Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation

Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation

Bin Yuan[1,3,2,4,*], Yan Jia[5,7,2,*], Luyi Xing[2,†],
Dongfang Zhao[2], XiaoFeng Wang[2], Deqing Zou[1,3], Hai Jin[6,3], Yuqing Zhang[7,5]
[1]School of Cyber Science and Engineering, Huazhong Univ. of Sci. & Tech., China, [2]Indiana University Bloomington,
[3]{National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, and Big Data Security Engineering Research Center, Huazhong Univ. of Sci. & Tech., China},
[4] Shenzhen Huazhong University of Science and Technology Research Institute, China, [5]School of Cyber Engineering, Xidian University, China,
[6]School of Computer Science and Technology, Huazhong Univ. of Sci. & Tech., China,
[7]National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China

Abstract

IoT clouds facilitate the communication between IoT devices and users, and authorize users' access to their devices. In this paradigm, an IoT device is usually managed under a particular IoT cloud designated by the device vendor, e.g., Philips bulbs are managed under Philips Hue cloud. Today's mainstream IoT clouds also support device access delegation across different vendors (e.g., Philips Hue, LIFX, etc.) and cloud providers (e.g., Google, IFTTT, etc.): for example, Philips Hue and SmartThings clouds support to delegate device access to another cloud such as Google Home, so a user can manage multiple devices from different vendors all through Google Home. Serving this purpose are the IoT delegation mechanisms developed and utilized by IoT clouds, which we found are heterogeneous and ad-hoc in the wild, in the absence of a standardized delegation protocol suited for IoT environments. In this paper, we report the first systematic study on real-world IoT access delegation, based upon a semi-automatic verification tool we developed. Our study brought to light the pervasiveness of security risks in these delegation mechanisms, allowing the adversary (e.g., Airbnb tenants, former employees) to gain unauthorized access to the victim's devices (e.g., smart locks) or impersonate the devices to trigger other devices. We confirmed the presence of critical security flaws in these mechanisms through end-to-end exploits on them, and further conducted a measurement study. Our research demonstrates the serious consequences of these exploits and the security implications of the practice today for building these mechanisms. We reported our findings to related parties, which acknowledged the problems. We further propose principles for developing more secure cross-cloud IoT delegation services, before a standardized solution can be widely deployed.

1 Introduction

The popularity of Internet of Things (IoT) gives rise to the demand for effectively managing these devices, which has been supported by *IoT clouds*. These clouds are operated by both device vendors (Philips Hue, LIFX, Tuya, etc.) and cloud providers (Google, Amazon, IFTTT, etc.), offering integrated services for IoT users to control their devices across the Internet in a convenient and transparent way. Prominent examples include SmartThings [33], IFTTT [13] and Google Home [12]. Such cloud services facilitate the communication between IoT devices and users, and manage users' access to the devices: IoT devices are registered to the clouds, and users' control commands (e.g., opening a lock, typically issued through the companion app of the device) go through the clouds' authentication and authorization, ensuring only authorized users can operate a device. Today's IoT clouds tend to provide complicated functionalities like cross-vendor/cross-cloud device control, sharing of device access, device control automation, etc., as enabled by the cloud's vast computing and communication resources. Of particular importance is the capability to delegate device access across different clouds and users: for example, Philips Hue may allow Google Home to control its smart light bulb, so the owner of the bulb can manage multiple devices from different vendors all through Google Home; an Airbnb host may temporarily give the access to the smart devices in her home to her guest during his stay. Such a capability can lead to a convoluted delegation chain, whose complicated authorization operations could easily go wrong. Although threats to IoT devices have been studied before [23, 52, 56, 57, 62, 76], little has been done so far to systematically analyze and understand the security implications of this IoT delegation process.

Risks in cross-cloud delegation. Delegation of authority has been studied in the access control community for decades, with security risks such as credential leakage, incomplete revocation and incorrect policy enforcement [61, 69, 73] being discovered. Unlike the theoretic models analyzed before in which all parties run the same delegation protocol and interact through unified interfaces, real-world IoT clouds often utilize their individual, heterogeneous delegation protocols

[*] Work was done when the first two authors were at Indiana University Bloomington.

[†] Corresponding author: Luyi Xing, Indiana University Bloomington.

that may not be compatible with those of other clouds and may not have been properly verified. For example, the LIFX and IFTTT clouds delegate access to the SmartThings cloud through different protocols: LIFX issues an OAuth token to SmartThings to access LIFX bulbs, while IFTTT provides SmartThings a secret URL as a security token to access its devices (see Section 3). To work with these delegator clouds, the SmartThings cloud runs a program from each of them that implements the corresponding protocol to enable SmartThings to communicate with the devices on each delegator cloud.[1] Such a program allows the delegator to participate in further access management of the device for users on SmartThings cloud; a security risk could arise, however, when the program is not fully compliant with SmartThings' security policies (Section 3.1). Given that a standard delegation protocol (such as WAVE [45]) could take a long time to be adopted and deployed in the wild, until all compatibility and usability issues are resolved, a systematic analysis of today's IoT delegation and in-depth understanding of its security risks are of critical importance. This, however, has never been done before, up to our knowledge.

Findings. In this paper, we report the first attempt to analyze the security weaknesses of real-world IoT access delegation and mitigate their potential threats to the IoT ecosystem. Using a semi-automatic verification tool, we evaluated the delegation supports provided by 10 popular IoT clouds, including both device vendors (Philips Hue, LIFX, iHome, etc.) and IoT cloud providers (IFTTT, Amazon, SmartThings, Google, etc.). Our study shows that device delegation on these clouds is often vulnerable (Section 3) and can be exploited to gain unauthorized access to the victim's devices or impersonate the devices to perform unauthorized interactions with other devices. The consequences of such attacks are serious, ranging from completely losing control on the delegated access rights on SmartThings (Section 3.2) to leaking sensitive device IDs on Google Home that enables the attacker to unlock the victim's home door by spoofing events (Section 3.1).

Most importantly, our research shows that the heterogeneous and ad-hoc delegation processes in the wild have led to conflicting delegation policies across IoT clouds. More specifically, under today's IoT cloud delegation model, the delegator and delegatee clouds are less decoupled than expected and therefore need to be aware of each other's security constraints when determining their own delegation policies. As mentioned earlier, the SmartThings cloud delegated with access rights to a device requires the delegator cloud to upload a program, a.k.a. *SmartApp*, to help access the device from the delegator. When SmartThings further grants to its users the access to the device, however, we found that a SmartApp not compliant with the delegation policies of SmartThings (e.g., the `IFTTT` SmartApp) exposes to the SmartThings users the privilege that SmartThings cannot revoke (Section 3.1). On the other hand, SmartThings' access delegation to the Google Home cloud needs to go through Google's interface that asks for both device ID and OAuth token. Although device ID is public for many IoT clouds (Belkin, Philips Hue and MiHome) [52, 76], it is an authentication token on SmartThings [56]. Unaware of this side effect, Google Home's policy of sharing SmartThings device ID with its users enables a malicious delegatee user to directly access devices on SmartThings cloud, even after Google Home has revoked his access to the device (Section 3.1). Such incomplete information of the other party's security policies and constraints turns out to be a fundamental problem in today's IoT delegation model.

Also, an IoT cloud tends to directly adopt existing authorization protocols such as OAuth, which however cannot meet all delegation requirements in the IoT environments (Section 3.2). Particularly, we found that a malicious delegatee user on the Tuya cloud can bypass its access control, by strategically delegating his access to Tuya devices to another cloud (e.g., Google Home) and leveraging the latter's OAuth token to access the devices, even after his access right has been revoked in the Tuya cloud – a violation of the transitivity property in delegation (Section 3.2). Also problematic is the heterogeneous and custom delegation protocols, which oftentimes did not go through a rigorous verification and are therefore error-prone in their policy enforcement, causing security risks such as leaking the OAuth token to an unauthorized party (Section 3.2).

Methodology. Our security analysis was facilitated by a semi-automatic verification tool, called *VerioT*, which performs model checking for IoT delegation systems. To this end, we came up with a simple, generalized security property that captures the requirements of IoT delegation. Most challenging here is to model different real-world delegation systems, which requires manual effort to read developer documentations and user manuals of those IoT clouds, and analyze communication traffic of their companion mobile apps, to understand their delegation mechanisms and operations. To reduce such manual efforts, we leverage the observation that cross-cloud delegation can be described by combinations of *basic* types of delegation operations (Section 4.2) and corresponding data flows, such as OAuth token issuing, which are similar across different clouds. This allows us to design a modeling approach involving a base delegation model that outlines the generic operations and data flows in a cross-cloud delegation and a set of templates for different *basic* types of delegation operations. To describe a real-world delegation system, one can directly use or customize existing templates to refine the base delegation model.The new model automatically produced by our tool is then verified against the security property using Spin [38], an off-the-shelf model checker. Any

[1] Throughout the paper, we call the party (an IoT cloud or the device owner) delegating access right to another party *delegator* and the recipient of the right *delegatee*. Also, we call a cloud with direct access to a device *device vendor cloud* or simply *vendor cloud* (as it is typically operated by the vendor).

counterexample produced by the checker represents a potential attack path, which can lead to the detection of a weakness in the delegation process. `VerioT` was used in our research to find all except one vulnerabilities reported in the paper. We have made the tool publicly available [34], including the base delegation model and operation templates.

Impacts. Running `VerioT` to analyze the delegation operations on leading IoT clouds, including Google Home, SmartThings, IFTTT, Philips Hue, etc., we have discovered 6 security-critical vulnerabilities that expose millions of IoT users and hundreds of IoT clouds to security risks (Section 5). We reported all these flaws to the affected parties, including Google Home, SmartThings, Philips Hue, etc., which are taking actions to address them: e.g., SmartThings has deployed two fixes and Philips Hue claimed that they will release a fix to our reported vulnerability in their upcoming update. We are helping other cloud providers and device vendors to find solutions. The demos of our attacks are online [34].

Contributions. We summarize the contributions of the paper as follows:

- *New understanding of IoT delegation*. We performed the first systematic study on security risks in IoT device access delegation. Our research has brought to light new categories of unexpected and security-critical vulnerabilities in the delegation process of today's leading IoT cloud providers and device vendors, the serious consequences once these vulnerabilities have been exploited, their fundamental causes and the challenges in fixing them. The lesson learned will contribute to better protection of real-world IoT delegation.

- *IoT delegation verification*. We developed and released the first support for formal verification of real-world IoT delegation. Our base model, delegation operation templates, security property and refinement technique have made the first step toward convenient and automated discovery of delegation vulnerabilities, which helps secure not only today's but also tomorrow's IoT delegation operations.

Roadmap. The rest of this paper is organized as follows: Section 2 provides the background information of IoT device delegation and discusses its security requirements and potential risks; Section 3 elaborates the vulnerabilities we discovered; Section 4 describes the semi-automatic methodology we used to find these vulnerabilities; Section 5 reports a measurement study on the impacts of the delegation flaws; Section 6 discusses the design principles for developing secure delegation mechanism, the limitations of our work and potential future directions; Section 7 compares the related prior studies with our work and Section 8 concludes the paper.

