Guest & Admin
Geonfino [5]	Remote	Guest & Admin
Tile [4]	Remote	Guest & Admin
Chipolo [3]	Remote	Guest & Admin

Secure protocol design. Facing the security challenges of MaaG IoT, we distill common design goals to avoid the pitfalls that we witnessed in real-world systems. We then design a coherent access control model/mechanism, and a novel lightweight protocol to securely synchronize access policies between a cloud service and an IoT device without trusting the gateway, which can defend the attacks we have discovered.

Contributions. We summarize the contributions of the paper as:
- We distill and formulate the unique security challenges of MaaG IoT, from the two main aspects: access model translation and access policy synchronization.
- We study ten top-of-the-line MaaG IoT devices and discover a number of weaknesses. They allow us to develop a number of attacks, achieving stronger consequences than previous results.
- We design and implement a secure protocol that is tailored to MaaG IoT. It avoids all the common pitfalls and is lightweight.

2 BACKGROUND

In this section, we first introduce two major and maybe contradicting functionalities of MaaG IoT devices: remote access sharing/revocation and offline availability. We then describe the common workflow of MaaG IoT.

2.1 Remote Access Sharing/Revocation and Offline Availability

As of 2022, we observe that all MaaG IoT access control systems in this paper have tried to enable access sharing/revocation and offline availability. However, different systems employ slightly different security models, as shown in Table 1.

We can see that all devices except Ultraloq can natively support remote access sharing/revocation. It means that a (privileged) user doesn't need to physically approach the IoT device to share/revoke access to/from an invitee: she can simply add that invitee to the cloud-side access control policy [34]. The invitee's app can later ask the cloud to endorse its eligibility to access the device, e.g., by authenticating to the cloud to obtain an access token/credential. In contrast, in local access sharing/revocation, a (privileged) user has to physically approach the IoT device to share/revoke access to/from an invitee.

We can also see that all devices support offline availability: users may access the device even when their apps do not have Internet connections. This can be done, for example, using access tokens/credentials that are recognized by the IoT devices directly. Offline availability has become an indispensable feature for smart locks because otherwise Internet outages or server downtime can lock out residents. It is interesting to see that some devices allow offline availability for low-privileged guest users while others allow only high-privileged administrative users. In any case, this feature is prevalent.

2.2 Common Workflow of MaaG IoT

To implement the two functionalities, IoT device manufacturers deploy cloud services to maintain generally authoritative and up-to-date access policies in order to decide whether a user is allowed to access a device at a certain time. However, an IoT device can also have its own on-device access control model and mechanism, which are required by the device to decide whether a user agent (i.e., the companion app) can have access, as well as to implement offline availability which can improve usability as mentioned earlier.

With the MaaG IoT architecture, the device manufacturer's official IoT device companion app acts as the gateway (also, a user agent) between the cloud and the device [15, 77, 78], and translates responses from the cloud into application-layer messages understood by the device. These messages are delivered to the device using wireless protocols such as Bluetooth Low Energy (BLE). A device may also have a feedback mechanism to inform the cloud that a specific access policy update has been applied. Currently, ad-hoc access policy synchronization protocols are designed by different IoT device manufacturers to implement remote access sharing/revocation and offline availability.

Figure 1 illustrates that with the MaaG IoT architecture, the cloud service and the IoT device have different but closely related access control models/mechanisms. For example, an MaaG IoT device may have offline access code, allowing device access without using the companion app, that is not synchronized to the cloud at all. The cloud service, companion app, and the IoT device have to cooperate to securely synchronize access policies between the cloud service and the IoT device in order to ensure secure access control. In the next section, we will show that the MaaG IoT architecture has subtle and serious security implications.

3 SECURITY RISKS IN COOPERATION OF MAAG IOT

Overview. A trustworthy MaaG IoT system needs security cooperation and coordination among the cloud, the companion app, and the IoT device. In MaaG, the cloud generally serves as the authority to issue/manage policies and the device is the party to enforce the policy when a user attempts to operate the IoT device (e.g., to unlock a smart lock). The cloud can support/manage increasingly complicated access control semantics and models, such as complicated user roles [64], delegation relations between users [44, 76], and grouped or location-based permissions [61, 65]. In the meantime, it is difficult for the device to maintain access semantics/policies of the cloud's complexity, which can be overly complicated and expensive, e.g., considering power consumption [59, 81], delays/difficulties in firmware updates [35, 57, 73], or even the increasing cost of software development [43]. For example, a smart lock may not need to be aware of users' delegation relations (e.g., whether user A's permission is granted by user B), as long as it can serve user A when he attempts to operate the lock while denying unexpected users. To support the access control in MaaG IoT, real-world manufacturers developed a set of protocols to translate the semantic-rich cloud-side access models/policies to lighter weight device-side access models/policies, presenting a mechanism which we call *access model translation (AMT)*. Section 4 reports the first and most in-depth security analysis of real-world AMT processes, and reveals the fundamental security design challenges with our end-to-end attacks.

Further, the cloud intends to keep the device-side policies in sync with the cloud, although this is difficult since MaaG IoT essentially is featured with network partition and weak consistency. Section 5 shows that the prior "eventual consistency" model [69] for data synchronization in distributed systems bestows low security assurance for modern MaaG IoT, and real world vendors and stakeholders failed to fully understand and come up with a sufficient access policy consistency model between the cloud and the MaaG IoT device, leaving tremendous space for new attacks.

In this paper, we summarize the flaws we discover into two classes related to access model translation (Section 4) and access policy synchronization (Section 5) in the MaaG IoT scenario.

Threat model. First, we assume the cloud service and the IoT device are trusted. We also assume the mobile device and the companion app of the legitimate user (e.g., owner) are trusted (e.g., free of malware). However, we assume the companion app and mobile operating systems of the attacker (e.g., temporary and low-privileged users such as invited guests) can be arbitrarily tampered. For example, an attacker can root/jailbreak his own smartphone [74, 84], reverse engineer the publicly available companion app, and modify the app. More specifically, the attacker can read any user manuals or developer-facing APIs (if any) and understand the protocol/interactions between the device, the companion app, and the cloud service. This means that the attacker can arbitrarily replicate and change the logic of the companion app to interact with the device and the cloud. We do not assume the attacker can inspect or alter either the device firmware (e.g., not even a factory reset) or the cloud-side code [71].

The goal of a malicious temporary user is to retain her access for as long as possible, to distribute such access further, or even to escalate her privilege.

Responsible Disclosure. At the time of writing, we have reported every product vulnerability in this work to related vendors, and we have received acknowledgements from seven vendors. Three vulnerabilities already have unpublished CVE numbers, and four vendors have already patched their vulnerabilities (e.g., August/Yale, Level, and Geonfino).

4 RISKS IN ACCESS MODEL TRANSLATION

A key security challenge in MaaG IoT is how the IoT cloud can translate modern semantic-rich security policies/models to lighter weight policies for the device side to enforce. By studying a set of

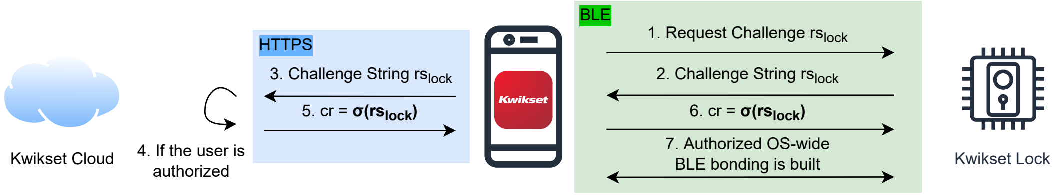
Kwikset Cloud
4. If the user is authorized
HTTPS
3. Challenge String rs_{lock}
5. cr = $\sigma(rs_{lock})$
Kwikset
BLE
1. Request Challenge rs_{lock}
2. Challenge String rs_{lock}
6. cr = $\sigma(rs_{lock})$
7. Authorized OS-wide BLE bonding is built
Kwikset Lock

Figure 2: Kwikset AMT process

ten popular MaaG IoT devices, our study shows that it is generally difficult for mainstream IoT manufacturers to ensure that the policy-translation preserves sufficient security semantics, and consequently it is difficult for the IoT devices to soundly enforce access policies (Weakness 1, Section 4.1). Further, the device-side access policy/model is not merely a simplified version of the cloud-side policy, and the device may also maintain access policies that are not intended to be fully shared and synchronized with the cloud, of which the management is often chaotic and vulnerable in reality (Weakness 2, Section 4.2).

4.1 Weakness 1: Semantic Loss in AMT

Although the MaaG IoT device commonly aims to support lighter weight access model than the cloud, it ought to maintain commensurate, sufficient semantics when the complex cloud-side access model is translated to the device-side, a security-critical process that was never adequately defined.

In MaaG IoT, the device owner can manage users (e.g., managing user roles such as *admin*, *guest*; inviting guests and granting them individualized permissions such as locking, unlocking, and inviting additional users). Such modern access management is performed through the IoT companion app (mobile app), and the policies are maintained on the device manufacturer's cloud side.[1] Although the cloud offers functionality-rich access management, in the MaaG architecture (Section 2), it is the IoT device that enforces access policies when a user attempts to operate the device (e.g., to unlock the smart lock) and for this purpose, the device side maintains a simpler access model AM_D. For example, the access model AM_C in the cloud side includes managing the access roles R (e.g., whether it is an admin user or guest user), delegation relations DR (e.g., user A authorized user B as a guest), and permissions P (e.g., lock, unlock, add access code) of IoT users based on their user identity id (e.g., email address, phone number, or account nickname) and supports sophisticated user authentication mechanisms UA (e.g., entering passwords in the mobile app, clicking links in the email, or 2FA [16]):

$$AM_C := (id,\ UA,\ R,\ P,\ DR) \tag{1}$$

In contrast, AM_D (the device side model) might not be maintaining the user IDs or supporting the complicated authentication (likely for simplicity and the lack of resources/hardware such as I/O [48]). To support access control, a user's identity and her permissions from the cloud-side access model is translated to an abstract notion of secret credential cr and a set of attributes $Attr$ related to cr (e.g., including permissions), which are all recorded in the device-side access model:

$$AM_D := (cr,\ Attr) \tag{2}$$

The credential cr is endorsed by the cloud (e.g., based on a signature, see below), so the device is assured that it represents an intended/authorized user. When an intended user wants to operate the device, her IoT mobile app presents cr (obtained from the cloud, see below) to the device, which can make access decisions based on information stored in its access model AM_D.

