P-Verifier: Understanding and Mitigating Security Risks in Cloud-based IoT Access Policies

Ze Jin*
Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Indiana University Bloomington

Luyi Xing*†
Indiana University Bloomington

Yiwei Fang
Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences; Indiana University Bloomington

Yan Jia
College of Cyber Science, Nankai University

Bin Yuan
School of Cyber Science and Engineering, Huazhong University of Science and Technology

Qixu Liu†
Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences

ABSTRACT

Modern IoT device manufacturers are taking advantage of the managed Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) IoT clouds (e.g., AWS IoT, Azure IoT) for secure and convenient IoT development/deployment. The IoT access control is achieved by manufacturer-specified, cloud-enforced IoT access policies (cloud-standard JSON documents, called *IoT Policies*) stating which users can access which IoT devices/resources under what constraints. In this paper, we performed a systematic study on the security of cloud-based IoT access policies on modern PaaS/IaaS IoT clouds. Our research shows that the complexity in the IoT semantics and enforcement logic of the policies leaves tremendous space for device manufacturers to program a flawed IoT access policy, introducing convoluted logic flaws which are non-trivial to reason about. In addition to challenges/mistakes in the design space, it is astonishing to find that mainstream device manufacturers also generally make critical mistakes in deploying *IoT Policies* thanks to the flexibility offered by PaaS/IaaS clouds and the lack of standard practices for doing so. Our assessment of 36 device manufacturers and 310 open-source IoT projects highlights the pervasiveness and seriousness of the problems, which once exploited, can have serious impacts on IoT users' security, safety, and privacy. To help manufacturers identify and easily fix *IoT Policy* flaws, we introduce *P-Verifier*, a formal verification tool that can automatically verify cloud-based *IoT Policies*. With evaluated high effectiveness and low performance overhead, *P-Verifier* will contribute to elevating security assurance in modern IoT deployments and access control. We responsibly reported all findings to affected vendors and fixes were deployed or on the way.

CCS CONCEPTS

• **Security and privacy** → **Access control**; • **Software and its engineering** → *Formal software verification.*

KEYWORDS

IoT, Formal Verification, Access Control Policy, Cloud.

ACM Reference Format:
Ze Jin, Luyi Xing, Yiwei Fang, Yan Jia, Bin Yuan, and Qixu Liu. 2022. P-Verifier: Understanding and Mitigating Security Risks in Cloud-based IoT Access Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560680

1 INTRODUCTION

The Internet of Things (IoT) cloud is one of the key pillars of the foundation upon which modern IoT systems rest. Newer IoT devices and their manufacturers take advantage of the much-less studied, third-party, managed Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) IoT cloud services (e.g., AWS IoT Core [10], Azure IoT Hub [21], and Tuya IoT Cloud [44]), which offload much of the security responsibilities and deployment burden to the public cloud providers. Such PaaS and/or IaaS IoT clouds (referred to as *IoT clouds* in this paper) must trust-manage hundreds of millions of IoT devices and users, and provide device manufacturers with reliable and usable tools for secure IoT deployments. In the IoT cloud systems, compromised security or improper deployments can cause hazardous and deadly consequences. Despite the importance, the security of managed third-party IoT clouds was not fully understood [6, 52, 58, 83]. In particular, it is imperative to systematically explore whether and to what extent their PaaS and IaaS design and practices effectively help device-manufacturers make secure IoT development and deployment, which will be studied in this paper.

Cloud-based access policies and IoT access policies. In general, the convenience of accessing resources in the cloud is made secure by developer-specified access control policies. A cloud-based *access policy* is an expressive specification of what resources can be accessed, using what actions (e.g., read/write/create), by whom, and

*The first two authors Ze Jin and Luyi Xing are ordered alphabetically.
†Corresponding authors: Luyi Xing, Qixu Liu.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560680

under what conditions. Cloud providers such as AWS and Azure generally define a policy language (e.g., in the JSON format, called IAM policies across all AWS services [37]) that lets developers govern access to resources in the cloud. A cloud-based IoT access policy (or *IoT Policy* for short), taking AWS IoT as an example, inherits the syntax of the service-agnostic AWS IAM policies and is an extension of IAM policy in the IoT context. An *IoT Policy* specifies which users/clients can access which IoT devices/resources under what constraints and can specify action/resource types in the IoT context (e.g., *publish* an MQTT message — a type of action from the popular IoT messaging protocol, see § 2).

Prior works [53, 54, 61, 108] studied the misconfiguration of cloud-based access policies, in particular focusing on AWS IAM policies [53, 54, 61]. In the meantime, the *IoT Policies* introduce new challenges due to the IoT-unique, complicated semantics, syntax, and constraints (see § 4) and, thus, could not be effectively analyzed by prior cloud-based policy analysis tools (see the comparison in § 5.2).