2 Cross-cloud IoT Access Delegation

2.1 Background

Cloud-based IoT access. Figure 1 shows the typical procedure for accessing devices through IoT clouds. Specifically, the owner of an IoT device first registers her device to the device vendor's cloud (e.g., through presenting a client certificate embedded in a device, as an iHome smart plug does) or a third-party cloud adopted by the device vendor (e.g., KEYGMA devices are registered to Tuya cloud [16]), so the device could be managed through the cloud. As mentioned in Section 1, we call a cloud with direct access to a device *device vendor cloud* or simply *vendor cloud*. When a user attempts to access the device (e.g., through her mobile app or web console), the cloud authenticates the user and then sends her commands to the target device if the user is authorized. This IoT access paradigm has been adopted by mainstream IoT device vendors (e.g., Philips, August, iRobot, LIFX, iHome, Tuya, etc.), as well as third-party IoT clouds (SmartThings, Google, Amazon, etc.).

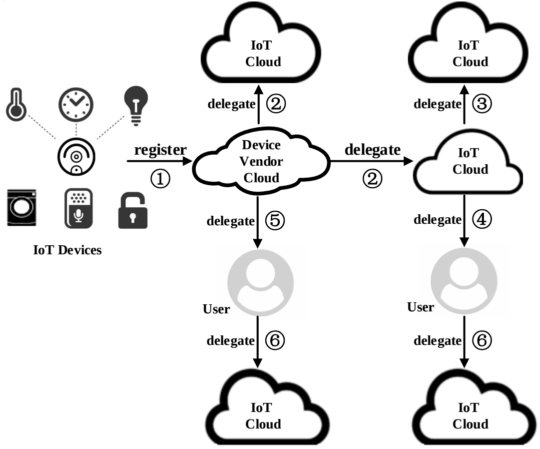

Figure 1: The complex delegation in IoT

In addition to the direct device access initiated by the user, mainstream IoT clouds, such as SmartThings, IFTTT, Amazon Alexa, etc., also allow the user to define trigger-action rules to automatically trigger her devices' actions under some events: for example, once the user's front door lock is unlocked, the cloud also turns on her living room bulb.

Cross-cloud access delegation. To control an IoT device, the user needs to install its vendor's app, which is not scalable with an increasing number of devices from different vendors one needs to manage. To address this challenge, cross-cloud IoT delegation has emerged to provide a uniform and transparent interface to handle devices from different vendors. For example, through the app of Google Home, which has been given access rights to all devices in the user's possession, she could control her smart bulb in the Philips cloud, smart lock in the SmartThings cloud, smart plug on iHome cloud, etc. This access paradigm is made possible by a delegation process.

As illustrated in Figure 1, to manage a smart lock in the

SmartThings cloud, Google Home needs to get an access token (e.g., an OAuth token [24]) from the SmartThings cloud the lock is registered to. With the token, an authorized Google user can issue commands through Google to SmartThings so as to operate on the device. For this purpose, SmartThings needs to delegate the right to access the device to Google, through OAuth [24] or other custom authorization solutions. Taking OAuth as an example, the user can trigger this delegation process through the following steps: (1) logs onto her Google Home console (e.g., a mobile app); (2) selects "set up device" to enter her SmartThings account credential; (3) if the user is allowed to use the device, SmartThings generates an access token and forwards it to Google Home. Such cross-cloud delegation has been supported by all mainstream IoT clouds.

2.2 Complexity in IoT delegation

Delegation chain. Real-world IoT delegation often involves multiple parties and can become quite complicated, as illustrated by Figure 1. As we can see, the access right to an IoT device is first given to its *device vendor cloud* (①). The vendor then delegates the access to a *delegatee cloud* (②), such as Google Home, for controlling a user's devices managed by different vendor clouds. The *delegatee cloud* may further hand over the access right to another (delegatee) cloud (③). On each of these clouds, access to the device can be granted (by the device administrator) to one or more (delegatee) users (④ ⑤). Along the delegation chain, the (delegatee) user may further give her access to another (delegatee) cloud (⑥).

Cross-cloud delegation mechanisms. Different IoT clouds today have their own authorization mechanisms to delegate device access rights to or receive delegations from other clouds and their own users. Each party on a delegation chain is expected to follow the mechanisms (and their input constraints) of its *upstream* (delegator) and *downstream* (delegatee) actors. Such heterogeneous authorization gets each IoT cloud entangled in the device management of another cloud, even after the delegation happens. In our research, we analyzed the typical authorization mechanisms as deployed on 10 mainstream IoT clouds, which are presented below:

• *OAuth and its customizations*. OAuth is an open standard for access delegation [24], which has been widely adopted by IoT clouds. A problem is that OAuth is not designed for IoT, and therefore some IoT clouds have customized it to facilitate cross-cloud device management. An example is *Actions on Google* protocol, which is a customization to OAuth by Google [1]: any *device vendor cloud* (such as Philips Hue, LIFX, iHome) that delegates access tokens (i.e., OAuth token) to Google Home is required to provide a set of information about the target device(s), including device IDs, device types, device names, etc. Such information is used by Google for cross-cloud device control, which allows the Google Home user to find and operate on devices based on their IDs, names, etc., albeit the devices are actually behind another cloud.

• *Custom authorization*. We also found that IoT clouds can use custom, sometimes ad-hoc authorization mechanisms for cross-cloud delegation. For example, IFTTT cloud delegates access to SmartThings cloud through a secret URL: when a SmartThings user needs to access the device behind IFTTT, SmartThings sends her requests to IFTTT through the URL. In the meantime, SmartThings asks its delegator to upload a SmartApp (e.g., the *IFTTT* SmartApp) for communicating with both the delegator-side device and its client who uses the device.

2.3 Security Requirements

As mentioned earlier, cross-cloud IoT delegation involves different actors, with different security policies and complicated, sometimes ad-hoc enforcement. Access control under this circumstance faces unique challenges and is expected to meet unique security requirements, as summarized below:

Safe and consistent delegation policies. IoT delegation involves parties from different organizations (vendors, clouds, users), with discrepant security needs. Under the delegation model deployed on today's clouds, a party on a delegation chain could get involved in another party's management of a device. Therefore, in the absence of a full picture of other parties' security constraints, a delegation process could get to the situation where a delegatee's policy could bring in a risk that could put a delegator's security in jeopardy. So ideally, delegation policies across all actors on a chain should be consistent with each party's individual security policies, ensuring that they will not be exposed to new threats during the whole process.

Non-bypassable and transitive delegation control. On a delegation chain, access rights to an IoT device could be distributed across multi-parties. Enforcement of a delegation policy, therefore, is expected to be comprehensive, blocking all avenues of unauthorized access. Also important is the transitivity in delegation control: once a delegator enforces a policy (e.g., revoking its delegatee party's access right), all downstream parties should all follow suit (e.g., even access rights further delegated out by the delegatee should also be automatically revoked).

2.4 Threat model

We define two user roles in the distributed IoT system, the administrator and the delegatee user. The administrator can be a device owner or a system administrator of an organization. The administrator can delegate the access to IoT devices to other users – the delegatee user. Delegatee user's access is subject to revocation and expiry. The delegatee user may further delegate to others.

In our research, we consider the system administrator and the IoT clouds to be honest, while the delegatee user can be malicious, who may attempt to get unauthorized access to IoT devices. We assume that the malicious user would make full use of his power to acquire the credentials and useful information he is not entitled to access: e.g., making API requests to gain extra credentials, extracting information from system logs, official developer documents and captured network traffic generated by his mobile app. In the meantime, we do not consider the adversary capable of eavesdropping on the communication between other parties.

3 Security of Cross-Cloud IoT Delegation

In this section, we report a security analysis on cross-cloud IoT delegation operated by 10 leading IoT clouds, including Google Home, SmartThings, IFTTT, Philips Hue, August, LIFX, Tuya, etc. Through discovery of five flaws and construction of their end-to-end attacks (see video demos online [34]), our study shows that in the absence of a standardized, verified cross-cloud delegation protocol, delegation across mainstream IoT clouds is hard to make right, often containing serious flaws in its policy design or enforcement.

One of the flaws (Flaw 4, see below) was found manually, which led to this research and our development of `VerioT` that helped discover all other flaws, based on our modeling of real-world cross-cloud delegation and formal verification (Section 4). With respect to the two security requirements summarized earlier (Section 2.3), we classify all flaws identified in our study into two categories: (1) inconsistent security policies between the delegator and delegatee clouds (Section 3.1); (2) inadequate enforcement of delegation transitivity in the presence of customized, often ad-hoc delegation management across real-world IoT clouds (Section 3.2).

3.1 Inadequate Cross-Cloud Coordination

As mentioned earlier, under the heterogeneous and often ad-hoc authorization on today's IoT clouds, a cloud on a delegation chain often cannot decouple its access delegation management from those of other clouds, and thus easily gets involved in others' access control decisions. So it is important for these clouds to be aware of each other's security assumptions and constraints. Such a coordination, however, is not in place today, as discovered in our research, which brings in security risks to the parties on the chain. Here, we report two vulnerabilities discovered on popular IoT cloud services that characterize the real-world challenges in securing cross-cloud IoT access management.

Flaw 1: Device ID disclosure. As mentioned earlier (Section 2.2), clouds that delegate device access to Google Home must follow an OAuth protocol customized by Google: the delegator cloud is required to provide both an OAuth token and its device information to Google. For example (see Figure 2), to enable a Google Home user to manage devices behind the SmartThings cloud (e.g., smart lock, smart switch, etc.), Google needs to be given both an OAuth token (through a regular OAuth process), and additionally the device information (e.g., device IDs, device names, etc.) from SmartThings. Such device information is further passed to the Google Home user to command the target device.

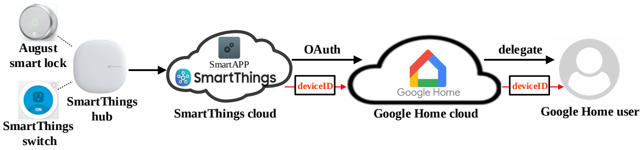

Figure 2: Google Home leaks the *deviceID* of SmartThings switch, enabling the adversary to unlock smart lock

In our study, we found that such a delegation process introduces a new security risk, due to Google's lack of knowledge about the security implication of SmartThings' device ID. Although the device ID normally just serves the purpose of locating its device on a cloud (such as Philips Hue and MiHome), it is also used as an authentication token on SmartThings for its trigger-action management: by presenting the ID of a device (a 32-digit string, unique to a device under SmartThings), one can issue events (e.g., temperature change, switch toggled, presence detected, etc.) to the SmartThings cloud on behalf of the device (through the *sendLocationEvent* API, see our PoC attack); such events can further trigger other devices managed under the SmartThings cloud (through trigger-action rules, see Section 2.1). Note that, although the presence of the device ID allows an event to be issued and related operations to be triggered on SmartThings (according to predefined rules), it is not an authorization token for device access: that is, an unauthorized party cannot use the ID to command the device he is not entitled to access.