The key question here is whether AM_D maintains sufficient semantics commensurate with AM_C for making access decisions. In our study, we found that AM_C and AM_D are often extended/customized by individual vendors based on the access control features they offer (e.g., grouped permissions based on locations on SmartThings [61], user roles, etc.). In the absence of a standard, security-assured mechanism to translate AM_C to a corresponding access model AM_D that features light footprint and efficiency — called *access model translation* or *AMT* in this paper — we show that mainstream IoT vendors generally failed to preserve commensurate semantics when translating the access models and policies between the cloud and device. We elaborate on the *AMT* processes of a few vendors, their security weaknesses, and our attacks as follows.

Lost identities in AMT. Our study shows that in the absence of a principled security guideline and approach, real-world manufacturers' efforts to translate the AM_C side user identities, roles and permissions to the device-side counterparts in AM_D were ad-hoc and could easily go wrong.

Figure 2 outlines the AMT process we recovered from Kwikset (i.e., Kwikset Aura Smart Lock [2]) by reverse engineering the Kwikset mobile app and app traffic. The user with the Kwikset app first needs to be authenticated to the lock before operating it. Based on a BLE connection (non-authenticated, based on Just Works [66]), the Kwikset app obtains a random string rs_{lock} from the lock (step 1&2), and sends it to the Kwikset cloud (step 3). Based on the cloud-side policy in AM_C, if the user is authorized (e.g., a guest, tenant, employee authorized by the owner/administrator), the cloud replies with a user credential cr, which is a signature signed on rs_{lock} by the cloud (step 4&5). The lock receives cr (step 6), verifies the signature, and thus is assured that the user is authorized by the cloud. Then the lock trusts the user app and establishes a BLE bonding with the mobile phone following a standard BLE pairing/bonding process, so her phone can connect to and operate the lock in the future without going through steps 1 to 6 again. After these steps, the lock drops cr and relies solely on the BLE bonding to recognize an authenticated user.

There are multiple problems in the AMT process of the Kwikset Aura lock, as found out in our study. The first is the loss of *trusted* user identities in AM_D, which can lead to a number of consequences.

[1] The term "app" in this paper always refers to the IoT vendor's mobile app (sometimes called companion app).

Although the IoT device is only bound with authorized users who have authenticated to the Kwikset cloud and are endorsed by the cloud, the device side AM_D does not maintain the user's identity or the user-related credential cr known to the cloud (as mentioned above). Instead, the lock maintains only a BLE-level binding relation, i.e., ($BLE\_device\_name, BLE\_bonding\_long\_term\_key$) with the phone denoted as r_ble; $BLE\_bonding\_long\_term\_key$ is also maintained on the user's phone so she can connect to the lock in the future without asking the cloud again (for offline access). Note that here the BLE_device_name (e.g., JaneDoeNexus6) is provided by the attacker-controlled Kwikset app, which can be set to any arbitrary value unrelated to the user's identity known to the cloud (id in AM_C, see Equation 1), e.g., the user ID or email address. This means that a benign "owner" will not be able to revoke the access of a malicious user at the device, because there is no mapping stored anywhere between the $BLE\_bonding\_long\_term\_key$ and the original trusted user identity. Indeed, based on the Kwikset user manual [7], when a benign "owner" (or "admin" user) denoted as ow wants to revoke the user's permission (a delegatee user), ow can use the Kwikset app to remove a delegatee user based on her user ID, which will only remove the user from the cloud (AM_C) behind the scene.

To try to clean up AM_D, we find that ow will have to go physically to the lock and use the Kwikset app to remove the user from the lock, although this is still problematic. Behind the scene, the app sends a query message $query\_paired\_smartphones$ to the lock (through the BLE bonding), retrieves all recorded BLE binding relations such as r_ble, and displays device names such as BLE_device_name for the "owner" to select and delete. The owner's selection of device name is sent to the lock, which correspondingly deletes the r_ble, so the target user can no longer connect to (or control) the lock. The problem is that in AM_D, BLE_device_name is untrusted, not reliably related to user identities (or the user credential cr known to the cloud). In practice, a malicious delegatee user (e.g., an Airbnb/hotel guest [52], prior employee) can use a deceptive device name (e.g., name of the owner), so the real owner can easily get confused and fail to locate the delegatee user correctly, or mistakenly believe the delegatee is already removed.

Lost roles, permissions, and lifecycle control in AMT. The problem of the above AMT process did not stop here. Unlike August/Yale locks (see §5.2) that differentiate user roles based on a logical "slot" number recorded in the device, the design of Kwikset's access model AM_D lacks the semantics to identify user roles and their different privilege levels. Specifically, although the Kwikset cloud keeps track of each user's roles and permissions in AM_C (e.g., only "admin" users can invite other users, only authorized users can bind with the lock), in step 6 outlined in Figure 2, when the lock receives a credential cr that assures the legitimacy of the user, there is no companion attribute associated with cr that can describe the user's role or permissions. Interestingly, the "privilege level" attribute seems to be recorded locally by the app. For a low-privileged guest user, the Kwikset app will not display privileged operations in its GUI, e.g., adding/reading offline access code, or removing other users from the lock. Nevertheless, the attribute is apparently missing in the device's AM_D. This means that a malicious low-privileged guest user who may only have temporary access to the lock (e.g., an Airbnb/hotel guest, visitor [79], prior employee) can essentially act as an "owner" and send any privileged commands supported by the lock. These privileged commands can be practically obtained by reserve engineering the Kwikset app.

PoC Attack. For Kwikset Aura Smart Lock, we implemented the following attack where a less-privileged guest user can perform high-privilege (i.e., security-critical) operations. First, we, acting as an invited guest attacker using attacker's Kwikset account (authorized as a guest user of the device) and attacker's own smartphone, will authenticate to the cloud and obtain a corresponding application-layer credential to the guest account. This credential can then be presented to the lock using the non-authenticated BLE connection, in order to establish a BLE bonding (pairing) with the lock. Once the BLE bonding is established, the attacker can send commands directly to the lock from his smartphone. Even though the app GUI will not contain options for performing any privileged operations, e.g., creating/reading access codes, we bypass that by modifying the Kwikset app states/logic using dynamic instrumentation tools (e.g., Frida [62]). Specifically, we crafted BLE messages to the device containing privileged operations, e.g., creating/reading access code by sending BLE messages like `new byte[]{TLV8CommandTxType.CMD_TX_TYPE_SET_ACCESS_CODE, access_code};` where `access_code` is the attacker-controlled access code. This effectively broke the security requirement that only administrative users can create/read access code.

4.2 Weakness 2: Asymmetric and Misplaced Security Responsibilities

In the design of MaaG, the device-side access model AM_D is not merely a simplified version of the cloud-side model AM_C. In certain circumstances, the device needs to maintain access policies that are not intended to be fully shared with the cloud, and thus needs to properly coordinate its security responsibility and control with the cloud. However, our study shows that real vendors' design and practice are often ad-hoc and vulnerable in reality.

For example, a Kwikset lock owner or authorized, invited users can add offline access code to Kwikset locks (used for unlocking without using the app, see Section 2). Specifically, the authorized user whose Kwikset app has bound with the Kwikset lock (see Section 4.1) can simply use the app to designate an access code, and the app will encode the access code into a device-specific command and send the command to the lock to add the access code. Behind the scene, the AM_D recorded the offline access code denoted as ac. While the access code is security/safety-critical, we found that ac is not designed/intended to be shared with the Kwikset cloud, which does not record the lock's access code in AM_C. Indeed, by design, the Kwikset cloud does not maintain any credential that can be directly used to unlock/lock the Kwikset device — AM_C only keeps track of who the authorized users are and helps them bind with the device when they need to as outlined in Figure 2. Such a design does help mitigate some security risks so the Kwikset cloud does not become the single point of security failure: even if the cloud is compromised and leaked credentials it stored, attackers would not otherwise get the access codes to control all Kwikset users' devices.

Despite the security benefit from this design, serious problems arise: while a typical IoT owner may rely on the vendor's app (which communicates with the cloud) to remotely manage the device and fully inspect the accessibility states of the device (e.g., all current authorized users, locked/unlocked status), the Kwikset cloud could not show there is access code added on the device, even after a malicious delegatee user adds one during her stay (e.g., in Airbnb or vacation rental). Even after the owner revokes the delegatee user from the app — an operation that actually removes the user from AM_C, we found that the access code is still recorded in AM_D and effectively allows the malicious user to still unlock the smart lock after his permission is revoked from the cloud's/app's perspective (see our PoC attack below). In our experiment, we found that the owner has to go physically near the lock, so her app could query the full status from AM_D in the lock: the app has a dedicated UI to show access codes already added to the lock and allows the owner to add/remove the codes. Such a design has two practical problems in modern IoT usage. First, modern Airbnb hosts and hospitality services often only remotely manage the access for guests and leverage the vendor's app to manage (add/remove) users and grant/revoke access without the intention/assumption to physically go to the house. Second, from the owner's perspective, the full, secure management of the IoT device is split into two complementary parts without being made clear by the vendor: the remote management of AM_C and the local management of AM_D (e.g., the management of access code has to be done locally), and the oversight of managing any part, which is highly possible in reality, can leave the device under an insecure, unintended state, with serious security and safety implications.