Security analyses and attacks. To understand the error space and real-world impact of flawed *IoT Policies*, we systematically studied *IoT Policies* developed by mainstream device manufacturers on AWS IoT. By analyzing the policy designs/practices of 36 mainstream device manufacturers (e.g., Onelink, Belkin, Govee, Netvue) and 310 open-source IoT projects (on GitHub [30]), our study brings to light the fundamental design challenges and tremendous error space for device manufacturers to develop secure *IoT Policies* (see below). The design challenges/flaws we found also highlight that PaaS and IaaS IoT cloud providers (e.g., AWS, Azure) generally failed to provide an IoT infrastructure with necessary tools that easily, and reliably helps manufacturers secure modern IoT development and deployment. Our analysis is guided by the theories of strings and automata [8] and partially automated by *P-Verifier*, a formal verification tool to verify cloud-based IoT access policies (see below).

Regarding the design-space challenges, above all, we found that the semantic gap between IoT contexts and the cloud-general policy language (e.g., AWS IAM [37]) makes the development of *IoT Policy* extremely error-prone (§ 3.1). In particular, the AWS-wide policy enforcement mechanism can be abstracted as an automaton model (a finite state acceptor for strings [47]): each policy essentially defines a string-acceptor *sa* which decides, given an input string *res* that describes a resource to access (e.g., an S3 file path, an IoT device/topic, see § 2), whether or not *res* is accepted by the *sa*, and, thus, the access is allowed. Although such a general enforcement model has long been successful in cloud computing, we find that it fails to soundly restrict IoT resources whose semantics are more complicated than common resources in cloud computing. Specifically, in IoT, one string in a request can refer to multiple IoT resources in the cloud; one IoT resource can be referred to by multiple strings (§ 4) — we call such multiple strings *IoT synonyms* (or *IS*es for short) of the IoT resource. Without a thorough semantic analysis of *IoT synonyms*, we find that *it is difficult for IoT device manufacturers to soundly specify a string-acceptor-based automaton model (i.e., an IoT Policy) that can fully deny all ISes of an IoT resource to protect. This has led to serious over-privileges in the IoT policies of many real vendors (§ 3.1).*

Further, our research shows that the logical relations between IAM policies and *IoT Policies*, and between AWS IoT's authorization logic and Cognito (the AWS-wide authorization service) are far more complicated than expected. The actual complexity generally failed to be understood and properly handled by real-world IoT vendors/developers, leading to security-critical logic loopholes in cloud-based IoT access-control policy development (§ 3.2). Also alarming are the flawed practices of mainstream device manufacturers in deploying *IoT Policies* (§ 3.3). These mistakes are diverse and non-trivial to avoid in the first place, highlighting the lack of standard, adequate security practices with cloud-based IoT policy deployment.

Measurement of impacts. To understand the impact and pervasiveness of the problems, we studied devices/mobile-apps of 36 IoT manufacturers and 310 open-source AWS-IoT-based projects from GitHub. 13 IoT vendors were confirmed to have 17 instances of *IoT Policy* flaws, affecting at least 3.3 million users. *IoT Policies* in 172/310 open-source projects suffered from the flaws we found. The attack consequences (Appendix Table 1) are serious, impacting safety, security, and privacy. For example, any users can control all *t2Fi* users' smart grills, with serious danger of fire/safety; any users can collect all *Biobeat* users' sensitive health/medical information such as blood pressure, height, weight, and age. We reported all problems to affected vendors and helped them fix the issues.

Logical encoding and automated reasoning for IoT policies. To help manufacturers detect security flaws in *IoT Policies*, we designed and implemented *P-Verifier*, a formal verification tool that can effectively verify cloud-based IoT access policies (§ 4). *P-Verifier* takes *IoT Policies* as input and develops formal models that semantically, fully represent the policies; the models are encoded with Satisfiability Modulo Theories (SMT) formulas, and we leverage the state-of-the-art, off-the-shelf SMT solver Z3 and CVC4 to verify the formulas with a set of generalized IoT-access properties. *P-Verifier* reports counterexamples indicating security flaws in the *IoT Policies*. In doing so, *P-Verifier* addressed a few fundamental, IoT-unique challenges. In particular, it is difficult to check whether a string-acceptor model (the cloud-general policy enforcement model) fully denies access to an IoT resource without thorough semantic analysis, and it is difficult to reason about the actually allowed permissions by an *IoT Policy* due to the semantic complexity/flexibility of IoT resources (§ 4). To enable a thorough evaluation, we introduce *IoT-Policy Bench* (§ 5.1), a new test suite (with 403 flawed and 303 secure *IoT Policies*) that is designed to evaluate *IoT Policy* analysis tools. With a thorough evaluation, *P-Verifier* shows high effectiveness (zero false positives/negatives) and low performance overhead (see § 4.1).

Contributions. The contributions are outlined as follows:

• *New understanding*. We performed a new, systematic study on the security of cloud-based IoT access policies on modern PaaS/IaaS IoT clouds. Our research brings to light new categories of security-critical vulnerabilities in the design and development of *IoT Policies*, the serious consequences once the vulnerabilities are exploited, and the fundamental challenges in addressing the problems. The lessons learned will contribute to more secure design and practices in the modern, cloud-based IoT development/deployment infrastructure.