Unaware of this side effect, Google discloses this ID to any client with the access right to the device, which brings in the security hazard. Specifically, on Google Home, as long as the administrator (e.g., an Airbnb host, a property manager) delegates a device's access right to a user once (e.g., an Airbnb guest, a tenant), the ID of the SmartThings device is permanently exposed. Even after the delegatee's access is revoked on Google Home (e.g., after he checks out of the Airbnb apartment), he still retains the capability to fake events of the device using the ID and triggers the administrator's other devices. Depending on the trigger-action rules configured by the owner/administrator, the fake events can open a smart lock (letting in an unauthorized individual), turn off an alarm, etc., through the SmartThings platform [10].

In our research, we performed a measurement study on the consequences of the attack (Section 5.2), and found that potentially many vendors could be affected by such a flaw. Further, we did not find any mechanisms in place to allow the clouds like Google (delegated with device access rights from almost 1,000 vendors) to get information about their delegators' security assumptions and constraints, based upon the documentations we inspected (see Section 5.2).

PoC exploit on Flaw 1. Exploiting the above weakness, we implemented an end-to-end PoC attack on our own devices – an August smart lock and a SmartThings switch (a virtual switch that can be toggled in mobile app – not a physical switch). The experiment setting is outlined in Figure 2. Specifically, the victim set a trigger-action rule on the SmartThings cloud: if the switch is toggled, then lock/unlock the smart lock; also, she linked her SmartThings account to Google Home. Then through Google Home, she granted the access right of the switch to a malicious user (e.g., an ill-intentioned Airbnb guest). At this point, the attacker obtained the SmartThings device ID of the switch by inspecting the network traffic between his Google Home mobile app and the Google cloud. Later, the victim revoked the malicious user's access right and he could not control the device from his Google Home app. However, using a simple SmartApp of SmartThings (see our source code online [34]), which sends fake *switch.off* events to the SmartThings cloud with the switch's device ID, the attacker was able to open the smart lock.

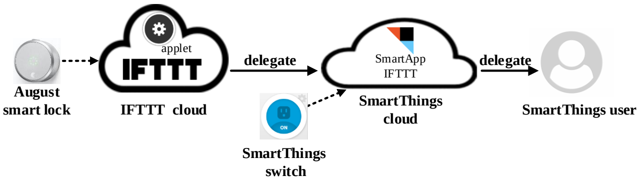

Figure 3: Security policy confliction between IFTTT cloud and SmartThings cloud

Flaw 2: Leaking secret of delegatee cloud. Delegation Flaw 1 shows that a delegatee cloud (Google) could leak the secret of its delegator cloud (SmartThings). Surprisingly, we found that such unintended information disclosure could also go other way around: a delegator cloud can also expose the sensitive data that the delegatee cloud intends to protect. The problem affects multiple leading IoT clouds (SmartThings, IFTTT, etc.). Again it is caused by the entangled delegation process, with both the delegator and the delegatee involved in the other's authorization process, and the lack of coordination to align their security policies.

Unlike Google that uses a delegation protocol with a mandatory input interface its delegator cloud must follow, SmartThings cloud provides a flexible mechanism: its delegator is allowed to upload a software module called *SmartApp* [35] to SmartThings to help execute its delegation protocol and manage the access to the device under the delegator cloud. As an example, the IFTTT cloud delegates access to its device through sharing a secret URL with SmartThings: through the URL, when a SmartThings device issues an event (e.g., a smart switch is operated), the SmartThings cloud can trigger the actions of an applet on IFTTT to control another device behind the IFTTT cloud (e.g., August smart lock, see Figure 3), based upon pre-defined trigger-action rules. This rather adhoc delegation protocol runs on the SmartThings side by an IFTTT SmartApp (Figure 3), which acts as an interface between the two clouds. With the IFTTT SmartApp, the user can define such trigger-action rules to connect the devices (e.g., the switch and the lock) across the clouds. As a result, her operation on the SmartThings device will be used by the SmartApp to invoke the activities on the device behind IFTTT cloud.

Since the IFTTT SmartApp runs on the SmartThings cloud and interacts with the end user, it is important to make sure that the SmartApp strictly follows SmartThings' security policies. Our research, however, shows that these policies have not been fully respected by the IFTTT SmartApp, possibly due to the lack of coordination between the two clouds. Specifically, we found that by calling an API provided by the IFTTT SmartApp, the SmartThings user can get the secret URL. This violates SmartThings' policy that allows a device administrator (e.g., an Airbnb host) to temporarily delegate the access right on her devices to a user (e.g., delegating the SmartThings switch to an Airbnb guest to operate the lock) and later revoke the right. That is, once the delegatee user acquires the URL, he retains a direct channel to communicate with the IFTTT device, even after the administrator revokes his access to the device on SmartThings: for example, through the URL, which serves as an authentication token, the user can send a fake event (e.g., switch is off) on behalf of the SmartThings device (e.g., the smart switch) so as to trigger the action on the IFTTT side (e.g., open the lock).

PoC exploit on Flaw 2. We conducted a PoC attack to exploit the flaw. As outlined in Figure 3, we configured our August smart lock on the IFTTT cloud, such that an applet in the IFTTT cloud will open/close our lock upon receiving the switch event from the SmartThings cloud – a normal usage scenario intended by the IFTTT platform; we also configured our smart switch on SmartThings: when the switch is turned on or off, an event will be issued by the IFTTT SmartApp through the IFTTT cloud, leading to different operations on the lock.

After that, we temporarily invited a "malicious" user to access the switch on the SmartThings cloud. Note that, on SmartThings, devices a user can access are organized as a group called *location*, which also includes the SmartApps related to the devices (e.g., the IFTTT SmartApp); also, *location* is SmartThings' smallest unit for device delegation: if the administrator wants to give the access to devices at a *location*, he needs to pass the control on the *location* to the delegatee user. In our attack, with access to the *location*, the malicious delegatee user could obtain the secret URL from the IFTTT SmartApp by calling the Web API it hosts (a URL endpoint, such as `https://graph.api.smartthings.com/api/smartapps/[32-digit-string]/subscriptions`, specific to the IFTTT SmartApp in a particular *location*). To get the Web API, the delegatee user called a public Web API of the SmartThings cloud (`https://graph.api.smartthings.com/api/smartapps/endpoints`), which is designed to return to a client the Web APIs available in the *location* that the client has access to. In our attack, by simply calling the returned Web API, the delegatee user obtained the secret URL from the IFTTT SmartApp; then by posting an HTTP request to the URL, the user was able to trigger the IFTTT applet to open the smart lock, even after his access had been revoked in the SmartThings cloud.

Discussion. Here, the problem comes from the misaligned security policies between SmartThings and IFTTT. Specifically, the IFTTT SmartApp simply trusts any user with access to the devices, and discloses the URL – a security token – to the malicious delegatee user. Such an operation, however, completely invalidates SmartThings' enforcement of its delegation revocation policy. Also importantly, since the URL is hosted by the IFTTT cloud, there can be no easy way for SmartThings to revoke the URL, without the proper policy coordination from IFTTT or a mechanism supported by IFTTT for SmartThings to revoke the URL anytime it wants (e.g., once SmartThings revokes access of a delegatee user). However, our study indicates that *proper cross-cloud policy coordination is absent in today's IoT ecosystem.*

Responsible disclosure. We reported both flaws to affected parties including Google Home, SmartThings, etc., who acknowledged the seriousness of the problems. SmartThings awarded us through their bug bounty program.

3.2 Inadequate Policy Enforcement

In addition to conflicting security policies across clouds, the access delegation mechanisms developed by individual clouds also turn out to be ad-hoc, likely due to the constraints of their systems' functionalities and the absence of a standardized IoT delegation protocol. This leads to a problem that those real-world delegation operations often have not been rigorously verified and therefore their enforcement of delegation policies can be vulnerable. Following we elaborate on three security-critical flaws discovered from popular IoT clouds, which demonstrate the importance of systematic security analysis on today's heterogeneous, ad-hoc IoT delegation ecosystem, as we propose in the paper.

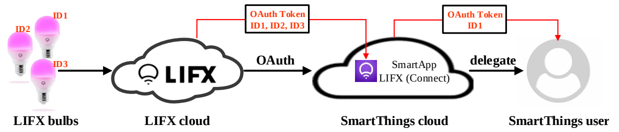

Figure 4: Policy violation between LIFX and SmartThings

Flaw 3: Exposing hidden devices in the delegator cloud. LIFX [17] is a popular IoT manufacturer whose cloud can delegate device access to other clouds, such as SmartThings, for example, when the device administrator wants to use SmartThings to manage all her devices. To support the delegation protocol of LIFX, the SmartThings cloud runs a LIFX SmartApp for the cross-cloud device access (Figure 4), like the delegation from IFTTT (Flaw 2). Again, the LIFX SmartApp not only serves as the interface between the two clouds, but also helps SmartThings manage the access to LIFX devices.

As mentioned earlier (Section 3.1), on SmartThings, devices that the user can access and related SmartApps are grouped into a *location*, including devices behind the delegator clouds. In reality, however, a property manager or an Airbnb host (the administrator) may not want to give her tenant access to *all* her devices (e.g. the smart lock for the owner's room in a rented-out apartment). This is supported by the LIFX SmartApp in her *location*, which can be authorized to access all LIFX devices of the administrator (with the OAuth token issued by the LIFX cloud), and in the meantime can be configured to expose only a subset of the devices to the *location*. As a result, the delegatee user (e.g., the tenant) will only be able to use the subset of devices at the *location* he is given access to.

However, we found that the LIFX SmartApp has not been properly protected on the SmartThings cloud. It turns out that the delegatee user on SmartThings is allowed to read from the private storage of the SmartApp at the location, which allows him to obtain the OAuth token kept there. This exposure has serious consequences. No longer can the administrator hide some LIFX devices from her delegatee, who can retrieve all device IDs from the LIFX cloud through its *List Lights* API [19] using the OAuth token, and further use the IDs and the token to command any device under the administrator's control through the *Set State* API [20]. Even more seriously, this unauthorized privilege will be kept by the user even after his right has been revoked by the administrator on SmartThings. Further, our measurment study (Section 5.2) shows that the problem also affects tens of other IoT vendors that delegate to SmartThings access to their devices.

PoC exploit on Flaw 3. To exploit the above weakness, we performed an end-to-end attack with our own LIFX bulbs. As outlined in Figure 4, we first delegated all LIFX bulbs to the SmartThings cloud and configured the LIFX SmartApp to expose only bulb$_1$ (the one with `ID1`) to our SmartThings *location*. Then, we delegated the SmartThings *location* to the malicious user for accessing bulb$_1$ (with all other bulbs hidden). However, since the malicious delegatee gained the access to the *location*, he successfully obtained the OAuth token from SmartThings IDE system [37], SmartThings' Web-based management console that shows SmartApps in the *location* and their storage. With the OAuth token, the malicious user could acquire IDs of the administrator's other bulbs from LIFX *List Lights* Web API (`https://api.lifx.com/v1/lights/all`, used to list all devices available to a client) [19] and then control them remotely [20].