PoC attacks. We implemented the following attack. We, acting as an authorized, invited user of Kwikset Aura, first have to authenticate to the cloud and obtain a valid credential and establish a valid BLE bonding with the lock (just like the previous PoC). Then we add a malicious access code using the attacker's smartphone directly to the lock via the BLE message mentioned in the previous PoC. As discussed, such access codes are not synchronized to the cloud. We verified that even after the owner revoked our access permission/role from AM_C, and later physically removes our BLE bonding from the lock, we can still use the previously installed access code to unlock the lock. This is because the offline access codes are entered through physical keypads (which are separate from the BLE bonding). A benign owner will need to query the list of access codes through BLE and then remove them.

5 RISKS IN POLICY SYNCHRONIZATION

In modern IoT usage scenarios (e.g., remote Airbnb/hospitality management), the cloud and device in MaaG IoT have to rely on guest/untrusted users' phones as gateways to communicate and synchronize access policies. That is, MaaG IoT essentially is featured with network partition and weak consistency. Hence, another key challenge in MaaG IoT is the design of proper mechanism to synchronize access policies between the cloud and device to reach a sufficient level of consistency. The "sufficiency" and corresponding security assumptions are yet to be well understood. Notably, even though the classic "eventual consistency" model [69] has been proposed as a potential solution in similar contexts [36], it is not clear how to properly achieve it in the context of MaaG IoT. As we show in this section, real-world vendors and stakeholders failed to fully understand and come up with a sufficient consistency model between the MaaG cloud and device, leaving significant opportunities for new attacks.

5.1 Weakness 3: Inadequately Defined Causal Consistency in Access Policy Synchronization

In MaaG IoT, when there is a policy update on the cloud (e.g., the owner/administrator uses the mobile app to grant/revoke permissions for a user), the cloud needs to synchronize the policy update to the device, so that the device can enforce the up-to-date policies when a user attempts to operate the device.

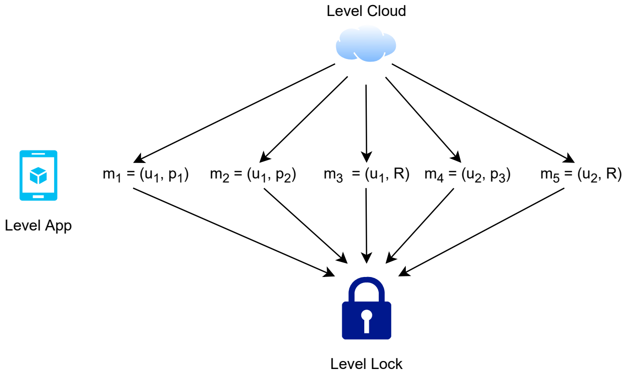

Figure 3: Level Lock's Policy Synchronization Messages

Take the Level lock as an example (Figure 3). The on-device policies in AM_D maintain a set of <user, permission> records and the cloud sends policy-sync messages to the device as "updates" to the device, e.g., to report the fact that a new user is created. For each Level user account, a public/private key pair is recorded in the Level app. As an example, a device owner may grant a "guest" permission, p_1, to a user u_1 (aiming that the user can lock/unlock the lock but cannot configure the lock or invite other users), and correspondingly the Level cloud sends the policy update as a message, m_1, to the device: m_1, signed by the cloud, includes the public key of u_1 and the permission p_1. Note that in the design of Level and other eventual consistency models [26], the app of any authorized user, if physically near the device, serves as the Internet gateway to relay the messages. After m_1 is received by the lock, the lock records the public key of u_1. Thus, when the user u_1 uses his Level app to operate the lock, the app signs the commands with his private key, which can then be recognized by the lock. Additionally, the owner may grant the same user u_1 another permission p_2 (e.g., for lock configuration such as adding access code, see policy-sync message m_2 in Figure 3), and then revoke all his permissions by removing the user u_1 (see policy-sync message m_3). Additionally, the owner might bestow another user u_2 access to the device and revoke his permission later (see m_4, m_5).

In real-world scenarios, policy-sync messages (e.g., m_1, m_2) to arrive at the device may be disordered. For example, if m_2 is received after m_3 (e.g., due to unintended network partitions/interruptions or intentionally manipulated reordering, see our PoC attack below) — with the order denoted as (m_1, m_3, m_2) — the eventual policy state in the device will include (u_1, p_2), which violates the security expectation. Traditional distributed systems often leverage temporal-order causal relation between messages to handle the nondeterministic order of message arrivals to ensure that an older version of a data object, if received later than a newer version of the data, will not overwrite the newer data [46]. Using a state-of-the-art causal relation model based on vector clock [31, 49, 53] (adopted by AWS DynamoDB, an industry-leading distributed database [26]), for example, the messages are labeled with temporal information by the sender and messages such as m_1, if received later than m_2, should be dropped by the receiver (m_1 is created earlier than m_2 and the recipient should not replace m_2 with m_1).[2]

The proper synchronization of policies between the cloud and MaaG IoT devices bears greater logical complexity and cannot directly adopt the prior, industry-testified temporal-order causal consistency models. Following the previous example, despite the temporal order (m_1, m_2, m_3) with which the messages are created, both m_1 and m_2 are important to keep regardless of their orders to arrive (m_1 received later than m_2 should also be kept). Hence, it appears that the causal relation between policy-sync messages in MaaG must bear a "loose" causal relation such that regardless of the order to receive, for example, m_1 and m_2, both messages should be processed to update the access policy in the device. However, the order to receive m_3 relative to m_1 and m_2 is security-critical (see above). This is because m_3 is causally related to both m_1 and m_2, although m_1 and m_2 are logically independent with each other. Further, the relative order between the two groups of messages m_1, m_2, m_3 and m_4, m_5 may not be security sensitive (they are concerning security policies of two separate users); however, if the cloud issues a message m_6 to remove all users (or users with a particular role concerning both u_1 and u_2), the order to process m_6 (on the device) relative to the five messages (m_1 to m_5) is security-critical (e.g., processing m_6 then m_4 will leave the user u_2 on the device). Hence, the design here needs to clearly define the relative logical relations between multiple policy-sync messages, and the prior temporal causal models are insufficient for the security of MaaG IoT access policy synchronization.

In the absence of an in-depth security analysis and properly designed mechanism for MaaG policy-sync, our study indicates that real-world MaaG vendors and systems failed to appreciate the essential logical relations between the policy-sync messages, leaving opportunities for practical attacks. Further complicating a proper design of the MaaG policy sync is goals to ensure the cloud's awareness of the latest device-side policy states, since an IoT user would naturally rely on the cloud to understand/manage status of her device (e.g., using the mobile app as a control console, see Section 1). In the case of Level (Figure 3), when the lock receives a policy-sync message (e.g., m_1, m_2), it replies to the cloud (through the mobile as the Internet-gateway) a response message indicating that the particular message (e.g., m_1, m_2) has been received and processed on the device. Again, such a response message, aiming that the cloud is notified about the policy status on the device, cannot reliably reach the cloud based on the order they are generated due to the network partition nature in MaaG. For example, a response message to m_2 might arrive at the cloud later than the response message to m_3 (because the device fundamentally lacks a reliable Internet connection in MaaG IoT), even if m_2 and m_3 arrive at the device in the right temporal order. In such a situation, it is non-trivial for the cloud to know the order of m_2 and m_3 processed at the device and thus the real policy states on the lock. We find that, for the Level lock, a malicious user (e.g., an authorized employee, tenant, guest) could manipulate the order of m_2 and m_3, or the response messages of m_2 and m_3, such that the policy state on the device will be different from the one on the cloud.

PoC attacks. Due to inadequately defined causal consistency, an attacker, acting as a malicious invited guest, can reorder or retransmit messages when forwarding them from the cloud to the device using the attacker's smartphone. This can lead to various unexpected states at the Level lock — in this case, the attacker can retain its access even after it is revoked by the owner. We first did a trivial experiment as follows: when the cloud issues an initial remote invitation message that grants the attacker access, denoted as m_{addA} and later a revocation message m_{revA} removing the attacker's access, an attacker can simply reorder the two messages if both are forwarded through the attacker's smartphone. It would then cause the m_{revA} to apply first, which effectively does nothing at the lock, and then the m_{addA} message will allow the attacker to gain access subsequently. However, a more likely scenario is that m_{addA} and m_{revA} are sent some time apart, e.g., a guest stays for a few days in a rental property and then leave. In such a scenario, we show that an attacker can first apply the initial m_{addA} and still retain access after m_{revA}. What the attacker has to do is to save a copy of the original m_{addA} while forwarding it to the lock. Later on, the attacker simply re-sends it after m_{revA} is applied (possibly by another benign user), re-allowing the attacker access. It's worth mentioning that these messages do have timestamps and the lock can see that the re-sent message m_{addA} is an older one compared to m_{revA}. Nevertheless, because of the unreliable nature of message delivery in MaaG IoT, the device cannot expect to see messages arrive in order and would therefore still accept an older message, enabling the attack.