• *New techniques*. Based upon the understanding, we developed a formal verification tool *P-Verifier* that can effectively verify cloud-based *IoT Policies*. *P-Verifier* can help IoT manufacturers automatically identify policy flaws, elevating security assurance in modern

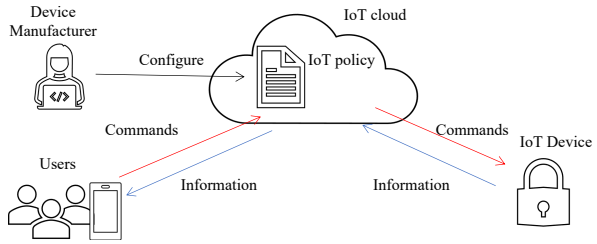

Figure 1: Architecture of cloud-based IoT communication

IoT deployments. We also introduced *IoT-Policy Bench*, a benchmark designed to evaluate *IoT Policy* analysis tools. We open-sourced *P-Verifier* and released *IoT-Policy Bench* [43].

2 THE CLOUD-BASED IOT ACCESS-CONTROL INFRASTRUCTURE

Architecture of Cloud-based IoT Communication. A cloud-based IoT system typically includes three components: the *IoT cloud*, the *IoT device*, and the user's *management console* (mobile apps in particular) to control the device, as illustrated in Figure 1. Central to the system is the cloud that facilitates/mediates the communication between the device and the app, through which the app sends control messages (commands) to the device (e.g., to lock a smart door) and gets information back from the device (e.g., sensor values, the status of a lock). To protect such interactions, the cloud enforces security policies specified by the device manufacturer, and decides whether a user should be allowed to control a device or receive messages from it.

Message Queuing Telemetry Transport (MQTT). MQTT is arguably the most popular messaging protocol in cloud-based IoT communication. MQTT leverages a classical publish-subscribe pattern [2]: the client publishes a message to a named *topic* hosted by the server, which then relays the message to other clients that subscribed to the topic; a topic is named similar to a file path with multiple/many levels separated by "/", such as `/[DeviceId]/status`. Figure 2 illustrates the communication. First, the client (a device or app) connects with the cloud server. An IoT device *subscribes* to its unique *topic* (e.g., `/[DeviceId]/cmd`) by sending a *SUBSCRIBE* message (including the *topic* name) to the server. The server maintains the subscription status. The user's app can command the device, by *publishing* messages with commands (e.g., start or stop) to the topic the device subscribes to. Also, the device can publish messages (e.g., sensor values, activities, status) to its topic that the user app subscribes to. *A message can include commands or informational texts.*

A client can use MQTT wildcards # (matching any number of levels in a topic) or + (matching one level) to subscribe to multiple topics. For example, by subscribing to the topic `/a/#`, one would receive messages published to `/a/b`, `/a/c`, ..., `/a/b/a`, `/a/b/b`, etc. By subscribing to `/a/+`, one would receive messages published to `/a/b`, `/a/c`, etc.

AWS IoT Policy. An *IoT Policy* on AWS inherits the syntax of AWS IAM policies. An *IoT Policy* is a JSON document that contains one or more policy statements (see Figure 3), and is applied to a principle to make access decisions against the principle's request (to AWS IoT). Each statement contains a tuple: `(Effect, Action, Resource)`.