Flaw 4: OAuth pitfall. Not only do custom delegation mechanisms operated by today's mainstream IoT clouds (e.g., Google, SmartThings, IFTTT, etc.) all contain serious security weaknesses (Flaw 1-3), but our research further shows that even a direct application of OAuth [24] – a cross-service delegation standard, to the complicated IoT delegation turns out to be error-prone. The new security problem we discovered affects several IoT vendors, including KEYGMA, MOES, Useelink, etc. (see Section 5.2)

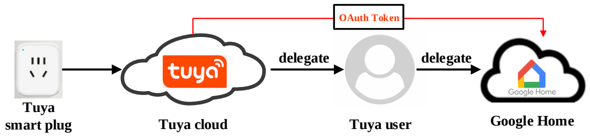

Figure 5: Independent OAuth Tokens for third-parties

Specifically, let us use Tuya as an example (Figure 5). The cloud allows the user to control her devices through Google Home. To delegate the privilege to Google, Tuya implements the standard OAuth protocol (Section 2): the user on Google Home can enter her Tuya credentials, and if she is allowed to access the device, Tuya will forward an OAuth token to Google Home, enabling her to control the device.

To find out whether Tuya's OAuth protocol can ensure secure delegation, we inspected the OAuth specification [39] in our research, and found that one of its recommended implementation options, which has been taken by Tuya, actually violates the *transitivity* property expected in IoT delegation (see Section 2.3). When a service gives an OAuth token to its delegatee service, the token can be issued on behalf of either (1) the user who initiates the OAuth process (e.g., the user in Figure 5), or (2) the delegatee service (e.g., Google Home in Figure 5).[2] In our study, we found that Tuya's implementation of OAuth takes the second option, which introduces a serious security risk. Specifically, after a Tuya user delegates to Google her device access, the OAuth token Tuya issues to Google is not on behalf of the user but in the name of Google. As a result, when the user's access right is revoked on the Tuya cloud, he can still access the device through Google Home using its OAuth token, since the token from Tuya's perspective is independent of the user so it will not be invalidated when the user's access right (which has already been delegated out to Google) is taken away.

Again, this weakness can be exploited in the real world, e.g., by a malicious Airbnb guest (or property tenant) to retain unauthorized device access. Specifically, the administrator on the Tuya cloud (e.g., an Airbnb host or property manager) delegates device access to the user, and later revokes the access through the Tuya cloud's management console. However, if the user has already strategically delegated his device access to Google Home, he will still be able to stealthily operate on the devices through Google Home, even after his access is revoked by the Tuya cloud.

PoC exploit on Flaw 4. We conducted a PoC attack to exploit the vulnerability. As shown in Figure 5, we first delegated the access to a Tuya smart plug to a malicious Tuya user through the Tuya cloud. Then, the delegatee further gave his privilege to his own Google Home account. After the user's access right was revoked on the Tuya cloud, he could still use his Google Home app to control the plug.

Discussion. This problem was first discovered through a manual check of the Tuya delegation process, which motivated this research. This case inspired us that the cross-cloud delegation in IoT can introduce new risks compared to traditional cross-website delegation scenarios, due to the different application paradigm and security requirements. For example, in a traditional scenario, when a user delegates access to her Facebook account to another Website via OAuth, her Facebook account would be almost impossible to be revoked by another party, and thus the need to invalidate the OAuth token following a revocation of the user's own Facebook access is less prominent. That is, delegation transitivity (see Section 2.3) is a less prominent security risk there. In contrast, delegating access to an untrusted user (e.g., an Airbnb guest or property tenant) and frequent access revocation in IoT context are commonplace. This requires IoT applications to appreciate delegation transitivity when applying OAuth, which, however, is less understood before.

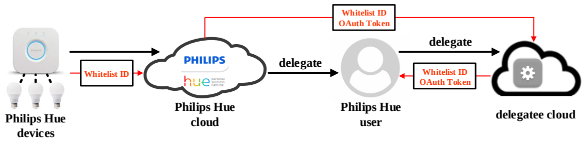

Figure 6: Insecure revocation of Philips Hue

Flaw 5: Abusing cross-cloud delegation API. As mentioned earlier, IoT device vendors' clouds (*device vendor cloud*) allow the user to operate on her devices through either the *device vendor cloud*, or the delegatee cloud. In our study, we found that an unauthorized user in the *device vendor cloud* can abuse the delegation APIs provided to the delegatee cloud, and get unauthorized device access, as elaborated below.

Philips Hue lets the device administrator (e.g., an Airbnb host or property manager) delegate device access to another Philips user (e.g., an Airbnb guest or property tenant). For example, Figure 6 shows that the delegatee user is given access to a Philips Hue bridge. Use of the Hue bridge can be done through the delegatee user's mobile app: (1) he first presses a physical button on the device (to enable a binding process); (2) the Philips app automatically fetches a secret token, called `whitelistID`, from the device through the local network they are all connected to; (3) the user then logs into his Philips app to obtain an OAuth token from the Philips cloud. With these two tokens, the user can issue commands to

[2] OAuth protocol specifies that *"OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf"* [39].

the Hue bridge through the Philips cloud. The cloud checks the OAuth token, and forwards the commands to the device, which verifies `whitelistID`. To revoke the delegatee user's access (e.g., after the tenant/guest checks out), according to its official documentation [30], Philips takes an easy path: the administrator just needs to go to her cloud console to remove the delegatee's `whitelistID`. As a result, the delegatee's future commands will be denied by the device, since it already drops his `whitelistID`.

Although the Philips delegatee user apparently loses his access to the device, this revocation enforcement turned out to be incomplete: even without `whitelstID`, the delegatee user's account still remains on the device's access list maintained by the Philips cloud. This may not be exploited if we look at the Philips cloud alone, but allows the delegatee user to re-obtain the access to the device by abusing Philips Hue's cross-cloud delegation APIs, which is another avenue to access the device.

Specifically, the Philips cloud has an API interface for delegating device access to another IoT cloud, allowing a user to remotely control Philips Hue devices from the delegatee cloud. For this purpose, the user enters her Philips credentials in the delegatee cloud, which then calls Philips Hue cloud delegation API [28]; this will return not only an OAuth token but also a fresh `whitelistID` generated by the device, which based on the Philips cloud-side access list is accessible to the user. With the two tokens, the delegatee cloud can issue commands to the Philips Hue cloud to operate the device (through Philips Hue remote control API [31]). Such a cross-cloud delegation mechanism can be utilized for an attack: the Philips delegatee user whose access has been revoked can leverage a delegatee cloud (e.g., SmartThings, see PoC exploit below) to still control the Philips device.

PoC exploit on Flaw 5. In our attack, we used SmartThings as the delegatee cloud, to gain the unauthorized access to our Philips Hue bridge. In particular, SmartThings allows us to upload a SmartApp to work with the Philips Hue cloud, which gives us a vantage point to observe the internal operations on the delegatee cloud side (e.g., what the delegatee can receive from Philips). In our implementation of the SmartApp, we utilized the SmartThings service (e.g., the Web Services SmartApp [41]) to construct an OAuth client and registered a service with Philips Hue to initiate the delegation process [27]. By doing so, our SmartApp successfully invoked Philips delegation APIs to obtain a fresh `WhitelistID` and OAuth token. This enabled us to get access to the Philips Hue bridge even after our access had been revoked on the device.

Responsible disclosure. We reported all the problems to related parties, e.g., Samsung SmartThings, Tuya, Philips Hue, which are taking serious actions to address them: SmartThings and Tuya have deployed a fix, and Philips Hue confirmed that they will release the fix to our reported vulnerability in their upcoming update.

4 System Modeling and Formal Verification

In this section, we elaborate on the design and implementation of `VerioT`, our semi-automatic tool for detecting delegation flaws in real-world IoT clouds.

4.1 Overview

At a high level, IoT delegation systems should ensure that the unauthorized delegatee user should not have a path to access IoT devices which he is not entitled to access. To detect security flaws in real-world delegation systems, our approach is to model their delegation operations as a transition system, and leverages a model checker to verify whether pre-defined security properties hold in the model. The counterexamples reported by the model checker indicate security flaws in the IoT cloud systems that are described by the model. Also, the flaws reported by `VerioT` are manually validated on real-world IoT clouds to confirm their presence.

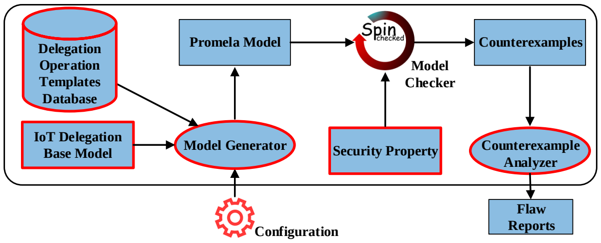

Figure 7: The architecture of `VerioT`

Architecture. Since different clouds often support different sets of delegation operations (e.g., either issuing an OAuth token or secret URL to its delegatee cloud, or hosting APIs for delegatee cloud to invoke, etc.), we cannot build a single unified model that can describe any clouds and their corresponding delegation operations. Hence, our approach is to model each specific set of real-world clouds between which delegations can happen (e.g., LIFX cloud and SmartThings cloud in Figure 8), called a *delegation setting* (or *dele-setting*). The model of a *dele-setting* then goes through our model checker for flaw detection.

To this end, we built `VerioT` that includes 3 core components: a *model generator*, a *model checker*, and a *counterexample analyzer* (or simply *analyzer*), as outlined in Figure 7. More specifically, *model generator* generates the model for each *dele-setting* found in the real-world. To this end, *model generator* takes as input a `configuration` file that lists the actors (the delegator and delegatee clouds, user, device) in the *dele-setting* and delegation operations supported by the actors (e.g., issuing a new OAuth token), and generates a state machine model for the *dele-setting*. The model describes the states of all actors (the clouds, user, etc.) in the system, and delegation operations an actor can perform which triggers state transitions; an actor in a state records its set of data that have been generated and transferred between actors due to delegation operations, e.g., an OAuth token it obtained from its delegator through an OAuth operation (Section 4.2). The model is specified using the Promela language [32] – Promela models can be verified with the off-the-shelf model checker Spin [38] that is used in our research (Section 4.3). Internally, to generate a model, *model generator* leverages a general base model (with a few actors and basic types of delegation operations specified in Promela language), and extends it by adding additional operations supported in the *dele-setting* to the base model, which are already modeled as a one-time effort and stored in our *delegation operation templates database* (Section 4.2).