Discussion. Notably, a quite related work [36] proposed adopting eventual consistency model that may apply to MaaG scenarios when the cloud needs to synchronize policies to the device. However, the approach, based on a few key assumptions, is difficult to work in modern MaaG architecture, and has not been adopted in any of the devices we studied. Above all, it assumed completely equivalent access models between the cloud and device. In Section 3, we show that real-world MaaG cloud needs to maintain more complicated access models than the device; for example, the device may not be able to authenticate the user or directly maintain the user ID, and in reality needs an AMT process, which is critical but was not considered in [36]. Second, [36] is based on relatively simple access

[2] In a distributed system, multiple storage/computing nodes maintain the same data objects (as replicas for high availability) and when any one node has/receives an updated version of the data, it tries to send out the data to other nodes, aiming that all nodes eventually have consistent, latest version of the data (also known as eventual consistency [69]). To this end, in the prior models, a sender node should label temporal-order version information (called *version-clock*) along with the data (e.g., based on vector clock [31, 49, 53] adopted by AWS DynamoDB, an industry-leading distributed database [26]), and the version-clock can indicate causal relation between multiple copies of a data object. For example, a sender node $node\_1$ sends out multiple versions of a data object obj it received, with each version accompanied with the version-clock $(node\_1, version\_1)$, $(node\_1, version\_2)$, etc. Although the multiple versions of the data object may not arrive at other nodes strictly following the temporal order (e.g., due to network partition or failure), a node that receives the data with temporally newer version-clock can disregard a later received copy with temporally older version-clock.

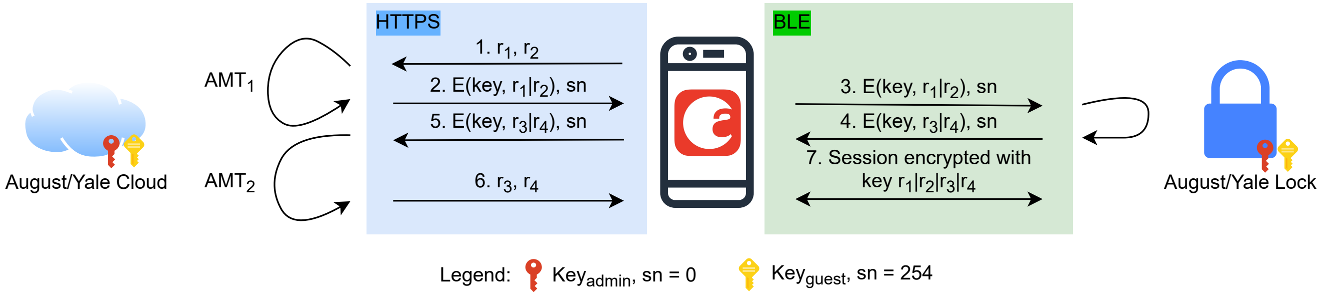
August/Yale Cloud
AMT$_1$
AMT$_2$
HTTPS
1. r$_1$, r$_2$
2. E(key, r$_1$|r$_2$), sn
5. E(key, r$_3$|r$_4$), sn
6. r$_3$, r$_4$
BLE
3. E(key, r$_1$|r$_2$), sn
4. E(key, r$_3$|r$_4$), sn
7. Session encrypted with key r$_1$|r$_2$|r$_3$|r$_4$
August/Yale Lock
Legend: Key$_{admin}$, sn = 0 Key$_{guest}$, sn = 254

Figure 4: August Online Authentication Protocol

models (access control list) compared to those in real devices and assumed that when there is a policy update on the cloud, the cloud sends its entire policy to the device, which can be cumbersome for IoT with limited resources/power and is not the case in real MaaG IoT systems we observed.

5.2 Weakness 4: Lack of Conflict Identification in Access Policy Synchronization

Our study finds that in the absence of a reliable, standard mechanism to synchronize access policies, there are a variety of possibilities in the design space of MaaG IoT that can be leveraged by attackers to cause conflicts of access policies between the cloud and devices. When the conflict is indeed introduced, there lacked the proper understanding and techniques to identify the conflicts, not to mention adequately reconcile them without serious impacts on availability and usability.

The August Smart Lock [1] and the Yale Smart Lock [11], among the most popular in the U.S. market of smart locks, come with a relatively complicated interaction protocol between the cloud and locks leveraging the mobile phone as the gateway (based on their US Patent No. 2016/0189454A1 [42] and a third-party analysis [41]). The protocol offers multiple advanced capabilities. In particular, two offline access features are provided: (1) in-app offline access (even if the cloud is temporarily down or the app loses Internet access, authorized admin users' app can still operate the lock), and (2) offline access code (used for unlocking without the app, see Section 2). We find that the sophisticated access control features in modern MaaG IoT can easily go wrong and lead to intractable access-policy inconsistencies between the cloud and devices.

Figure 4 outlines the patented access management protocol involving the August/Yale cloud service, mobile app, and lock. Each lock comes with two built-in, persistent secrets pre-shared with the cloud under the factory setting, namely key_{admin} and key_{guest} (recorded at two logical slots, slot 0 and slot 254 respectively in AM_D). At a high level, a legitimate user can fetch a token tn encrypted by the cloud with the key corresponding to her role (admin or guest), so the device receiving the token can determine her legitimacy and role based on the key needed to decrypt the token. More specifically, the app sends two random numbers r_1 and r_2 (step 1); the cloud service encrypts them to obtain token tn and sends tn to the lock (step 2 and 3); the lock decrypts tn to obtain r_1, r_2, knows the user's role based on the decryption key needed, and encrypts two new random numbers r_3 and r_4 using the same key, and sends the result rt to the cloud (step 4 and 5); the cloud decrypts rt to obtain r_3 and r_4 and release them to the app (step 6). From this point of time, the app and lock both know the four random numbers and concatenate them to form a 128-bit value $r_1|r_2|r_3|r_4$ as a session key. The app can then send commands corresponding to the user's role (e.g., locking, unlocking, configuring) encrypted with the session key.

Exploiting offline access features. Since a session can expire soon (after tens of seconds), to support sophisticated fault tolerance (e.g., the app works even when the cloud is temporarily down [12], see above), the August/Yale lock supports two offline-access features. We found that supporting such features unwittingly bestows the mobile gateway excessive trust. Exploiting the trust, we identified at least two practical opportunities for a malicious user to manipulate the interaction process and introduce inconsistent policy states between the cloud and device. First, to enable the offline app (either the app lost Internet access or the cloud is down) to access the device, based on an established, valid session, the app can add a new key, such as $key_{offline}$, to the lock (by calling a lock API), recorded at a logical slot in the device AM_D (with a slot number between 1 to 253, in parallel to the two built-in keys at slot 0 and 254 discussed above). When the app lost connection to the cloud (and thus cannot establish a fresh session), $key_{offline}$, known to both the app and lock, is used to derive a new session key between the app and lock similar to the process in Figure 4. A fundamental problem here is that, in modern MaaG IoT contexts, the mobile gateway is not always honest (e.g., a guest or more generally delegatee user) and after the $key_{offline}$ is added to the lock, there lacks a reliable protocol for the cloud to fully monitor the addition, existence, or revocation of the $key_{offline}$. As a consequence, when the August/Yale device owner removes the authorized user from the cloud, she could not track any $key_{offline}$ left by the user or identify any inconsistent policy states between the cloud and the lock.

Second, the August/Yale locks support offline access code whose states are not reliably synchronized between the cloud and device. Specifically, based on an established, valid admin session, the user app can add an offline access code to the lock. The user app is supposed to synchronize such a policy to the cloud (by calling the cloud API to record the same offline code to the cloud), but it does not have to do so possibly because August/Yale favors the offline support or fault tolerance. Consequently, even if the owner removes the invited user from the cloud-side policy, the user retains access to the lock with the offline access code.

PoC attacks. We, acting as an attacker with `owner` (August's/Yale's terminology equivalent to admin) privilege, could inject a malicious offline key into the August and Yale Smart Lock without having the offline key recognized / recorded at the cloud. Specifically, we first allow the attacker's app to complete its handshake with the lock (see Figure 4), so that the app is authenticated to the lock as `owner` and can add new offline keys to the lock. By design, when a benign user attempts to add an offline key via app GUI, the app will check whether the offline key has been successfully added to the lock via BLE, and subsequently inform the cloud about the success or failure. However, an attacker can simply change the app's behavior (again using dynamic instrumentation as mentioned in other PoCs) and omit the last step of informing the cloud. This means that the cloud will never be aware of the fact the offline key is successfully installed on the lock. Even after another honest `owner` user revoked the attacker's cloud-side access and then physically synchronized the lock with the cloud, the attacker could still retain access to the lock (e.g., to lock/unlock the smart lock) using the "hidden" offline key. This is because the lock itself does not have a way to report its current policy state to the cloud.

6 SUMMARY AND DISCUSSION OF FLAWS AND ATTACKS IN ALL MAAG IOT DEVICES

For the sake of clarity, we do not go into the details of all ten MaaG IoT devices which we analyzed. However, we summarize all the results in Table 2, which shows that not a single MaaG IoT device is free of the vulnerabilities we discovered. As we can see, weakness 1 is the most common flaw that is observed in five smart locks and two item trackers, showing the generality of the weaknesses that we have found. For example, Ultraloq, Honeywell, Schlage, Geonfino, Tile, Chipolo all implement access sharing/revocation by sharing static keys that never change/rotate to untrusted users/invitees. Once the keys are known to the untrusted users/invitees, they can always dynamically instrument the companion apps to control the IoT devices. We have developed PoCs for all of these devices, showcasing that an attacker can retain unfettered access with the static keys, even after their accesses being "revoked". Since these attacks are straightforward to understand, we omit the details. Vulnerabilities of this style is categorized as weakness 1 (semantic loss in the AMT process) because the static key (after the AMT) does not retain any of the original information in AM_C, e.g., user id, permissions, access time. Interestingly, even though some smart locks share the same weakness (e.g., weakness 1), they can lead to different attack consequences. This is because the underlying flaws sharing the weakness may differ. The next most common weakness is weakness 4, affecting three smart locks.