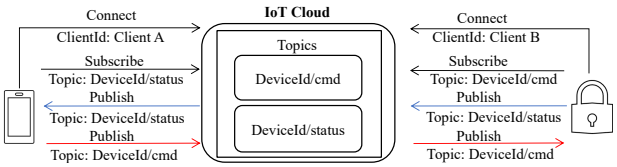

Figure 2: MQTT-based IoT communication

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "client/${iot:Connection.Thing.ThingName}"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
                "topicfilter/a/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": "topic/a/b/*"
        }
    ]
}
```

Figure 3: An Example of IoT policy

`Effect` is either *Allow* or *Deny*. By default, access to a resource is denied. *Allow* statements override the default permissions, and *Deny* statements override the permissions granted by *Allow* statements. That is, to get access to a resource, there must be an allow statement that grants the access and no deny statement that revokes that access. The `Action` construct specifies the IoT-related action(s) that are either allowed or denied on the corresponding resource, such as *iot:Publish* and *iot:Subscribe* (or simply referred to as *Publish* and *Subscribe*). The `Resource` construct specifies the list of IoT resources to which access is either granted or denied, such as an MQTT topic. Like any AWS resource, an IoT resource is unique and identified by a string value. String values can include the wildcard * which matches any number of characters. An IoT policy does not include a principal field, and is assigned to a principal/user by the developers (e.g., through the AWS API [18]). When the principal/user makes requests to AWS-IoT, the cloud enforces the policy against the principal/user.

Considering the potentially huge number of IoT end-users, instead of creating a separate IoT policy (a JSON document) for each user — AWS IoT supports/recommends a template-style IoT policy with *policy variables* (Figure 3). With such an IoT policy, when a client makes an API request to AWS IoT, the variable values are populated, based on which an access decision can be made. In Figure 3, for example, the variable `${iot:Connection.Thing.ThingName}` will be populated at runtime by AWS IoT to be the user-unique *thing name* [14] (similar to the user identity). This template feature can help device manufacturers avoid hard coding specific *thing name* in the IoT policy, and, thus, use the same policy for many/all users.

3 ERROR SPACE IN CLOUD-BASED IOT ACCESS POLICIES

To understand the error space and security challenges in cloud-based IoT deployment, we studied the policy design and practices of 36 mainstream IoT device manufacturers (e.g., WeMo, Govee, Netvue) who deploy IoT devices leveraging the AWS IoT. Our systematic security analysis brings to light the fundamental, general design challenges and tremendous error space for device manufacturers to program/deploy flawed IoT policies. To confirm the weaknesses we identified, we purchased 12 real devices of the affected vendors (or used their mobile apps), and performed end-to-end and/or proof-of-concept exploits. The exploits would have allowed unauthorized users in the wild to control target devices or even completely control all devices of the manufacturer deployed under the cloud, or stealthily receive private personal information (e.g., name, ID, behavior habits, routines, and medical/health data such as blood pressure, weight, and height, see measurement in § 3.4). We also thoroughly inspected 310 open-source cloud-based IoT projects on GitHub and 172 of them have different levels of logic flaws in their *IoT Policies*.

Threat model. We consider realistic attack and application scenarios. In particular, the adversary can open user accounts with IoT device manufacturers and IoT clouds and is capable of collecting and analyzing network traffic between the IoT cloud, the IoT device, and the app under his control. On the other hand, he cannot eavesdrop on or interfere with the communication of other users' devices and apps. He can read public (not proprietary) documentation of the IoT clouds and public protocol specifications. We consider the IoT cloud infrastructure and systems to be benign (the cloud, management console, IoT hardware and firmware in the device).

3.1 Design-Space Flaw 1: Semantic Gap in IoT Access Policies

In general, an access policy on AWS (concerning both IAM policy and IoT policy) specifies the set of resources that a principle is allowed and denied to access. The IoT policy of `Hippokura X MyNavi` (called `Hippokura` for short, a popular IoT-based medical application), for example, specifies that the user cannot subscribe to the topic with a wildcard `xmd/session/#`. In a legitimate scenario, the user will only subscribe to a topic such as `xmd/session/[Session ID]` in which the last part in the topic is her confidential session ID only known by herself or intended users. She can subscribe to such an MQTT topic to receive private/confidential messages but cannot subscribe to the wildcard topic which would otherwise effectively subscribe her to all other users' topics with different session IDs.

The AWS-general policy enforcement mechanism is that, given a request to access a particular resource, abstracted by a string (e.g., the MQTT topic `xmd/session/#` or `xmd/session/[a particular session ID]`, AWS allows the request if the string matches the string(s) defined in the *allow* statement(s) of the policy (notably the wildcard "*" represents any strings on AWS, see Figure 3) and does not match the string(s) in the *deny* statement(s). To facilitate the reasoning of possible security gaps, we can abstract the AWS policy enforcement mechanism as an automaton model (i.e., a finite state acceptor for strings): regarding a type of action (e.g., *subscribe*), each policy defines a string-acceptor *sa* which decides, given an input string *res* that describes a specific resource to access (e.g., an MQTT topic), whether or not *res* is accepted by the *sa*, and, thus, whether or not the access is allowed.