For detecting delegation flaws, the model generated by *model generator* then goes through our *model checker* for verification with respect to pre-defined security properties (Section 4.3). Specifically, *model checker* reports a counterexample if it can find a state that has an access path across actors in the system that enables an unauthorized user actor to reach the device actor. More specifically, since the state machine model records the dataset each actor holds, if a user actor holds a token (e.g., OAuth token) that is issued by a cloud actor for accessing a device, we consider the user has an access path to access the device (via the cloud), no matter through what operations the user actor obtained the token (e.g., through regular OAuth operation, invocation of cloud delegation APIs, etc.). When the user actor is in a state following a delegation revocation operation, while he still has a path to the device, *model checker* reports a counterexample. Then the counterexample is analyzed and manually confirmed through proof-of-concept exploit (Flaw 1-3 and Flow 5 in Section 3) on real IoT clouds.

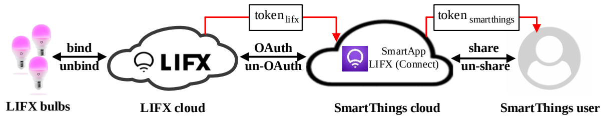

Figure 8: Delegations between LIFX and SmartThings

Example. Here, we take Flaw 3 (Section 3.2) as an example to describe how our approach detects IoT delegation flaw. In our model of *dele-setting* for LIFX and SmartThings (outlined in Figure 8), each actor is specified in Promela language as a variable (e.g., $LIFX\_bulb$, $LIFX\_cloud$, $SmartThings\_cloud$, $SmartThings\_user$, see section 4.4); each delegation operation is represented by a function that can be called on the actor variable(s), e.g., actor $LIFX\_cloud$ can perform $OAuth$ operation with actor $SmartThings\_cloud$, which generates a new token $token_{lifx}$ and gives it to the latter actor; for a state transition, a non-deterministic choice is made to randomly choose one operation to execute. In the state machine model, after the user's access is revoked by SmartThings cloud through the revocation operation (`un-share` in Figure 8), the checker verifies the current state using our security property – he should not have an access path to reach the device (see the formal definition of security property in Section 4.3). To this end, the checker inspected the data each actor in the current state holds, and found a counterexample: the user actor holds a token issued by LIFX cloud (i.e., OAuth token $token_{lifx}$), which he obtained from reading the storage of LIFX SmartApp (in SmartThings) in a previous state (before his access is revoked); such a token allows the user, whose access right has been revoked on SmartThings cloud, to still access the device through LIFX cloud.

4.2 Modeling IoT Delegation

The state machine model. We model IoT delegation as a state machine $\mathcal{M} = (\mathcal{A}, \mathcal{S}, \mathcal{O}, \mathcal{T}, s_0)$. Here $\mathcal{A}$ is the set of actors (clouds, devices, users); $\mathcal{S}$ is a set of states, in each of which an actor can perform a delegation related operation (e.g., issuing an OAuth token to another actor, invoking an API); each state records the data that each actor holds (e.g., an OAuth token, device ID, etc., obtained through delegation operations) and the access control list if the actor maintains one; s_0 ($s_0 \in \mathcal{S}$) is the initial state where no delegation operation has been performed in the system. $\mathcal{O}$ is a finite set of delegation related operations (e.g., `OAuth` – issuing an OAuth token, see definition below). $\mathcal{T}$ is a transition function that drives the system to transit from one state to the next.

Specifically, for each actor a_i ($a_i \in \mathcal{A}$), we use two data sets, $Recv_{a_i}$ and $Issu_{a_i}$, to record the tokens a_i *received* (from its delegators) and *issued* (to its delegatees) during delegation operations, respectively. For example, when a_i issues an OAuth token to a_j during an `OAuth` operation, the token will be recorded in both $Issu_{a_i}$ and $Recv_{a_j}$. Further, each cloud actor in $\mathcal{M}$ maintains an access control list, i.e., a relation that maps the tokens it issues and the tokens it receives. For example, the SmartThings cloud (see Figure 8) sends out a token ($token_{smartthings}$) to the user, which is mapped to the token SmartThings receives from the LIFX cloud ($token_{lifx}$) for accessing a certain LIFX bulb; intuitively, when the user presents $token_{smartthings}$ to the SmartThings cloud to access the bulb, based on the mapping, only if $token_{smartthings}$ is mapped to the $token_{lifx}$ – an access control check – will the SmartThings cloud forward the access request to the LIFX cloud together with $token_{lifx}$. To model such access control mapping maintained by a_i, we use a set ACL_{a_i}, which consists of a set of 2-tuple $(token, T)$, where $token \in Issu_{a_i}$, and $T \in P(Recv_{a_i})$[3]; intuitively, for example, the mapping from $token_{smartthings}$ to $\{token_{lifx}\}$ indicates that the access right represented by $token_{lifx}$ (the access right to the bulb delegated out by LIFX cloud), is delegated to $token_{smartthings}$ (by SmartThings cloud).

Def. 1. **State**: A state $s_k (s_k \in \mathcal{S})$ records each actor's token

[3] $P(x)$ is the power set of set x. We use $P(Recv_{a_i})$ (versus $Recv_{a_i}$) since the cloud such as SmartThings can map a token it issued to its user to multiple tokens it received from its delegator cloud (to access multiple devices in the delegator cloud).

sets and ACL set: $s_k = \bigcup_{a_i \in \mathcal{A}} \{ Recv_{a_i}, Issu_{a_i}, ACL_{a_i} \}$, among which, the initial state $s_0 = \bigcup_{a_i \in \mathcal{A}} \{ \varnothing, \varnothing, \varnothing \}$.

Def. 2. **Operation**: A delegation operation from a_i to a_j indicates a_i grants/revokes access right to/from a_j, which implies changes to their token sets and ACL sets. For example, the `OAuth` operation (`OAuth` $\in \mathcal{O}$), e.g., `OAuth` between the LIFX cloud (a_i) and the SmartThings cloud (a_j) in Figure 8, modeled as `OAuth`(a_i, a_j, T), performed by a_i, to delegate access right T ($T \in P(Recv_{a_i})$) to a_j, is defined as:

$$\begin{cases} token := newOAuthToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{ \ token \ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{ \ token \ \} \\ ACL_{a_i} := ACL_{a_i} \cup \{ \ (token, T) \ \} \end{cases}$$

Further, a_i may need to revoke the issued OAuth token; correspondingly, the `un-OAuth` operation (`un-OAuth` $\in \mathcal{O}$), modeled as `un-OAuth`$(a_i, token)$, performed by a_i, to revoke token *token*, is defined as:

$$\begin{cases} Issu_{a_i} := Issu_{a_i} - \{ \ token \ \} \\ ACL_{a_i} := ACL_{a_i} - \{ \ (token, T) \mid (token, T) \in ACL_{a_i} \} \end{cases}$$

Altogether, we modeled nine delegation operations, as summarized in Table 1 (with their formal definitions in Appendix A).

Table 1: Summary of delegation operations

Operation Type	Semantic Meaning
`bind`	bind (register) a device to its vendor cloud
`unbind`	unbind the device from vendor cloud, e.g., reset
`share`	delegate an access to a user
`un-share`	revoke an access of a user
`OAuth`	authorize another party through OAuth protocol
`un-OAuth`	revoke an OAuth authorization
`setTrigger`	set a trigger-action rule
`un-setTrigger`	remove the trigger-action rule
`APIRequest`	send API requests (e.g., Web API request)

Def. 3. **Transition** : $\mathcal{T}$: $\mathcal{S} \times \mathcal{O} \to \mathcal{S}$ is a function that drives the transition from one state to the next. For example, $\mathcal{T}(s_i,$ `OAuth`$) = s_j$ (where $s_i, s_j \in \mathcal{S}$) indicates that an `OAuth` operation in state s_i drives the system to state s_j.

Generating models for different real-world *dele-settings*. Based on the model definition, our *model generator* models each *dele-setting* and generates the model specified using the Promela language [32]. Given the heterogeneous delegation mechanisms supported in different clouds, and the large number of *dele-settings* in the real world – for example, the delegations from SmartThings to Google Home (Flaw 1), IFTTT to SmartThings (Flaw 2), and LIFX to SmartThings (Flaw 3), are all different *dele-settings* – manually modeling each *dele-setting* needs substantial human efforts. To address this scalability problem, *model generator* leverages an observation: IoT clouds' delegation mechanisms are often comprised of common, basic types of operations, such as issuing/revoking a token, giving/removing access to a user, API requests that come with data exchange, etc. Hence, to generate a model for a specific *dele-setting*, *model generator* leverages a general base model and extends it by adding actors, and delegation operations supported in the *dele-setting*.

Specifically, the base model includes a minimum set of actors (i.e., two devices, a delegator cloud, a delegatee cloud, and a user), and delegation operations that trigger the state transitions. As mentioned earlier (Section 4.1), the base model is specified in the Promela language: each actor is a variable and has a corresponding dataset (including token set and ACL set, see Def. 1); each operation is specified as a function that can be called on the actor variable(s), which incurs changes to the dataset of the actors. The extending process is facilitated by adding basic types of operations to the base model, which are all modeled (one-time effort) and stored as template functions in our *delegation operation templates database*: it includes the basic delegation operations (see Table 1) summarized from 10 mainstream IoT clouds (see Section 5). Note that, the same operation in two *dele-settings*, e.g., `share` (denoting that the cloud delegates an access to the user, see Table 1), may incur different dataset changes to the actors: in one *share* operation, the cloud actor may issue a new token to the user, and in the other the cloud actor may also pass along an existing token obtained from its delegator cloud to the user (see Flaw 3). In this case, our *template database* keeps the two different *share* operations respectively (as different template functions), recorded as sub-types of `share`.

Also, as mentioned earlier, the list of actors and operations to add to the base model are specified in the `configuration` file of a *dele-setting*. The `configuration` file is built manually after inspection of those clouds' delegation operations. Specifically, we look at the delegation operations supported by the clouds, and understand the data flows incurred by each operation (e.g., issuing a token to the user). This is done by reading their developer documentations, user manuals, and inspecting the network traffic of their mobile apps, etc.

4.3 Detecting Flaws

Formal verification. With the generated models specified in Promela, we leverage an off-the-shelf model checker Spin [38] to verify the models on a generalized security property, which we elaborate as follows.

Def. 4. **Access Path**: An **access path** from a_j to a_m is an ordered sequence of actors, $v = (a_j, a_1, a_2, ..., a_n, a_m)$, along which a_j can reach a_m, if either $n > 0$ and $F(a_j, a_m, v) \neq \varnothing$ (see definition below), or $n = 0$ and $Recv_{a_j} \cap Issu_{a_m} \neq \varnothing$.

$$\begin{cases} f(K, ACL_{a_i}) = \bigcup\limits_{token \in K, (token, T) \in ACL_{a_i}} \{ t \mid t \in T \} \\ F(a_j, a_1, v) = f(\ Recv_{a_j}, \ ACL_{a_1}) \\ F(a_j, a_k, v) = f(\ F(a_j, a_{k-1}, v), \ ACL_{a_k}), \ (2 \leq k \leq n) \\ F(a_j, a_m, v) = F(a_j, a_n, v) \cap Issu_{a_m} \end{cases}$$

Intuitively, an access path allows a user to access a target device through the delegation of other actors on the path; in a special case, $n = 0$ means the user can access the target device directly using his token issued by the device. Taking the SmartThings-LIFX example above (Section 4.2), a user (a_j), with the token $token_{smartthings}$, is allowed to access a_m (a LIFX bulb), along the path through two actors (the SmartThings cloud and the LIFX cloud), given the tokens they issue ($token_{smartthings}$ and $token_{lifx}$) and their ACL sets.