Generality of the flaws. We have showcased the flaws in 8 smart lock devices and 2 other IoT devices. We believe the flaws we identify are general across an even a wider range of MaaG IoT devices, as long as they have the notion of access sharing. We see that it is the de facto standard that the IoT cloud will maintain a primary copy of the access control policy to facilitate remote management. On the other hand, the IoT device itself needs to be able to enforce the policy in some way, and it needs to do so independent of the cloud due to the offline access requirement. This implies that they will need some version of the policy from the cloud. As such, access model translation and synchronization are natural concerns for these MaaG IoT devices. In addition, there is a trend of IoT devices becoming more and more multi-user friendly. For example, we have seen a number of recent studies (described in related work) covering a variety of IoT devices that allow device sharing across users [36, 39, 50].

Table 2: Summary of Measurement Results

MaaG IoT device	Weakness	Consequence	Google Play Installation
Level [8]	3	(a)	10k+
August [1]	4	(a)	1,000k+
Yale [11]	4	(a)	100k+
Ultraloq [10]	1,4	(a)	100k+
Kwikset Aura [2]	1,2	(a),(c)	100k+
Honeywell [6]	1	(a),(b)	1,000k+
Schlage [9]	1	(a)	100k+
Geonfino [5]	1	(a),(b)	100k+
Tile [4]	1	(a),(b)	5M+
Chipolo [3]	1	(a),(b)	500k+

(a) allowing a temporary user retaining permanent access to the MaaG IoT device;
(b) allowing a temporary user to share the access to other unauthorized users;
(c) allowing a temporary user to escalate her privilege.

7 MITIGATING VULNERABILITIES IN MAAG IOT ACCESS CONTROL

In this section, we first present the security goals of a secure MaaG IoT access control system. Then, we design a novel protocol that satisfies all the security goals and defends against all identified MaaG IoT attacks. After that, we perform formal security analyses that can provide security guarantees for the proposed protocol. Finally, we implement the protocol end-to-end to show that it is practical in the real world.

7.1 Security Goals

The security goals of our design are the following: (1) only currently and explicitly authorized users according to the access policy on the cloud can access the IoT device, subject to a bounded delay (configurable) that allows users to retain access for some limited time, (2) permanent users (e.g., owners) can enjoy "offline availability" [50] with an extended period of time (configurable).

7.2 Secure Protocol Design

High-level Design. A key principle in our design is that we *force the user app to help synchronize the cloud-side policy to the IoT device while authenticating to the IoT device.* To implement this, we require some form of "credentials" the app has to acquire through the cloud when interacting with the IoT device. During the process of acquiring the credentials, the app has to also relay the most recent device-side access policy to the cloud service for it to perform any appropriate access model translation and achieve cloud-to-device synchronization. The credentials have limited lifetime of validity which is configurable depending on the privilege level and the desired trade-off between security and usability. The longer the lifetime, the more convenient and offline accessibility it provides. On the other hand, longer lifetime also means that an attacker can potentially retain longer access (if the IoT device does not get synchronized with the cloud through other channels, e.g., a benign user app). Below we detail the design of our Secure Access Policy Synchronization (*SAPS*) Protocol.

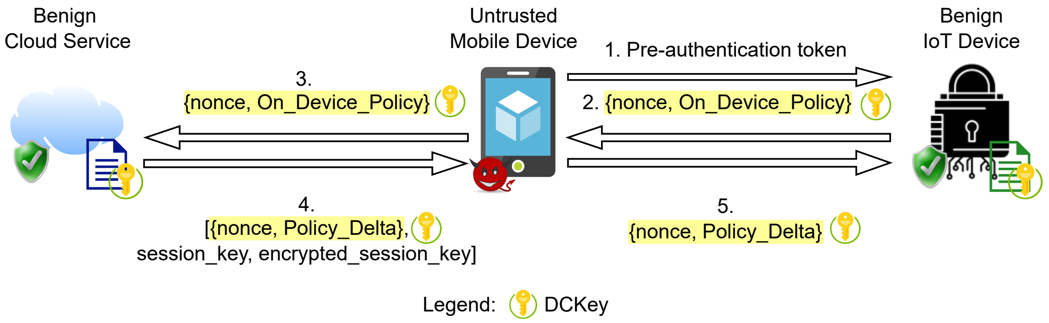
Benign
Cloud Service
Untrusted
Mobile Device
Benign
IoT Device
1. Pre-authentication token
2. {nonce, On_Device_Policy}
3. {nonce, On_Device_Policy}
4. [{nonce, Policy_Delta}, session_key, encrypted_session_key]
5. {nonce, Policy_Delta}
Legend: DCKey

Figure 5: Secure Access Policy Synchronization (SAPS) Protocol

Design Regarding Access Model Translation. As mentioned in §4.1, we allow the cloud service to deploy expressive access control and dynamically translate access models for MaaG IoT devices, which enable lighter weight access control. For example, the cloud can combine role-based access control with proper permission cascading (i.e., revoking permissions automatically and cascadingly). We require the MaaG IoT devices to use at a minimum the credentials associated with attributes to authenticate user apps. The attributes should contain allowed operations (e.g., to lock/unlock, to add a new access code, to factory-reset). Additionally, we require the attributes to contain a configurable timeout, determining the lifetime of credentials as mentioned above. The timeout can be set by the cloud depending on the role of each user (an owner can have an unlimited lifetime). The device does not have to be aware of the various user roles and only needs to enforce the timeout instead.

In addition, our design specifies that the credentials endorsed by the cloud should be tightly coupled (generated along) with access model translation. Finally, our design requires that the credentials be cross-cutting across all parties — cloud, app, and device. This means that the credentials must be delivered from the cloud, through the app (which stores the credentials), and then to the device. This effectively can prevent the semantic loss problem in access model translation (described in §4.1).

Design Regarding Access Policy Synchronization. As mentioned earlier, the key intuition is that we force apps (benign or malicious) to participate in the synchronization of access policies. In particular, given that a malicious user tends to have less privileges, unlike the owner, their credentials will timeout and be forced to participate in the synchronization protocol. With the correctly designed synchronization protocol (to be detailed next), we can then ensure the freshness of the policy without relying exclusively on benign users (as is the case in the design of August/Yale locks, etc.). This means that if the malicious user's access is revoked at the cloud, it can retain the (offline) access only for a limited time window. We will reason about the security property of our protocol in §7.3 and show how it can defend all attacks described previously.

Details of the SAPS Protocol. We assume the IoT device and cloud service pre-share (1) a symmetric encryption key (Device-Cloud Key, or DCKey in short) and a symmetric encryption/decryption algorithm (this is fairly standard and adopted by current smart lock vendors already), (2) a message authentication code (MAC) algorithm, (3) an app-to-device session key encryption key (SKEK). Note that DCKey and SKEK are never exposed to other entities (never transmitted over the network and not observable by an untrusted app being a MitM) and are the trust anchors our design relies on. The usages of these keys are described later.

The cloud service dynamically translates access models and releases endorsed, fresh access policy (i.e., in the form of (*cr, Attr*) as mentioned in §4.1) and user-specific credentials to user apps. The fresh access policy to be applied to the device is cryptographically protected so that the app cannot tamper with it.

Authorized user apps have to follow the secure access policy synchronization protocol in Figure 5 to access the device. Authorized user apps can obtain the physical (MAC) address of the IoT device from the cloud service beforehand to discover the device. We require the app to initially authenticate to the IoT device through an unauthenticated connection (e.g., BLE Just Works local link). Specifically, we require the smartphone to obtain a pre-authentication token from the cloud to initiate the authentication process with the IoT device. This is necessary because otherwise any user physically adjacent to the device can initiate the entire protocol, which can be quite expensive, and lead to DoS attacks. We reuse the pre-authentication token generation method described in [60], which is based on monotonically increasing counters. The device, seeing a valid pre-authentication token, starts the policy synchronization. It generates a nonce to guarantee freshness [45, 60] and concatenates it with all the on-device access-policy identifiers (i.e., `On_Device_Policy`). Then, the nonce and the `On_Device_Policy` are encrypted with DCKey, and a message authentication code (MAC) is calculated for the encrypted payload to guarantee message integrity. Finally, the payload and the MAC are transmitted to the user app (step 2 in Figure 5).

Then, the app accepts the encrypted payload and directly forwards it to the cloud service to obtain the fresh access policy and the access credentials `session_key` and `encrypted_session_key`. When receiving the encrypted payload, the cloud performs AMT. It first checks if the user is an authorized user of the device. If that is the case, the cloud generates a random `session_key` for the upcoming app-to-device encrypted connection, and encrypts {`session_key`, `user_ID`} using the session key encryption key (SKEK), outputting `encrypted_session_key`. If the user is a `guest` user, the cloud assigns `guest` permissions and a very short session key validity, such as 30 seconds, to `session_key`. If the user is an `owner` user, the cloud assigns `owner` permissions and a configurable session key validity set by the IoT device manufacturer. After performing AMT for the current user, the cloud service also performs AMT for other users of the same device, generating a fresh access-policy to be synchronized to the device. Then, the cloud service verifies the MAC and decrypts the encrypted payload from the app. If the MAC is correct, the cloud calculates the delta between the fresh access-policy and `On_Device_Policy`, generating `Policy_Delta`. The cloud does not modify the nonce because the nonce is a primitive for the IoT device to guarantee access-policy freshness [45, 60]. The nonce's space should be large enough, e.g., we used 64 bits in our implementation. The cloud then encrypts the payload {nonce, `Policy_Delta`} with DCKey, and calculates a MAC for the encrypted payload. The cloud sends back the encrypted payload to the app along with `encrypted_session_key`, `session_key`, and other attributes (such as the session key validity) through secure tunnels (in our implementation HTTPS).

The app then knows the `session_key` is valid within a time window, and forwards the encrypted, fresh access policy to the device. The device verifies the MAC, decrypts the payload, and applies `Policy_Delta` only when (1) the nonce in the returned payload matches the nonce originally generated by the device itself, and (2) the MAC is correct.