Although such a general enforcement model has long been successful in cloud computing, our research shows that it cannot soundly restrict IoT resources whose semantics and expressions are more complicated than what was understood before with serious implications to access control. In particular, *it is fundamentally difficult for a string-acceptor-based automaton model to deny all synonyms of an IoT resource without thorough semantic analysis.* That is, if two inputs/strings share the same/similar semantic (e.g., referring to the same MQTT topic), the definition of the automaton *sa* must be sound to exclude all related synonyms (strings) that relate to the MQTT topic. Figuring out such synonyms relies on the context of IoT and cannot be cloud-general. For example, based on the MQTT protocol, subscribing to the topic `xmd/#` effectively subscribes the client to many topics including all under `xmd/session/[any string]`. Without excluding the topic `xmd/#` in the policy of Hippokura (using the *deny* statement), we found that a malicious user could subscribe to the topic and receive all other users' messages under `Hippokura`. In our experiment, the leaked messages to an attacker include doctor conversations and personal information related to the patients (see PoC exploit below).

Difficulties for a policy fix. A sound policy fix can be difficult in practice for IoT vendors who use AWS IoT. In the above example, the policy flaw can be difficult for the vendors to notice since the normal functionalities are not affected — a user normally only accesses the topic xmd/session/[his/her session ID], which is allowed by the policy (allow xmd/session/*, deny xmd/session/#). Further, adding a *deny* statement for topic `xmd/#` is insufficient and because of the highly expressive, flexible syntax for describing the same IoT resources, a malicious user can alternatively subscribe to # to actually subscribe to all topics and receive all messages of the doctors/patients under `Hippokura` (see our PoC exploit below). Actually, our study shows that the attackers may alternatively subscribe to `xmd/session/+` and potentially lots of other IoT-synonyms to bypass IoT access control (see § 4.1).

The problem is general and potentially affects many IoT device manufacturers (see measurement in § 3.4) and other IoT clouds (see § 5). As also shown in the prior work [102], the leaked MQTT messages can be highly privacy- and security-sensitive, including SwitchMate users' device IDs, private MQTT topics, device activities, etc., or can be used to infer personal usage habits. Notably, using the leaked device topics, combing a flaw discussed in Section 3.3, we found that a malicious user could control (turn on/off) any SwitchMate switches anytime (Section 3.4).

Further, we found that the policy of Govee smart plugs (Figure 4) plausibly avoided the problem above although it actually failed to do so. The policy intends to deny subscription to topics using MQTT wildcards (#, +) and only allows subscription to particular, known MQTT topics — in a format of `GD/[MD5 of the device ID]`. Effectively, one cannot subscribe to topics such as `GD/#` or `GD/+` for subscribing to all/many users' device topics. However, our study (detailed in the measurement Section 3.4) shows that such a policy, although likely carefully crafted, *is still unsound and insecure* after more thorough reasoning (see our tool in § 4): the filtering

of wildcards is incomplete, and a malicious Govee user can still subscribe to the topic +/ (and effectively subscribes to multiple topics of Govee such as LWT/, detailed in Appendix § .1).

Notably, due to the potentially huge number of users, it can be extremely cumbersome for device manufacturers to create/maintain separate policies for each individual IoT user, and, thus, device manufacturers commonly develop a unified IoT policy — as also recommended/supported by AWS IoT [17] — and apply it to many users. The policy of Hippokura, for example, needs to be permissive supporting many users (allow xmd/session/*) and thus should explicitly exclude/deny certain wildcard-topics (xmd/session/#) which otherwise can subscribe one to other users' topics.

PoC exploit. We conducted PoC experiments for the above problems using real devices of SwitchMate and Govee and the app of Hippokura. We developed a script to connect to their public endpoints on AWS IoT (a7zl8evrsaz7q-ats.iot.us-east-1.amazonaws.com, aqm3wd1qlc3dy-ats.iot.us-east-1.amazonaws.com, and a2qare4ca4-lmz2-ats.iot.ap-northeast-1.amazonaws.com respectively), just as their mobile apps could do. We used our own account credentials (obtained by reverse engineering the apps) to authenticate the connection. Our script could make requests to those endpoints in an attempt to subscribe to intended topics, just like what real attackers could do. We confirmed that the subscription to the aforementioned topics could succeed and stopped the connection immediately without impacts to other parties (see IRB approval, responsible experiment design, and vendor acknowledgment in § 3.4). We reported all findings to the vendors, discussed the possible consequences, and helped them fix the issues, which were acknowledged by the vendors.