To validate whether a_j to a_m has an access path v, we leverage $F(a_j, a_m, v)$: intuitively, $F(a_j, a_m, v)$ yields the set of access rights that are delegated out by a_m, through other actors on the path and finally delegated to a_j (represented by $Recv_{a_j}$); $F(a_j, a_m, v) \neq \varnothing$ means a_j has at least one access right (represented by $Recv_{a_j}$) that allows to access a_m. Also note that, $f(K, ACL_{a_i})$ yields, given a set of tokens K, the access rights of a_i that are delegated to the receiver of K. Taking the SmartThings example again, $f(\{token_{smartthings}\}, ACL_{a_{smartthings}})$ yields a set $\{token_{lifx}\}$~~,~~ that represents the right of SmartThings to access the bulb, which is delegated by SmartThings to the user receiving the token set $\{token_{smartthings}\}$.

Def. 5. **Security Property**: *Given a device a_m, a user a_j should not find an access path that lets him access the device that he is not entitled to.* The property can be specified as: $Recv_{a_j} \cap Issu_{a_m} = \varnothing$, and $\forall\ v \in \{\ (a_j, a_1, a_2, ..., a_n, a_m) \mid a_k \in \mathcal{A} - \{a_j, a_m\}, 1 \leq k \leq n,\ n > 0\ \},\ F(a_j,\ a_m, v) = \varnothing$.

With respect to the property, our model checker reports a counterexample (security property violation) if it can find an access path across actors in the system that enables an unauthorized user actor to access the device actor. Such a detection is performed when certain state transitions occur in the system, e.g., once a user's access is revoked, the checker verifies whether he can still access the device.

Analyzing the counterexamples. Fully automated validation of reported counterexample on real-world IoT clouds is nontrivial, since the end-to-end validation requires one to set up devices under those clouds, register user accounts, and perform delegation operations on the clouds' management consoles, mobile apps, and even through physically touching the device, etc. So we manually validated the reported counterexamples by performing proof-of-concept end-to-end exploits on corresponding clouds (see PoC exploits in Section 3).

4.4 Implementation of VerioT

We provide implementation details of `VerioT` in this section; its full source code and a video demo on its usage are released online [34].

• *The Delegate Operation Templates Database.* We inspected delegation operations supported on 10 mainstream IoT clouds (SmartThings, IFTTT, Google Home, Wink, Amazon Alexa, Philips Hue, LIFX, August, MiHome, iHome) and generalized nine basic types of operations, such as `OAuth`, `share` (see Table 1). Each operation also incurs data changes to the actors in the states, e.g., `OAuth` operation performed by a delegator will generate a token, held by both delegator and delegatee actors in their storage. In our implementation, each basic operation is represented as a template function using Promela language. Different sub-types of an operation (see Section 4.2) has separate template functions, indexed in the *Templates Database* by the operation name and template number. All template functions in the *templates database* are released online [25].

• *The Configuration file and the Model Generator*. The `Configuration` file lists actors in the *dele-setting*, operations supported by the actors (with reference to the operations' template code in the *database*), and optionally lists more actors (e.g., additional clouds and devices) that the *dele-setting* involves but missing in the base model. An example configuration (for the *dele-setting* of LIFX and SmartThings) is illustrated in Figure 9. It lists five actors (Line 1-6), and delegation operations supported by each actor (Line 8-17), for example, delegatee cloud LIFX can perform *share* operation with the user, whose template function is referenced in the *templates database* (by `share_template:2`). The *model generator* takes the `Configuration` file as input, constructs actors and their storage (token set and ACL set, see Def. 1) in Promela language; then based on the operations each actor supports, pulls template functions from the *templates database* to generate a model, represented in Promela code. The *generator* is implemented in Python with 1,000 lines of source code. The generated models of each *dele-setting* in our research (in Promela code) and all configuration files used to generate the models are released online [40].

```
 1 actors:
 2 device: LIFX_bulb
 3 device_2: LIFX_bulb_2
 4 delegator_cloud: LIFX_cloud
 5 delegatee_cloud: SmartThings_cloud
 6 user: SmartThings_user
 7
 8 supported_operations:
 9 operation:bind bind_template:1  actors:device,delegator_cloud
10 operation:bind bind_template:1  actors:device_2,delegator_cloud
   ...
17 operation:share share_template:2 actors:delegatee_cloud,user
```

Figure 9: `Configuration` file example

4.5 Results and Discussion

We applied `VerioT` to assess the security of 10 mainstream IoT clouds (see Section 5) for their cross-cloud delegation, and identified 6 new delegation flaws (including flaws of MiHome and Wink in Section 5, Flaw 1-3 and 5 in Section 3) affecting all the ten clouds. We manually confirmed all the flaws, and implemented end-to-end PoC attacks using real devices of ours for five of the flaws. The measurement details on these flaws are presented in Section 5.

Discussion of limitation and coverage. As mentioned earlier (Section 4.2), constructing a `Configuration` file in our study involves manual efforts to understand the target system, i.e., delegation operations it supports and data set changes they incur (e.g., token set, see Def. 1). To this end, we read their developer documentation and user manuals to learn such information; based on the devices we have, we also manually performed those delegations operations, and monitored the network traffic of the companion mobile apps (for each *dele-setting* in Section 3, we used a few devices discussed in the PoC attacks). The construction of a `Configuration` took 5 to 30 hours in our study (based on the length of the documentation and the number of supported delegation operations in the *dele-setting*).

In the absence of a standardized and well specified delegation protocol for IoT clouds, we may not know all information of a particular cross-cloud delegation system (all delegation operations it supports, all data/token flows incurred by the delegation operations, and internal access list the cloud maintains, etc.). Similar to the prior work [55], our strategy is to start with a simple model, and introduce additional complexity into the model if no counterexamples are found. Specifically, we progressively add more operations and their corresponding data flows to the model, along the way we understand the corresponding system's operations, until the model is complete enough to report a flaw.

Also, the access management and delegation-related operations on real-world IoT cloud systems can be very complex. Hence, in some cases we need to abstract the real-world systems so as to start with a relatively simple model, before we can progressively enrich the model to better approximate the real systems. In particular, we focus on the delegation operations and look for all possible avenues where tokens can be transferred and shared between actors (e.g., programmatic Web API calls and manual Web console access – both abstracted and modeled as `APIRequest` operation), and ignore complex usage contexts (e.g., whether it is programmatic or manual Web access, SmartThings' *location*, etc.). Let us take Flaw 1 as an example. Although SmartThings device ID is a security token only under certain usage context (i.e., it is a security token in trigger-action based device access, but not in direct device access, see Section 3.1), in our model of SmartThings-Google Home *dele-setting*, we made an assumption to ignore the usage contexts and simply considered SmartThings device ID as a security token that can be used to access the device. Further, as we studied Google Home documentation and looked at the network traffic of Google Home mobile app, we learnt that Google Home cloud transferred the SmartThings device ID to the user-end app when sharing with him the access to the SmartThings device; correspondingly, in our modeling, the user actor will add the SmartThings device ID to his token set once Google Home performs a `share` operation (see Table 1) with the user actor. Based on the model, when our checker Spin enumerates all possible states by running different delegation operations (implemented as functions in Promela language), it found that the user actor had an access path to the SmartThings device (via SmartThings cloud) based on the device ID he held in his token set, even after Google Home performed an `un-share` operation to revoke the user's access to the device. Note that, in our modeling of the `un-share` operation, Google Home will only revoke any token it generated and shared with the user, but will not revoke the SmartThings device ID – this is because we did not find any APIs or mechanisms provided by SmartThings for doing so, based on public documentations. Last, as mentioned earlier, the bug found in the verification process was then confirmed through PoC experiments using our real devices.

In the absence of a standardized IoT delegation protocol, although we may not have full information of a particular cross-cloud delegation system (all delegation operations it supports, its internal access management, etc.), our approach has demonstrated its feasibility in identifying delegation weaknesses with public information of those systems. Also, real-world IoT vendors, with full information about their delegation protocols and operations, can use our approach and tool to verify their systems.

Last, VerioT facilitates automatic search on all possible states in the model, under the constraints of the search depth set for SPIN, 20,000 in our experiment. Note that a delegation system with just a few actors can have hundreds or even thousands of states, considering the operations in different orders among the actors, which are hard to inspect manually.

5 Measurement

5.1 Prevalence of Vulnerable Delegation

With the help of `VerioT`, we evaluated the security risks in access delegation of 10 mainstream IoT clouds, including both *device vendor cloud*s – delegator clouds – and *delegatee clouds*.

Device vendor clouds. We evaluated 5 mainstream device vendor clouds, Philips Hue [26], August [8], LIFX [17], MiHome [21], and iHome [14], who collectively have millions of users worldwide [9, 15, 18, 22, 29]. It turned out all these clouds are either vulnerable themselves, or delegate device access to a vulnerable delegatee cloud (e.g., Google Home, SmartThings, etc., see below).

Of particular concern observed here is that those clouds typically developed their customized, often ad-hoc delegation management, which highlights the heterogeneous, problematic IoT delegation ecosystem in the absence of a standard, secure delegation protocol. Flaw 5 is one example of such. As another example (Flaw 6), we found MiHome cloud delegates two tokens – one token generated by the cloud and one secret string generated by the device – to its delegatee user; when

MiHome revokes the delegatee user's access, it invalids the token on the cloud, but does not inform the device to invalidate the secret string. Through our end-to-end experiment using our own device, we found this flaw introduces security risks in the real world scenario: even after the delegatee user's access is revoked (e.g., after an Airbnb guest checks out), as long as he can still connect to the local network where the device connects to (e.g., by going close to the Airbnb house), he can use the secret string as a token to command the device. Such a device under MiHome cloud can be a door lock, that can let the unauthorized user enter the house. Further, `VerioT` did not report any flaws on August or iHome clouds. However, their devices can be registered to SmartThings, and thereby potentially affected by Flaw 1.

Delegatee clouds. `VerioT` also helped us evaluate popular delegatee clouds, Google Home [12], Samsung SmartThings [33], IFTTT [13], Amazon Alexa [7], and Wink [42]. It turned out that all of them are affected by insecure delegation management. Specifically, in addition to Flaw 1-3 that indicate design faults of Google Home, SmartThings, and IFTTT, we found another flaw in Wink cloud (Flaw 7), who is also confused with its delegator clouds' security policies and unwittingly leak their device IDs to untrusted delegatee users. This presents a risk similar to Flaw 1. Further, albeit `VerioT` did not report a flaw on Amazon Alexa, it is affected by Flaw 3: Alexa supports to delegate access to SmartThings cloud, which was found to leak the delegator's token.