We borrow a part of Needham–Schroeder protocol [55] to let the app initialize an encrypted session with the IoT device in order to authenticate to the IoT device. The app first sends its `encrypted_session_key` to the device. If the device finds the enclosed `user_ID` in its on-device access-policy, the device encrypts a nonce `Nonce_D` with the enclosed `session_key` and sends it to the app. If the app can return `Nonce_D - 1` encrypted with `session_key`, the device grants the mobile phone's physical address the permissions of `user_ID`. The app can encrypt device commands using `session_key` to operate the device when that user's session key is still valid (i.e., within the session key's validity time window).

For simplicity, we allow only one ongoing session at a time. For each session, we set a timeout for protocol to complete (where we consider nonce to be fresh). If the session is not completed within the timeout, the protocol will abort. This means that the if an attacker drops messages or do not participate fully in the protocol, it will simply cause the session to abort and no policy update will occur. If the attack tampers with any messages, it will cause the MAC check to fail and the protocol will abort the session immediately.

Definitions and properties. Directly based on the design of the protocol, we define the following properties held by *SAPS*.

(1) Property of "nonce expiration" (NE): T_n is a time window (e.g., 20 seconds in our implementation) for the nonce (step 2) to expire. Based on the protocol definition, once the nonce expires, the device will not respond to step 5. T_n is configured by the system.

(2) Property of "one-time use nonce" (OUN): For any round rd_x of protocol execution with nonce n_x, once the step 5 is successfully finished (the device has accepted the message with n_x in step 5), rd_x is finished and the device drops n_x.

(3) Property of "device-side policy expiration" (DPE): T_p is a time window specified in policy updates regarding records of temporary users, it is a minimum of (1) the pre-configured value (30 seconds in our implementation) and (2) the absolute timeout in the record on the cloud-side policy P_c. T_p effectively determines when the device-side policy P_d will expire (we assume all records in P_d share the same T_p and expire together for ease of discussion), and it most likely is determined by the pre-configured value as it is typically much smaller than the longer-term timeout on P_c. Based on the protocol definition, P_d, if not empty, expires after T_p if P_d is not updated until expiration.

(4) Property of "At-most One Round of Protocol Execution at a Time" (AOR): The execution of step 1 - 5 sequentially is called one round of protocol execution. After step 1 is accepted by the device, one round starts and the device generates a nonce and starts to count down for the nonce to expire. Based on the protocol definition, at any time t, once one round has started, if the round has not successfully finished and its nonce has not expired, the device does not start another round of protocol execution (i.e. the device does not accept step 1 in another round).

7.3 Security Analysis

In this section, we elaborate the formal security guarantees of the proposed defense (the *SAPS* protocol) based on generalized theorems and corresponding formal proofs.

Assumption. According to our threat model, the device and cloud are honest and always respond to requests based on the protocol (step 1, step 3, step 5 being requests made by the phone/user; step 2 and step 4 being responses to their prior steps); the cloud always responds using the latest policy it has; the user/phone can be malicious and might not follow the protocol.

Def. 1: IoT operations O_d. We consider all IoT operations (e.g., to switch on/off, to reset) supported by the device as a finite set O_d. Intuitively, in contrast, there can be operations managed by the cloud, such as permission delegation, that may not be supported by the IoT device.

Def. 2: No semantic loss. We consider the device-side access policy P_d has no semantic loss from the cloud-side access policy P_c, denoted as $P_d \psi P_c$ (also called P_d is a no-loss translation from P_c), if both of the following conditions hold:

(1) For any policy record r as a tuple $<uid, Permissions, timeout, \dots>$, $r \in P_c$, r is uniquely mapped to a policy record r', $r' \in P_d$.

(2) For any record r' as a tuple $<cr, Attr(Permissions', timeout', \dots)>$, $r' \in P_d$, for each $p \in Permissions$ in the corresponding policy record in P_c, if p relates to any operation supported on the IoT device ($p \in O_d$), we must have $p \in Permissions'$. Also, $timeout'$ must be T_p as assigned by the system (see definition in §7.2). Note that $Permissions'$ is a subset of $Permissions$ in r because there may be permissions related to operations that are supported on the cloud only (beyond the ones in O_d).

Def. 3: Fresh policy. A *cloud-side* policy P_c is fresh at time t if P_c is the latest policy of the cloud at time t.

Def. 4: Expired policy. A *device-side* policy P_d is expired if the corresponding timeout T_p is triggered. This means that the policy is no longer effective on-device.

THEOREM 1. *Considering any state or time t with the cloud c, the device d, and an arbitrary user u in the protocol (Fig. 5), the device-side policy P_d is in one of the three states (1) P_d is empty; (2) P_d is expired; (3) P_d is a no-loss translation of the cloud-side policy P_c, where P_c is fresh at a time t' earlier than t where the time difference is bounded by $t - t' \leq T_p + T_n$ (see definition of T_p and T_n in the previous section).*

Intuitively, Theorem 1 says, at any time t, the device-side policy P_d is either an empty/expired policy (by default denying any access) or a no-loss translation of the cloud-side policy P_c, and P_c is fresh at least at a recent time point t'. We consider all three cases acceptable with regards to our security guarantee because the worst that can happen is that a malicious user has a prolonged (bounded by T_p) access due to case (3), which matches our goal outlined in §7.1.

Proof. Case 1: P_d is empty at time t, if no phone/user has executed the protocol successfully until t.

Case 2: If no phone/user successfully executes (finishes) the protocol between time $(t - T_p, t)$, based on Lemma 2, P_d is expired or empty at time t.

Case 3: If any phone/user successfully executes (finishes) the protocol between time $(t - T_p, t)$, based on Lemma 1 and Lemma 2 (see below), P_d is a no loss translation of the cloud policy P_c.

In Case 3, let the round of protocol execution that yields P_d on the device (from P_c on the cloud) be rd. Because rd finishes as early as $t - T_p$, rd starts and executes as early as after $t - T_p - T_n$ (otherwise, the nonce of rd expires and rd cannot finish). Hence, P_c (used in step 4 of rd) is fresh at time t', and the time difference between t and t' is bounded: $t - t' <= T_p + T_n$. □

Lemma 1. *Consider any state with the cloud c, the device d and an arbitrary user u in the protocol (Fig. 5), immediately after u successfully carries out any one round of the protocol (step 1 to 5), the policy of the device $P_d \not\psi P_c$, where P_c is the cloud policy used in this round (i.e., P_c is used to generate $Policy\_Delta$ in step 4).*

Proof. We consider all operations (e.g., to switch on/off, to reset) supported by the device as a finite set O_d without loss of generality.

Let the round of protocol execution be rd and rd starts at time t without loss of generality. Let the time when rd successfully finishes be t'. Once rd starts, based on the property of AOR, rd is not interrupted by any other round of protocol execution until t'.

At step 4 of rd, the cloud policy P_c is used to generate a $Policy\_Delta$. P_c is based on the access model AM_C and each record r, $r \in P_c$, denoted as $(uid, Permissions, expiring\_timestamp, \dots)$, includes (1) a unique user identity (uid), (2) all permissions of the user denoted as a set $Permissions$, (3) an $expiring\_timestamp$ value for $Permissions$.

Case 1: If the device-side policy is empty or expired at time t, indicating that On_Device_Policy in step 2 is empty, $Policy\_Delta$ is a set, and for each $r, r \in P_c$, there is a unique $r' \in Policy\_Delta$ denoted as $(uid, Attr(Permissions', relative\_timeout', \dots))$, where $Permissions' \subset Permissions$, and for each $p \in Permissions$, if $p \in O_d$, $p \in Permissions'$. $relative\_timeout'$ is T_p or set by the cloud based on $expiring\_timestamp$. After step 5 (at time t'), P_d is a copy of $Policy\_Delta$. P_d is a no-loss translation of P_c (at time t').

Case 2: if the device-side policy is neither empty nor expired at time t, indicating that On_Device_Policy in step 2 is not empty, On_Device_Policy will be transmitted to the cloud representing the on-device policy at time t. Let P_x be a no-loss translation of P_c. In step 4 of rd, $Policy\_Delta = P_x - On\_Device\_Policy$. After step 5 of rd, $P_d = On\_Device\_Policy + Policy\_Delta = P_x$.

With both Case 1 and Case 2, the lemma is proved. □

Lemma 2. *Consider any state or time t with the cloud c, the device d (with policy P_d), and an arbitrary user u in the protocol (Fig. 5). If one round of protocol execution starts (at time t) and fails to finish until the nonce expires (by time $t + T_n$), at time $t + T_n$, the device-side policy P_d is not changed (except to expire).*

Proof. Let the round that starts at time t be rd. Based on the property of AOR, once rd starts, the device blocks any other rounds unless rd successfully finishes or it is after time $t + T_n$. No round of protocol execution successfully finishes by time $t + T_n$. Hence, the device-side policy is not changed between time $(t, t + T_n)$ except that it can expire. □

Theorem 2. *For arbitrary two $Policy\_Delta$ messages (step 5 in the protocol) m_1 and m_2 accepted by the device, if m_2 is accepted after m_1, then m_2 is generated by the cloud after m_1 (m_2 is a newer policy than m_1 based on the cloud-side policy).*

Proof. Let the round of protocol execution that includes m_1 be rd_1, with nonce n_1. Let the round of protocol execution that includes m_2 be rd_2, with nonce n_2. Because the device accepted both m_1 and m_2, based on the property of OUN, $n_1 \neq n_2$. Hence, rd_1 is not rd_2.

Let the time when m_1 is accepted be t_{m_1}. Then rd_1 finishes at t_{m_1}. Let the time when m_2 is accepted be t_{m_2}. Then rd_2 finishes at t_{m_2}.

Let the execution period of rd_1 be (t_1, t_{m_1}). Let the execution period of rd_2 be (t_2, t_{m_2}). Based on the property of AOR, either $t_{m_2} < t_1$ (rd_1 is after rd_2) or $t_{m_1} < t_2$ (rd_1 is before rd_2) is true.