```
(( allow,
   action    :   iot:Subscribe,
   resource  :   *),
 ( deny,
   action    :   iot:Subscribe,
   resource  :   (*+, *#)))
```

Figure 4: IoT policy of Govee plugs

3.2 Design-Space Flaw 2: Flawed Cooperation between IAM Policy and IoT Policy

In addition to the challenges with enforcing policies (Flaw 1), our study shows that the highly-coupled but vaguely defined relations between IoT policy and the AWS-general IAM policy make the development of sound IoT access policies even more challenging with serious real-world implications.

AWS IoT directly leverages two other AWS services, i.e., Cognito and IAM, to help with authentication and authorization. As outlined in Figure 6, an IoT user/app logs in with AWS Cognito; based on the user identity (called *Cognito identity ID*) already recorded by the device manufacturer in an identity pool (Figure 6), Cognito returns to the user/client a *Cognito credential* (a secret string such as aws_access_key_id=ASIAR4BOIXWCSZG53UU5), referred to as *cred_authed*. As defined by AWS, the identity pool (with many/multiple identities in the pool) created by the manufacturer is associated with one (or more) IAM policy (e.g., *iam_p1* in Figure 6). The user/client presenting the *cred_authed* can make API requests to any AWS service (e.g., AWS IoT, S3, DynamoDB) as allowed by the IAM policy. On receiving the request, the target AWS service (e.g., AWS IoT, S3) queries Cognito based on the *cred_authed*, to obtain the user's Cognito identity and associated IAM policy, and based on the service-specific statements in the IAM policy, enforces the policy by determining whether the particular action in the request (e.g., s3:ListBucket, iot:Publish, iot:AttachPolicy) is allowed. Interestingly, unlike common AWS services (e.g., S3) that only use the IAM policies to authorize an API request, AWS IoT additionally requires an IoT policy associated with the client. In particular, a client/principle whose associated IAM policy allows the "iot:AttachPolicy" action can attach a given IoT policy to a given Cognito identity, by calling the AWS IoT API *attach_policy* [18].

Modeling the logical relation between IAM policy and IoT policy. We observed that both the IAM policy and IoT policy associated with a client/principle can specify its allowed IoT actions (e.g., iot:Subscribe, iot:Publish) and IoT resources (e.g., MQTT topics). Although poorly/not documented, we observed a few logical relations between a client's IAM policy and IoT policy: (1) The client must have a permissive IAM policy to connect to/make API requests with AWS IoT — the IAM policy should allow IoT-related actions (in contrast to those of other AWS services) such as `iot:AttachPolicy` and `iot:Subscribe`. Such a requirement of AWS IoT is likely inherited from *all* AWS services which generally rely on the AWS-wide IAM policy evaluation engine to determine whether or not the AWS service (i.e., AWS IoT here) should process the API requests [61]. (2) For the client to control/interact with an IoT device (by making requests to AWS IoT, see Figure 6), the specific IoT actions (e.g., `iot:Publish`, `iot:Subscribe`) [16] and target resources must be allowed in the IoT policy. (3) By default, an empty IoT policy is associated with the client, nullifying any allowed actions and resources in IAM policies related to IoT-device controls/interactions (e.g., `iot:Publish`, `iot:Subscribe`) — the security principle of "fail-safe default" [29]. To model and better reason about the logical relations between IAM and IoT policies, we abstract the IoT-related permissions (conceptually meaning the allowed IoT actions such as `iot:Subscribe` and resources such as MQTT topics) allowed by an IAM policy as P_{iam} and those allowed by an IoT policy as P_{iot}. The effective IoT-related permissions of a client c is

$$P_c = P_{iam} \cap P_{iot} \quad (1)$$

Simply put, from the device manufacturer's perspective, for the whole IoT application to function despite the complicated IAM/IoT policy development, (1) an IAM policy must/can be highly permissive, which otherwise prevents the clients from connecting to AWS IoT or making any IoT related API requests; (2) security is achieved as long as a proper, restrictive IoT policy is used. Indeed, based on our study of 36 IoT vendors and 310 open-source GitHub projects, we found that IoT vendors/developers generally bear the above (or similar) understanding, by actually defining a highly permissive IAM policy for IoT users (e.g., using `iot:*` to allow any IoT APIs to be called to AWS IoT, see the policy in Figure 5) while striving to use a restricted, secure IoT policy to achieve secure access control.

What logic can go wrong. However, our research shows that the logical relations between IAM policy and IoT policy, and between AWS IoT's authorization logic and Cognito (the AWS-wide authorization service) are far more complicated than expected. The