5.2 Scope of Impact

In our study, we also measured the scope of the impacts by major security problems discussed in Section 3.

IoT clouds affected by Flaw 1. In Flaw 1, Google Home discloses device ID of its delegator cloud (i.e., SmartThings), and an unauthorized delegatee user can leverage the obtained device ID to impersonate device events and unlock a smart door on SmartThings. With this flaw, any delegator of Google Home is affected if device ID on its cloud serves as a secret token. To better understand the scope of affected delegator clouds, we manually inspected nine delegator clouds and found that three of them use device ID as a secret token: SmartThings, TP-Link Kasa and elinkSmart (names of the nine clouds are released online [2]). We launched end-to-end attack against SmartThings (see PoC exploit on Flaw 1). For other two clouds, through inspecting their documentations and prior works [52, 76] that specified the functionality of their device IDs, the problem is also alarming: an attacker may leverage their device IDs to send fake device events on behalf of the device, and trigger other sensitive devices (e.g., door locks), based on trigger rules on the two clouds.

IoT clouds affected by Flaw 2. With Flaw 2 (Section 3.1), all vendors that delegate access to IFTTT are potentially affected. We illustrate 34 IoT vendors (names released online [2]) that delegate access to IFTTT, whose products/services range from smart lights (e.g., LIFX) to home security devices (e.g., Arlo).

IoT clouds affected by Flaw 3. In Flaw 3 (Section 3.2), SmartThings cloud leaks the credential (e.g., OAuth token) stored by its delegator clouds in their SmartApps. Therefore, any delegator cloud that stores sensitive information/token in its SmartApp is affected by Flaw 3. From 127 devices vendors that delegate access to SmartThings (see the list released online [2]), we manually reviewed their SmartApps that are open-source (on SmartThings' official Github repository [36]), and found that 18 SmartApps (see the full list online [2]) store sensitive information (e.g., OAuth token, authentication token, secret callback URL, etc). That is, 18 correspondingly delegator clouds of SmartThings are potentially affected by Flaw 3.

IoT clouds affected by Flaw 4. In Flaw 4, Tuya cloud introduced a security risk in applying OAuth to cross-cloud IoT delegation. Interestingly, we found that this single flaw affected many IoT vendors. Specifically, Tuya not only manufactures IoT devices itself, but also provides its IoT cloud services to other device vendors, who do not own a cloud themselves. That is, Tuya cloud serves as *device vendor cloud* for devices manufactured by many other vendors. Interestingly, given such a paradigm, all those vendors can be affected by Flaw 1 on Tuya cloud (see a list of 58 affected IoT vendors online [2]).

Conflicting security policies across clouds. As shown in Section 3.1, different clouds have conflicting security policies and may not have an effective mechanism to coordinate their security assumptions and operations. To better understand the scope of the problem, we inspected the developer documentations of popular delegatee clouds including Google Home [11], Alexa [6] and Wink [43]. We found that, to offer cross-cloud delegation services, they all ask their delegator clouds to provide device information, including device ID, name, model, version and type, in the delegation process. However, based on available information, none of them communicated their security assumptions with delegator clouds: they did not describe how they would handle the data (e.g., Google Home exposed the device ID and caused Flaw 1), or requested information from their delegators to confirm whether the data are security-sensitive. Such a finding further suggests the general lack of coordinated security management across IoT clouds.

6 Discussion and Future Work

Lessons learnt. The most important lesson learnt from our research is the caution one should take when applying a custom cross-cloud authorization scheme to today's already complicated IoT delegation. In the absence of a standardized, fully verified cross-cloud delegation protocol, there is no guarantee that the new mechanism would not inadvertently bring in new security flaws, in policy setting or enforcement. To be more specific, without fully understanding other parties' security constraints or adequately informing other parties of their own security expectations, there is a risk that the delegator and the delegatee violate each other's security policies. An equally common risk in IoT delegation is problematic security policy enforcement due to lack of rigorous verification.

The risks reported in this paper affect scenarios of IoT access delegation, which are common today. For example, an Airbnb host often needs to delegate the access to her door locks to Airbnb guests (for them to access the property) and revoke the access later. Convenient delegation of (the access to) smart locks to Airbnb guests during their reservation period has been a prominent feature advocated by both Airbnb and mainstream IoT vendors [3, 4, 5], including lock manufacturers August, Remotelock and AURMUR, etc.

New design principles. To avoid such risks, we propose three principles for developing the delegation mechanism for individual IoT clouds, before a consensus can be reached on a standardized solution:

• *Communicating security assumptions and constraints*. Inadequate coordination of security requirements cross the clouds is one of the major causes for security hazards found in the heterogeneous IoT delegation. The problem can be addressed by establishing a channel between clouds to exchange their security constraints and assumptions. In particular, all clouds in a delegation should coordinate their security policies: e.g., when one cloud discloses tokens/data shared with or obtained from another cloud, it should be given (by the latter) the security implications in doing so. To this end, a formal description of such information can be helpful. This coordination effort can also lay the foundation for the effort to standardize cross-cloud IoT delegation.

• *Decoupling the delegatee and the delegator clouds*. As shown in our research, real-world IoT clouds have developed heterogeneous and ad-hoc delegation protocols, which made IoT clouds hard to decouple from each other and get tangled in others' access management: for example, IFTTT runs its SmartApps on SmartThings to help the latter manage the access to IFTTT devices, but gets into a position that can inadvertently violate the latter's access policy. Over the longer term, we envision that an IoT delegation protocol should be standardized and verified, with necessary security requirements and practice fully defined. This addresses the fundamental cause of the flaws reported in the paper.

• *Verifying delegation design whenever possible*. As demonstrated by our research, formal verification of a real-world delegation mechanism can help reduce security risks. The security property violations uncovered by a verification tool tuned for IoT like `VerioT` can help vendors identify weaknesses in security policies and inadequate policy enforcement, and thus lead to more secure IoT delegation ecosystem. Our `VerioT` made a first step towards this end and further effort needs to make to improve its efficacy.

Compositional verification. Modeling and verifying real-world IoT delegation systems with many actors and operations is complicated. `VerioT` makes a first attempt towards this end, though the model it verified is relatively small, typically involving two clouds and their supported operations. Analysis of larger models needs compositional verification, which however cannot be provided by Spin, the off-the-shelf model checker used in `VerioT`. Other tools with the composition capability [60, 68] could potentially help us analyze more complicated models. Enhancement of our technique with these tools is left to our future research.

Automated vulnerability detection. Based on our understanding of the security risks in cross-cloud IoT access delegation, we believe that more automatic security analysis and vulnerability discovery are feasible. With the help of `VerioT`, we were able to find the vulnerabilities reported in the paper in a semi-automatic way. The manual effort in our current design was made to capture the control flow and data flow of a delegation process to build its state machine, which includes manual inspection of developer documentation, analysis of communication traffic on the related mobile app, etc. The knowledge discovery part (inspecting documents) could be automated using Natural Language Processing (NLP), as did in the prior research on security analysis of payment services [53]. We envision that significant effort will be seen on this subject.

7 Related Work

IoT platform security. Many works have been done to analyze security problems of the IoT Cloud, considering the important role it plays. [56] first reported the coarse-grained capability design and the insufficient protection in event subsystem of SmartThings platform. [76] and [52] found flaws in device management of IoT clouds, which both revealed the serious consequences of leaking device identity. [74] presented a system to discover inter-rule vulnerabilities on IFTTT. In contrast, our work attempts to understand the security risks in cross-cloud IoT operations instead of identifying the flaws on a single IoT cloud. Meanwhile, extensive works [46, 50, 51, 66, 72, 75] are proposed to protect IoT systems. For example, [46, 50, 72] and [58] provided methods on protecting the information/data flow. As for permission protection, [66] proposed a fine-grained context-based permission system. In contrast, we present a semi-automatic verification tool (`VerioT`) to conduct the first security analysis on the cross-cloud IoT access delegation process.

Permission delegation in IoT. Delegable authorization has been well researched in the literature [47, 48, 49, 67, 70]. Unlike the theoretic models and expressive languages analyzed before, access control on today's IoT clouds is not only distributed but also heterogeneous and ad-hoc. To cope with new application scenario, [59] introduced *Decentralized Action Integrity* to prevent an untrusted trigger-action platform from misusing OAuth tokens.[45] presented WAVE, an authorization framework offering fully *decentralized trust*, which supports decentralized verification, transitive delegation and revocation, etc. WAVE fulfills the requirements of today's complicated IoT delegation and, however, requires all parties (different vendors) collaborate together following the same framework API. How to deploy a cryptographically ideal authorization framework (requiring storage servers, auditors, cryptographic functions) to real-world, heterogeneous IoT environments that include diverse, complicated applications (e.g., automation control, trigger-action service), extremely large number of devices, and even devices with extremely low computing power (e.g., sensors) is not clear. On the contrary, we conducted a systematic analysis of today's off-the-shelf IoT delegation and obtained in-depth understanding of its security risks in real-world IoT systems. Due to the absence of standardized delegation protocol and a long time needed to deploy a standardized, secure, efficient, effective protocol, our work can lead to better understanding of today's IoT applications and provide valuable insights towards standardizing a practical protocol.

Model-based vulnerability discovery. Prior works have attempted to automatically find vulnerabilities using fuzzing, symbolic execution, formal verification, etc. [44, 54, 55, 63, 64, 65, 71]. [64] used a model-based approach to automatically discover the attacks in TCP congestion control, and [55] identified new forms of idle port scan attack with the help of model checking. [44] presented SmartVerif, a novel and general framework that leverages dynamic strategy to smartly search proof paths without human intervention. Most of these work focused on vulnerability discovery in a single system or protocol, while our work leveraged formal verification to (semi-)automate the security flaws discovered in the IoT access delegation, which involves multiple parties, different protocols and heterogeneous systems.

8 CONCLUSION

We performed the first systematic study on security risks in the cross-cloud IoT access delegation. We proposed a semi-automatic verification tool to conduct an extensive investigation of 10 leading IoT clouds. Our research reveals new security vulnerabilities in IoT access delegation that are pervasive in IoT clouds. Our findings suggest that the heterogeneous and ad-hoc delegation process is the root cause for such security flaws. Based on our new understanding on cross-cloud IoT access delegation, we proposed new generalized design principles for mitigation. Our new findings and understanding will lead to better protection of today's IoT applications and provide valuable insights towards securing IoT systems.