Case 1 ($t_{m_2} < t_1$): because $t_{m_1} < t_{m_2}$, we get $t_{m_1} < t_1$, a contradiction.

Case 2 ($t_{m_1} < t_2$): because of the contradiction with Case 1, Case 2 is true and thus rd_1 is before rd_2.

Hence, m_2 is generated by the cloud after m_1 and the theorem is proved. □

Defeating the attacks with Weaknesses 1 - 4. First, the original attacks with Weakness 1 succeeded because of semantic loss in AMT, specifically due to the loss of user identities, permissions and expiration control in the device-side policies (§4.1). Based on Theorem 1, at any time, the device-side policy is either empty/expired (by default denying any access) or is a no-loss translation of the cloud-side policy (including the user identities, user-specific permissions and permission expiration time). All the attacks discussed in §4.1 can not succeed based on our design.

Second, Weakness 2 also relates to AMT and comes with two key causes: (1) part of the on-device policy (e.g., offline access code) does not expire; (2) part of the on-device policy is not invalidated (removed) if the corresponding user is removed from the cloud-side policy. These problems/attacks are defeated in *SAPS* based on (1) Theorem 1: any on-device policy record is related to a user identity known to the cloud, and after any round of protocol execution, removed user from the cloud policy will lead to the removal of the user's policy record in the device; and (2) the DPE property: even without any new round of protocol execution, all on-device policies expire after time T_p.

Third, the attacks with Weakness 3 rely on the reordering of the policy messages (e.g., m_1 to m_3 in Fig. 3) in the absence of otherwise clearly defined causal relations between those messages (e.g., what message order is right/wrong to process on the device). Such attacks are defeated based on Theorem 2: policy messages ($Policy\_Delta$ messages in *SAPS*) are always processed by the device based on the order the messages are created by the cloud.

Last, the key problem with Weakness 4 is that at any time t, if the device and cloud have inconsistent policies (e.g., the device maintains a $key_{offline}$ for user A while A has been removed from the cloud, see §5.2), the inconsistency persists and there lacked a mechanism to audit and identify such a conflict. *SAPS* effectively defeated the attack based on the DPE property: all on-device policies expire after time T_p. *SAPS* further addressed the problem with a new auditing capability: after step 3 of any round of protocol execution, the cloud can audit the *On_Device_Policy* and check if it is inconsistent with the on-cloud policy.

Discussion of limitation. Although *SAPS* achieves key security goals and effectively defeats the MaaG attacks, we acknowledge a few limitations. In particular, at any time t, although Theorem 1 guarantees that the device-side policy, if not empty/expired, is a no-loss translation of a quite fresh version of the cloud-side policy, the cloud-side policy can still evolve (e.g., immediately after a recent round of protocol execution). Hence, it is not possible to ensure the device always has the latest policy (although this is expected because otherwise it will require constant/reliable communication with the cloud, which is not always possible for the MaaG architecture). Second, for simplicity and increased level of security, *SAPS* currently does not support the potentially complicated causal relations between policy messages, and leverages Theorem 2 to guarantee their correct order of processing (first created, first applied).

7.4 Evaluation

We implemented the protocol end-to-end to show its practicability. We deployed the MaaG IoT firmware written in `Node.js` on Raspberry Pi 4B (2GB RAM). We used the Cordova mobile application development framework and Flask server framework to implement the app and the cloud server respectively. We deployed the app on Google Pixel 6. We deployed the cloud server on Amazon EC2 (U.S. West). The response time of the cloud-to-IoT access policy synchronization on average is 6.8 seconds out of 10 rounds. As a comparison, the August/Yale online authentication protocol has an average end-to-end time of 3.1 seconds out of 10 rounds (where their cloud server is deployed in a similar location, with a similar RTT). Our implementation has significant room for optimization. Currently, a bottleneck lies in the app-to-device BLE communication where the throughput it achieves is much lower than industry benchmark. In addition, our implementation uses scripting languages (e.g., `Node.js` and python) which also contributes to the time cost.

8 RELATED WORK

Access Model/Policy Translation. Previous work [14, 21, 28, 47, 80] explored access control policy translation. This is driven by the requirement of interoperability between different systems. While it is easy to do such translation in trusted, resource-rich environments, how such translation can be done right in malicious, resource-constrained environments needs further research.

IoT Access Control. IoT generally suffers from weaker access control, often due to its unique resource constraints and design paradigms [36–39, 50, 76]. [36, 50] discovered limited forms of state inconsistency problems between IoT devices and cloud under the DGC architecture. Although our paper has a similar threat model, our contribution is that we distill the problems into access model translation and access policy synchronization. This fundamental understanding and modeling of the problem allowed us to not only discover many more root causes that can lead to state inconsistencies but also develop much stronger attacks. [39] discovered that different management channels of an IoT device might not have well-aligned security policy enforcement and lead to interference that harms security. Our paper, however, demonstrates that even the single authoritative management channel (i.e., through the companion app) can lead to insecure access controls.

On the other hand, many efforts have been devoted to improve IoT access control security [17, 38, 40, 44, 65]. [40] proposed a fine-grained context-based access control system for appified IoT platforms. [72] introduced P-Verifier, a formal verification tool that can automatically verify cloud-based IoT access-control policies. While these work take constructive steps, our work specifically addresses the cloud service and IoT devices' access model discrepancies and faulty access policy synchronization in the real world.

Smart Lock Security. [75] revealed that compromised mobile devices can leak an August smart lock's offline keys. In contrast, we are able to find protocol-level vulnerabilities in the August smart lock. [33] analyzed the security policy and the session key generation method of the August smart lock and failed to identify any weaknesses. Our research, on the other hand, finds that August smart lock's handshake key/offline key synchronization process was vulnerable, again due to our systematic modeling of the access model translation and synchronization.

Wireless Protocol Security. Much prior work has shown that wireless protocols are vulnerable to attacks from different layers. [18, 19, 63, 83] demonstrated that Bluetooth has weaknesses, allowing eavesdropping, packet injection and device spoofing. [67] demonstrated that malicious co-located apps can harm access control of BLE devices. These findings have motivated IoT manufacturers to develop ad-hoc application-layer encryption for IoT authorization/authentication [27]. This paper discovers that MaaG IoT access control systems based on application-layer encryption could fail in practice due to insecure co-operations of system components.

9 CONCLUSION

This paper systematically investigates security risks in Mobile-as-a-Gateway (MaaG) IoT, by distilling the problems into access model translation (AMT) and access policy synchronization. This has allowed us to understand the fundamental challenges in MaaG IoT and identify a variety of root causes that can lead to vulnerabilities in real-world systems. Our study demonstrates that real world manufacturers have failed to orchestrate the security responsibilities of MaaG IoT system components, which allows tremendous space for practical attacks. To mitigate the risks, we designed a coherent, secure access policy synchronization protocol and access control model to protect MaaG IoT devices from unauthorized access.

ACKNOWLEDGMENTS

This work is supported in part by NSF CNS-1652954, CNS-2145675, CCF-2124225, and Indiana University's FRSP-SF, REF, and IAS Collaborative Research Award.

REFERENCES

[1] 2022. August Smart Lock. https://august.com/products/august-smart-lock-3rd-generation.

[2] 2022. Aura Bluetooth Smart Door Lock | Kwikset. https://www.kwikset.com/aura.

[3] 2022. Chipolo ONE 4 Pack. https://chipolo.net/en-us/products/chipolo-one-4-pack.

[4] 2022. Find Your Lost Phone, Keys, or Anything with Tile's Bluetooth Tracker | Tile. https://www.thetileapp.com/en-us/store/tiles/pro.

[5] 2022. Geonfino Smart Lock. https://www.amazon.com/dp/B0957PSMBJ/.

[6] 2022. Honeywell Bluetooth Enabled Deadbolt Door Lock With Keypad, Satin Nickel | Honeywell Store. https://www.honeywellstore.com/store/products/honeywell-bluetooth-enabled-entry-deadbolt-nickel-8812309s.htm.

[7] 2022. Kwikset Aura Product Documents. https://www.kwikset.com/support/productdetail/aura-bluetooth-enabled-smart-lock#documents

[8] 2022. Level | Level Lock - The Smallest and Most Advanced Smart Lock Ever. https://level.co/products/lock.

[9] 2022. Schlage Sense™ Smart Deadbolt with Camelot trim. https://www.schlage.com/en/home/products/BE479CAMFFF.html.

[10] 2022. Ultraloq U-Bolt Pro Smart Lock | World's Most Versatile Smart Lock – U-tec. https://store.u-tec.com/products/ultraloq-u-bolt-pro-bluetooth-enabled-fingerprint-and-keypad-smart-lock.

[11] 2022. Yale Assure Lock Touchscreen, Standalone - Yale Home. https://shopyalehome.com/collections/keypad-locks/products/yale-assure-lock-touchscreen-standalone?variant=39341912162436.

[12] Giuseppe Aceto, Alessio Botta, Pietro Marchetta, Valerio Persico, and Antonio Pescapé. 2018. A comprehensive survey on internet outages. *Journal of Network and Computer Applications* 113 (2018), 36–63.

[13] Tahir Ahmad, Umberto Morelli, and Silvio Ranise. 2020. Deploying Access Control Enforcement for IoT in the Cloud-Edge Continuum with the help of the CAP Theorem. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*. 213–220.

[14] Apu Kapadia Jalal Al-muhtadi. 2000. IRBAC 2000: Secure interoperability using dynamic role translation. In *In Proceedings of the 1st International Conference on Internet Computing*. 231–238.

[15] Gianluca Aloi, Giuseppe Caliciuri, Giancarlo Fortino, Raffaele Gravina, Pasquale Pace, Wilma Russo, and Claudio Savaglio. 2016. A mobile multi-technology gateway to enable IoT interoperability. In *2016 IEEE first international conference on internet-of-things design and implementation (IoTDI)*. IEEE, 259–264.