```
(( allow,
   action     :   iot:*
   resource   :   *     ))
```

Figure 5: An overly permissive IAM policy

actual complexity generally failed to be understood and properly handled by real-world IoT vendors/developers, leading to security-critical logic loopholes in cloud-based IoT access-control policy development.

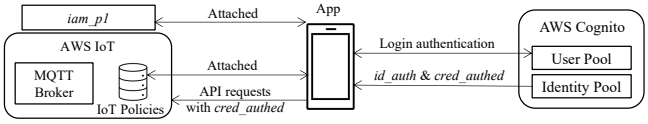

Figure 6: Cognito auth flow

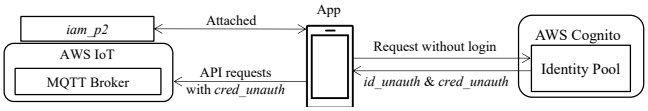

Figure 7: Cognito unauth flow

Specifically, although a highly permissive IAM policy may not lead to a security violation, as backed by an IoT policy if properly developed (see Equation (1)), we found that this could fail in certain circumstances. In particular, AWS allows unauthenticated users (without login) to still use part of the services, for example, to publish logs to AWS CloudWatch Logs [46] under the manufacturers' discretion. We found that a real example is with Molekule, a manufacturer of high-end smart air purifiers [4] (cleared by the Food and Drug Administration to kill bacteria and viruses including the coronavirus [5]). Before a login, the Molekule mobile app fetches a Cognito credential *cred_unauth* representing an AWS-supported *unauthenticated Cognito role* (or *unauth role* for short) and an anonymous/ephemeral Cognito identity *id_unauth* for the user. As configured by Molekule, the *unauth role* is associated with an IAM policy *iam_p2* (see Figure 7) which allows the unauthenticated user to access a few IoT-related functionalities. In particular, *iam_p2* allows the *unauth role* to perform an IoT action `iot:AttachPolicy`, so that when the user logs in (obtaining an authenticated identity *id_authed* and credential *cred_authed*), the client by presenting the *cred_unauth* calls the IoT API `iot:AttachPolicy` to attach an IoT policy to the authenticated identity. The Molekule app will then leverage the authenticated identity/credential (*id_authed*, *cred_authed*) to make API requests [20] to AWS IoT to control/operate the device.

Behind such complicated policy practices, we find that the manufacturer strives to seek simplicity and sets the IAM policy *iam_p2* (for the *unauth role*, see Figure 7) to be highly permissive (e.g., allowing multiple IoT actions on multiple resources), relying on the property that the default "empty" IoT policy (see above) will prevent the unauthenticated users from sending IoT requests/commands (e.g., `iot:Publish`, `iot:Subscribe`) to AWS IoT. We found that such a property is partially invalidated, since AWS IoT does not maintain IoT policies for anonymous/ephemeral Cognito identities (not even maintaining an empty policy, see the design rationale below). Hence, the actual permissions of the unauthenticated identity *id_unauth* is calculated as:

$$P_{id\_unauth} = P_{iam\_p2} \tag{2}$$

Note that the cloud imposed convoluted/unclear relations and cooperations between the cloud-general policies (IAM) and IoT policies. We found equation (2) reflects the true authorization logic for the unauthenticated identity. This effectively gives the *unauth role* all permissions in the *iam_p2*, which, thus, are overly permissive. We find that an unauthenticated Molekule user can send any IoT commands to any Molekule devices (e.g., `iot:Subscribe` to topic # to subscribe to all users' device topics, and `iot:Publish` commands to any users' devices such as for turning on/off). In contrast, an authenticated Molekule user, restricted by the IoT policy, has fewer permissions.

Notably, we found that the design rationale under which AWS IoT does not maintain IoT policies for anonymous/ephemeral Cognito identities is that, AWS IoT does not manage Cognito identities (which is the responsibility of the AWS Cognito service): when AWS IoT receives a Cognito credential *cred_authed* that comes with a request, AWS IoT queries the Cognito service to obtain the corresponding Cognito identity *id_authed*, and uses it to retrieve/index the IoT policy. Since AWS IoT does not issue/manage Cognito identities, it is difficult should it maintain IoT policies for ephemeral Cognito identities (e.g., difficult/costly to synchronously delete the IoT policies once an ephemeral identity is gone).

Last but not least, we found that the problem is general and other mainstream IoT vendors such as broil-king [25], sun-pro [42], and Biobeat VitalDisplay [24] all come with the same problem in their products, allowing unauthenticated users to control all other users' devices with serious security, safety, and privacy implications (see measurement in Section 3.4). Further, we discuss why AWS IoT needs separate IoT policies besides IAM policies in Appendix § .2.

PoC exploit. We performed PoC experiments using our real devices of Biobeat and Molekule and confirmed their all suffered from the above problem. We reverse engineered the mobile apps of the IoT vendors and without login in the IoT vendors' mobile apps, we obtained the Cognito credentials for the *unauth role* (by dynamically instrumenting the app) after we opened each app. With the Cognito credentials, our script could connect to the public endpoints of the vendors on AWS IoT, just like what their apps could do. Our script (representing an unauthenticated app user) could subscribe to the wildcard topic # (we filtered messages only related to our own devices based on our device ID and dropped the rest) and publish arbitrary commands to the "victim" devices (our own devices). We reported all findings to the vendors, discussed the possible impacts/consequences and helped them fix the issues, which were acknowledged by the vendors.

3.3 Implementation-Space Flaws: Chaotic Practices of IoT Policy Deployments

Flaw 1 and Flaw 2 above highlight the design-space challenges/flaws with IoT access policies. Further, our study shows that mainstream device manufacturers use diverse practices with *IoT Policy* deployment, and generally make implementation-level, security-critical mistakes. These mistakes are diverse, non-trivial to avoid in the first place due to the disturbing lack of standard, adequate security practices with cloud-based IoT policy deployment.