Acknowledgment

We would like to thank our shepherd Dr. Zakir Durumeric and the anonymous reviewers for their insightful comments. Bin Yuan, Deqing Zou and Hai Jin are supported by the National Natural Science Foundation of China (No. 61902138), the China Postdoctoral Science Foundation funded Project (No. 2018M640701), the National Key Research and Development Plan of China (No. 2017YFB0802205), the Key-Area Research and Development Program of Guangdong Province (No. 2019B010139001) and the Shenzhen Fundamental Research Program (No. JCYJ20170413114215614). Yan Jia and Yuqing Zhang are supported by the National Key R&D Program China (No. 2018YFB0804701), the National Natural Science Foundation of China (No.U1836210, No.61572460) and in part by China Scholarship Council. The authors of Indiana University are supported in part by Indiana University FRSP-SF, and NSF CNS-1618493, 1801432, and 1838083.

References

[1] Actions on Google. https://developers.google.com/assistant/smarthome/develop/process-intents. Accessed: 2019-11.

[2] Affected Vendors. https://sites.google.com/view/shattered-chain-of-trust-under/home/affected-vendors?authuser=0s. Accessed: 2020-05.

[3] Airbnb API. https://www.airbnb.com/partner. Accessed: 2020-05.

[4] Airbnb Integration with August. https://support.august.com/airbnb-integration-faq-B1YAULkC_z. Accessed: 2020-05.

[5] Airbnb Smart Locks . https://www.postscapes.com/airbnb-smart-lock/. Accessed: 2020-05.

[6] Alexa.discovery interface. https://developer.amazon.com/docs/device-apis/alexa-discovery.html. Accessed: 2019-11.

[7] Amazon Alexa. https://developer.amazon.com/en-US/alexa. Accessed: 2019-11.

[8] August. https://august.com/products/august-smart-lock-pro-connect. Accessed: 2019-11.

[9] August Installs. https://play.google.com/store/apps/details?id=com.august.luna. Accessed: 2019-11.

[10] Get SmartApp Endpoints. https://docs.smartthings.com/en/latest/capabilities-reference.html. Accessed: 2019-11.

[11] Google assistant-process intents. https://developers.google.com/assistant/smarthome/develop/process-intents#sync-response. Accessed: 2019-11.

[12] Google Home. https://developers.google.com/assistant/smarthome/overview. Accessed: 2019-11.

[13] IFTTT. https://ifttt.com/. Accessed: 2019-11.

[14] ihome. https://www.ihomeaudio.com. Accessed: 2019-11.

[15] iHome Installs. https://play.google.com/store/apps/details?id=com.sdi.ihomecontrol. Accessed: 2019-11.

[16] KEYGMA. http://www.keygma.com/en/index.html. Accessed: 2020-05.

[17] LIFX. https://www.lifx.com/. Accessed: 2019-11.

[18] LIFX Installs. https://play.google.com/store/apps/details?id=com.lifx.lifx. Accessed: 2019-11.

[19] LIFX List Lights API. https://api.developer.lifx.com/docs/list-lights. Accessed: 2019-11.

[20] LIFX Set State API. https://api.developer.lifx.com/docs/set-state. Accessed: 2019-11.

[21] Mihome. https://xiaomi-mi.com/mi-smart-home. Accessed: 2019-11.

[22] MiHome Installs. https://play.google.com/store/apps/details?id=com.xiaomi.smarthome. Accessed: 2019-11.

[23] Mirai Attacks. https://goo.gl/QVv89r. Accessed: 2019-11.

[24] OAuth 2.0. https://oauth.net/2/. Accessed: 2019-11.

[25] Operation Templates in VerioT. https://github.com/

VerioT/VerioT/tree/master/templates. Accessed: 2019-11.

[26] Philips HUE. https://www2.meethue.com/. Accessed: 2019-11.

[27] Philips Hue: Add new Remote Hue API app. https://developers.meethue.com/add-new-hue-remote-api-app/. Accessed: 2019-11.

[28] Philips HUE API. https://developers.meethue.com/develop/hue-api/. Accessed: 2019-11.

[29] Philips Hue Installs. https://play.google.com/store/apps/details?id=com.philips.lighting.hue2. Accessed: 2019-11.

[30] Philips Hue: Permission revocation. https://www2.meethue.com/en-us/support/apps-and-software#How_do_I_remove_an_unused_smart_device_from_my_Philips_Hue_system. Accessed: 2019-11.

[31] Philips Hue Remote Control API. https://developers.meethue.com/develop/hue-api/remote-api-quick-start-guide. Accessed: 2019-11.

[32] Promela. https://en.wikipedia.org/wiki/Promela. Accessed: 2019-11.

[33] Samsung SmartThings. https://www.smartthings.com/. Accessed: 2019-11.

[34] Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation. https://sites.google.com/view/shattered-chain-of-trust-under/home?authuser=1.

[35] SmartApps. https://docs.smartthings.com/en/latest/smartapp-developers-guide/. Accessed: 2019-11.

[36] SmartThings Git. https://github.com/SmartThingsCommunity/SmartThingsPublic. Accessed: 2019-11.

[37] SmartThings Groovy IDE. https://graph.api.smartthings.com/. Accessed: 2019-11.

[38] Spin. http://spinroot.com/spin/whatispin.html. Accessed: 2019-11.

[39] The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749. Accessed: 2019-11.

[40] VerioT Examples. https://github.com/VerioT/VerioT. Accessed: 2019-11.

[41] Web Services SmartApps. https://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/index.html#. Accessed: 2019-11.

[42] Wink. https://www.wink.com/. Accessed: 2019-11.

[43] Wink api. https://winkapiv2.docs.apiary.io/#reference/device. Accessed: 2019-11.

[44] Smartverif: Push the limit of automation capability of verifying security protocols by dynamic strategies. In *29th USENIX Security Symposium*, 2020.

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

[46] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If this then what?: Controlling flows in iot apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1102–1119. ACM, 2018.

[47] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. Trust negotiations: concepts, systems, and languages. *Computing in Science & Engineering*, 6(4), 2004.

[48] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *21st Annual Network and Distributed System Security Symposium*, 2014.

[49] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[50] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1687–1704, 2018.

[51] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *NDSS*, 2019.

[52] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. Your iots are (not) mine: On the remote binding between iot devices and users. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 222–233, 2019.

[53] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *28th USENIX Security Symposium*, pages 747–764, 2019.

[54] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium*, 2011.

[55] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle port scanning and non-interference analysis of network protocol stacks using model checking. In *19th USENIX Security Symposium*, pages 257–272, 2010.

[56] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *37th IEEE Symposium on Security and Privacy*, pages 636–654, 2016.

[57] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium,*, pages 531–548, 2016.

[58] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 531–548, 2016.

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

[60] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424, pages 292–307, 2007.

[61] Weili Han, Qun Ni, and Hong Chen. Apply measurable risk to strengthen security of a role-based delegation supporting workflow system. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 45–52, 2009.

[62] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David A. Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *11th ACM Asia Conference on Computer and Communications Security*, pages 461–472, 2016.

[63] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. BEADS: automated attack discovery in openflow-based SDN systems. In *20th International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 311–333, 2017.

[64] Samuel Jero, Md. Endadul Hoque, David R. Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *25th Annual Network and Distributed System Security Symposium*, 2018.

[65] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2015.

[66] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati,

Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. Contexlot: Towards providing contextual integrity to appified iot platforms. In *NDSS*, 2017.

[67] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[68] Corina S. Pasareanu and Dimitra Giannakopoulou. Towards a compositional SPIN. In *the 13th International Workshop on Model Checking Software*, volume 3925, pages 234–251, 2006.

[69] Yoshiki Sameshima and Peter T. Kirstein. Authorization with security attributes and privilege delegation: Access control beyond the ACL. *Computer Communications*, 20(5):376–384, 1997.

[70] Kent E. Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobson, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks*, pages 68–79, 2002.

[71] JaeSeung Song, Cristian Cadar, and Peter R. Pietzuch. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Trans. Software Eng.*, 40(7):695–709, 2014.

[72] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 361–378, 2017.

[73] Jacques Wainer, Akhil Kumar, and Paulo Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, 2007.

[74] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1439–1453. ACM, 2019.

[75] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*, 2018.

[76] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium*, pages 1133–1150, 2019.

Appendix

A. Definitions of the delegation operations

We generalized nine basic types of delegation operations (see Table 1) and constructed their operation templates in Promela language (the *Templates Database* is released online [25]). Note that, one basic type of delegation operation may have a few sub-types (see Section 4.2), and correspondingly a few operation templates in the *Templates Database*. We define each basic type of delegation operation as follows.

bind: device a_i issues a new token for cloud a_j.

`bind`(a_i, a_j) is defined as:

$$\begin{cases} token := newUniqueToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \end{cases}$$

unbind: device a_i removes all the issued tokens.

`unbind`(a_i) is defined as: $Issu_{a_i} := \varnothing$

share: cloud a_i delegates access right T to user a_j by issuing a new token and sharing the tokens the cloud received to the user.

`share`(a_i, a_j, T) is defined as:

$$\begin{cases} token := newUniqueToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \cup Recv_{a_i} \\ ACL_{a_i} := ACL_{a_i} \cup \{(token, T)\} \end{cases}$$

un-share: cloud a_i revokes the access right from user a_j by invalidating the token *token*.

`un-share`$(a_i, token)$ is defined as:

$$\begin{cases} Issu_{a_i} := Issu_{a_i} - \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} - \{(token, T) \mid (token, T) \in ACL_{a_i}\} \end{cases}$$

OAuth: cloud (a_i) delegates cloud (a_j) access right T by issuing a new token.

`OAuth`(a_i, a_j, T) is defined as:

$$\begin{cases} token := newOAuthToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} \cup \{\ (token, T)\ \} \end{cases}$$

un-OAuth: cloud (a_i) revokes access right from cloud (a_j) by revoking token *token*.

`un-OAuth`$(a_i, token)$ is defined as:

$$\begin{cases} Issu_{a_i} := Issu_{a_i} - \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} - \{\ (token, T) \mid (token, T) \in ACL_{a_i}\} \end{cases}$$

setTrigger: cloud (a_i) delegates cloud (a_j) access right T by issuing a new token and obtains read access right to the devices that cloud (a_j) has access to.

`setTrigger`(a_i, a_j, T) is defined as:

$$\begin{cases} token := newUniqueToken() \\ Issu_{a_i} := Issu_{a_i} \cup \{\ token\ \} \\ Recv_{a_j} := Recv_{a_j} \cup \{\ token\ \} \\ ACL_{a_i} := ACL_{a_i} \cup \{\ (token, T)\ \} \\ Recv_{a_i} := Recv_{a_i} \cup \{\ a_k \mid Recv_{a_j} \cap Issu_{a_k} \neq \varnothing, ACL_{a_k} = \varnothing\} \end{cases}$$

un-setTrigger: cloud (a_i) does nothing to disable the its triggers.

`un-setTrigger`(a_i) is defined as: *None*

APIRequest: user (a_i) makes API request to cloud (a_j) to obtain all the tokens a_j receives.

`APIRequest`(a_i, a_j) is defined as:

$$Recv_{a_i} := Recv_{a_i} \cup Recv_{a_j}$$