[16] Florian Alt and Stefan Schneegass. 2022. Beyond Passwords—Challenges and Opportunities of Future Authentication. *IEEE Security & Privacy* 20, 1 (2022), 82–86.

[17] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. 2019. {WAVE}: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1375–1392.

[18] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. 2020. BIAS: Bluetooth Impersonation AttackS. In *2020 IEEE Symposium on Security and Privacy (SP)*. 549–562. https://doi.org/10.1109/SP40000.2020.00093

[19] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. 2019. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1047–1061. https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli

[20] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT.. In *NDSS*.

[21] Somchai Chatvichienchai, Mizuho Iwaihara, and Yahiko Kambayashi. 2003. Secure Interoperability between Cooperating XML Systems by Dynamic Role Translation. In *Database and Expert Systems Applications*, Vladimír Mařík, Werner Retschitzegger, and Olga Štěpánková (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 866–875.

[22] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. 2019. 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)*. 222–233. https://doi.org/10.1109/DSN.2019.00034

[23] Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2022. Practical Data Access Minimization in {Trigger-Action} Platforms. In *31st USENIX Security Symposium (USENIX Security 22)*. 2929–2945.

[24] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2021. Data privacy in trigger-action systems. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 501–518.

[25] Haotian Chi, Chenglong Fu, Qiang Zeng, and Xiaojiang Du. 2022. Delay Wreaks Havoc on Your Smart Home: Delay-based Automation Interference Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 285–302.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[27] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 73–84.

[28] Csilla Farkas, Andrei Stoica, and Parag Talekar. 2003. APTA: An automated policy translation architecture. In *Int. Conf. Computer, Communication and Control Technologies*. Citeseer.

[29] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. 636–654. https://doi.org/10.1109/SP.2016.44

[30] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized action integrity for trigger-action IoT platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*.

[31] Colin J Fidge. 1987. Timestamps in message-passing systems that preserve the partial ordering. (1987).

[32] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. 2021. {HAWatcher}:{Semantics-Aware} Anomaly Detection for Appified Smart Homes. In *30th USENIX Security Symposium (USENIX Security 21)*. 4223–4240.

[33] Megan Fuller, Madeline Jenkins, and Katrine Tjølsen. 2019. Security Analysis of the August Smart Lock. *en. In:()* (2019), 17.

[34] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking Access Control and Authentication for the Home Internet of Things ({{{{{IoT}}}}}). In *27th USENIX Security Symposium (USENIX Security 18)*. 255–272.

[35] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *31th USENIX Security Symposium (USENIX Security 22)*.

[36] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. 461–472.

[37] Blake Janes, Heather Crawford, and TJ OConnor. 2020. Never ending story: Authentication and access control design flaws in shared iot devices. In *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE, 104–109.

[38] Yan Jia, Luyi Xing, Yuhang Mao, Dongfang Zhao, XiaoFeng Wang, Shangru Zhao, and Yuqing Zhang. 2020. Burglars' IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds. In *2020 IEEE Symposium on Security and Privacy (SP)*. 465–481. https://doi.org/10.1109/SP40000.2020.00051

[39] Yan Jia, Bin Yuan, Luyi Xing, Dongfang Zhao, Yifan Zhang, XiaoFeng Wang, Yijing Liu, Kaimin Zheng, Peyton Crnjak, Yuqing Zhang, et al. 2021. Who's In Control? On Security Risks of Disjointed IoT Device Management Channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1289–1305.

[40] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms.. In *NDSS*, Vol. 2. San Diego, 2–2.

[41] Jmaxxz. 2016. Backdooring the Front Door. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Jmaxxz-Backdooring-the-Frontdoor-UPDATED.pdf.

[42] Jason Johnson, Rolf Rando, Siddharth Gidwani, and Christopher Dow. 2021. Intelligent door lock system in communication with mobile device that stores associated user data. US Patent 10,993,111.

[43] Magne Jorgensen and Martin Shepperd. 2006. A systematic review of software development cost estimation studies. *IEEE Transactions on software engineering* 33, 1 (2006), 33–53.

[44] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. 2019. JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1519–1536. https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-sam

[45] Kwok-yan Lam and Dieter Gollmann. 1992. Freshness assurance of authentication protocols. In *European Symposium on Research in Computer Security*. Springer, 261–271.

[46] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2021. Quantifying and Generalizing the CAP Theorem. arXiv:2109.07771 [cs.DC]

[47] Gregory Leighton and Denilson Barbosa. 2011. Access control policy translation, verification, and minimization within heterogeneous data federations. *ACM Transactions on Information and System Security (TISSEC)* 14, 3 (2011), 1–28.

[48] Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. 2020. T2pair: Secure and usable pairing for heterogeneous iot devices. In *Proceedings of the 2020 acm sigsac conference on computer and communications security*. 309–323.

[49] Barbara Liskov and Rivka Ladin. 1986. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual*

ACM symposium on Principles of distributed computing. 29–39.

[50] Hui Liu, Juanru Li, and Dawu Gu. 2020. Understanding the security of app-in-the-middle IoT. *Computers & Security* 97 (2020), 102000.

[51] Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. 2021. Westworld: Fuzzing-Assisted Remote Dynamic Symbolic Execution of Smart Apps on IoT Cloud Platforms. In *Annual Computer Security Applications Conference*. 982–995.

[52] Shrirang Mare, Franziska Roesner, and Tadayoshi Kohno. 2020. Smart Devices in Airbnbs: Considering Privacy and Security for both Guests and Hosts. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 436–458.

[53] Friedemann Mattern et al. 1988. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science.

[54] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. 2014. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android.. In *NDSS*.

[55] Roger M. Needham and Michael D. Schroeder. 1978. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM* 21, 12 (dec 1978), 993–999. https://doi.org/10.1145/359657.359659

[56] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. 2018. IoTSan: Fortifying the safety of IoT systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 191–203.

[57] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications.. In *NDSS*.

[58] TJ OConnor, Dylan Jessee, and Daniel Campos. 2021. Through the Spyglass: Towards IoT Companion App Man-in-the-Middle Attacks. In *Cyber Security Experimentation and Test Workshop*. 58–62.

[59] Trevor Pering, Yuvraj Agarwal, Rajesh Gupta, and Roy Want. 2006. CoolSpots: Reducing the Power Consumption of Wireless Mobile Devices with Multiple Radio Interfaces *(MobiSys '06)*.

[60] Adrian Perrig, Robert Szewczyk, Justin Douglas Tygar, Victor Wen, and David E Culler. 2002. SPINS: Security protocols for sensor networks. *Wireless networks* 8, 5 (2002), 521–534.

[61] Amir Rahmati, Earlence Fernandes, Kevin Eykholt, and Atul Prakash. 2018. Tyche: A risk-based permission model for smart homes. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 29–36.

[62] Ole André Vadla Ravnås. 2016. Frida-A world-class dynamic instrumentation framework. *URL: https://frida. re* (2016).

[63] Mike Ryan. 2013. Bluetooth: With Low Energy Comes Low Security. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan

[64] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. 1996. Role-based access control models. *Computer* 29, 2 (1996), 38–47.

[65] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2018. Situational access control in the internet of things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1056–1073.

[66] B SIG. 2016. Bluetooth core specification version 5.0. *Specification of the Bluetooth System* (2016).

[67] Pallavi Sivakumaran and Jorge Blasco. 2019. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1–18. https://www.usenix.org/conference/usenixsecurity19/presentation/sivakumaran

[68] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. {SmartAuth}:{User-Centered} Authorization for the Internet of Things. In *26th USENIX Security Symposium (USENIX Security 17)*. 361–378.

[69] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[70] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the attack surface of trigger-action IoT platforms. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1439–1453.

[71] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. 2019. Looking from the Mirror: Evaluating {IoT} Device Security through Mobile Companion Apps. In *28th USENIX Security Symposium (USENIX Security 19)*. 1151–1167.

[72] Luyi Xing, Ze Jin, Yiwei Fang, Yan Jia, Bin Yuan, and Qixu Liu. 2022. Understanding and Mitigating Security Risks in Cloud-based IoT Access Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.

[73] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. 2019. Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1415–1430.

[74] Wen Xu and Yubin Fu. 2015. Own Your Android! Yet Another Universal Root. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/xu

[75] Mengmei Ye, Nan Jiang, Hao Yang, and Qiben Yan. 2017. Security analysis of Internet-of-Things: A case study of august smart lock. In *2017 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 499–504.

[76] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, XiaoFeng Wang, and Yuqing Zhang. 2020. Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1183–1200. https://www.usenix.org/conference/usenixsecurity20/presentation/yuan

[77] Thomas Zachariah, Neal Jackson, and Prabal Dutta. 2022. The internet of things still has a gateway problem. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*. 109–115.

[78] Thomas Zachariah, Noah Klugman, Bradford Campbell, Joshua Adkins, Neal Jackson, and Prabal Dutta. 2015. The internet of things has a gateway problem. In *Proceedings of the 16th international workshop on mobile computing systems and applications*. 27–32.

[79] Eric Zeng and Franziska Roesner. 2019. Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 159–176. https://www.usenix.org/conference/usenixsecurity19/presentation/zeng

[80] Aijuan Zhang, Jingxiang Gao, Jiuyun Sun, and Cheng Ji. 2013. Declaration and Translation of Spatial Access Control Policy. *J. Softw.* 8, 5 (2013), 1132–1139.

[81] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 105–114.

[82] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1074–1088.

[83] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. 2020. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 37–54. https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-yue

[84] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3649–3666. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng

[85] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. 2019. 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)*. USENIX Association, Santa Clara, CA, 1133–1150. https://www.usenix.org/conference/usenixsecurity19/presentation/zhou