```
1 ((allow,
2   action   :  iot:Publish,
3   resource :  (topic/dc/*,
4                topic/sc/${iot:Connection.Thing.ThingName}_*),
5  (allow,
6   action   :  iot:Subscribe,
7   resource : (topicfilter/cs/${iot:Connection.Thing.ThingName}/*,
8               topicfilter/cd/${iot:Connection.Thing.ThingName}/*))
```

Figure 8: Part of NetVue's IoT Policy

Flaw 3: Dilemma with IoT policy templates. Based on the "DRY" principle in the software industry (every piece of knowledge must have a single representation [81]), developing template-based IoT policy (using *policy variables*, see § 2) to reduce boilerplates (redundant policies/codes for each individual user with substantial maintenance burden and a higher chance of unintended inconsistency) is intended. However, our research shows that problems arise since the manufacturers commonly misuse *policy variables*.

For example, NetVue (a popular home-safety cam) has an *IoT Policy* as Figure 8. The resource field `cs/${ThingName}` with a variable `${ThingName}` will be populated at runtime to be the *topic* of a specific user (e.g., `cs/58412338f7944fb0`). This policy applied to all NetVue users effectively/securely restricts that a user can only subscribe to her own *topic*. Also, when a NetVue cam publishes a message/status to its device-specific topic such as `dc/4047512672901241/control`, based on internal forwarding rules (with AWS IoT rules engine [39]), the message will be forwarded to the legitimate user's (e.g., device owner) topic (e.g., `cs/58412338f7944fb0`). That is, security is achieved both by an IoT policy and the forwarding rules configured by the manufacturer.

We found that, although the usage of *policy variables* reduces the burden of creating separate policies for each user, the semantics and expressiveness of *policy variables* are insufficient to express necessary IoT authorization requirements — a design limitation. In particular, using a generic template-based policy for all/many users, the manufacturer was incapable of expressing which are the device topics that specific users can access (*publish* messages to). This is because AWS *policy variables* are populated from the API requester/user's attributes (e.g., source ip, MQTT *ClientId*, a pre-registered thing name [14]) to identify a user, and, likely for this reason, no *policy variables* can be used to specify complicated relations, i.e., the list of allowed devices of the user which may quickly change over time. Consequently, we observed that IoT vendors likely have to resort to few functionality-working workarounds, for example, commonly using a wildcard (e.g., *dc/**, see line 3 in Figure 8) to allow a user to publish to any device topics. We observed such an insecure practice in using IoT policy template with mainstream IoT vendors NetVue, Govee, Belkin, etc. (see the full list in measurement § 3.4). The consequences are severe. For example, a prior employee/Airbnb/hotel/guest user whose permission has been revoked can operate the NetVue cam (by sending commands to the device's topic, see PoC below), turning its angle up to 360 degrees so it cannot monitor the owner-intended space.

Flaw 4: The constraint of IoT-policy mutual exclusion. An access control system can be enhanced with the capability to establish relations/constraints between permissions. For example, two roles can be established as mutually exclusive, so the same user is not allowed to take on both roles [90]. Such a constraint helped fulfill the security principle "Privilege Separation" or increased the difficulty of collusion among individuals [96]. Interestingly, in the IoT context, our study shows that a similar principle is also required when associating a "high-privilege" IoT policy (e.g., with "admin" permissions for an IoT device) with users. We find that, for example, any user who can temporarily physically touch a button on the SwitchBot device can pair with it and is assigned the same "admin" level IoT policy to fully control the smart switch (turn on/off, factory-reset, monitor device activities). Consequently, for example, an ex-employee/Airbnb/hotel/guest user, once assigned the IoT policy for the device, can fully control the device when it is serving other users. Even worse, we find that even if the new user resets the device (by pressing a button both on the device and in the SwitchBot app), the ex-user still bears the IoT policy and can control the device. Such a practice comes with serious safety, security, and privacy implications considering, for example, home-safety, health-related devices are connected with the SwitchBot switch. Hence, the same IoT policy for managing the same IoT device should not be assigned to different users without restrictions. Our study suggests the risks without practice/adaption of "mutual exclusion" constraint in assigning IoT policies, which was not mentioned in AWS IoT best practices [7]. We have reported the findings to the affected vendors and AWS IoT.

PoC exploit. We performed PoC experiments using our real devices of NetVue, Govee, Beurer FreshHome, and FirstAlert Onelink (Smoke and CO Detector, see all device types in Appendix Table 1) for all the flaws above. Similar to PoC in the above sections, we had a script that could connect to the public endpoints of the vendors on AWS IoT. We obtained the AWS credential of our own accounts by instrumenting the apps. **Flaw 3:** with our NetVue cam, our "malicious" script, whose underlying credential/user did not have the permission for the cam, could publish commands (MQTT messages like {"payload" : "ptz":"x":-20,"y":0, "token" : "uewlylljoewbheek" , "clientId" : "957e9eb81b8e4fe5"}) to the target cam's topic and control the cam's angle arbitrarily. **Flaw 4:** We registered two user accounts with SwitchBot, which could independently pair with (after pressing a device button) and fully control our SwitchBot smart switch without restriction. Each user account by using the mobile app could not observe that the device was also granted to the other account.

3.4 Measurement of Impact

To understand the impact and pervasiveness of the problems with cloud-based *IoT Policies*, we studied devices/mobile-apps of 36 IoT manufacturers and 310 open-source AWS-IoT-based projects from GitHub. We purchased 12 real devices of 9 vendors for end-to-end PoC experiments, and the study of the other vendors was through analyzing their mobile apps (i.e., app behaviors and traffic).

Appendix Table 1 shows the overall results with real vendors in our experiments: 13 IoT vendors were confirmed to have 17 instances of *IoT Policy* flaws discussed in the above sections, affecting potentially at least 3.3 million IoT users (based on the number of app downloads on Google Play). All four types of *IoT Policy* flaws are general, each affecting multiple vendors; some vendors suffered from multiple flaws (see Appendix Table 1). For example, the Govee plug and Belkin Wemo smart plug (each with more than one million app downloads) have two different flaws in their policies. The attack consequences (Appendix Table 1) are serious, impacting

safety, security, and privacy. For example, any users can control all *t2Fi* users' smart grills (e.g., control temperature), with serious danger of fire/safety; any users can collect all *Biobeat* users' sensitive health/medical information such as blood pressure, height, weight, and age. We also searched (using keywords such as "AWS IoT", "IoT policy") and downloaded 2,587 open-source cloud-based IoT projects from GitHub; we filtered 310 projects involving IoT policies and IAM policies, among which 172 projects have the flaws discussed above. Specifically, 147 projects involved flawed IoT policies (related to § 3.1 and § 3.3), and 43 projects had overly permissive IAM policies (related to § 3.2). Detailed device models and GitHub repositories are presented online [43].

Vendor acknowledgments and responsible experiments. Notably, we confirmed the policy flaws and attack consequences based on (1) PoC experiments (using our own devices/traffic, and we never attacked/impacted other users' security, safety, and privacy) and (2) feedback from the IoT manufacturers. Additionally, our experiment was approved by our university's IRB. In particular, any data we processed was encrypted in transit and at rest; we do not store any personally identifiable information (PII), e.g., device ID, user ID; as a preventative procedure, any PII if appearing in our experiment was hashed through SHA-512 at runtime and again was never stored. Notably, in case a potential PoC experiment might impact other users, we did not fully conduct the experiment end-to-end; we reported/discussed the projected issues with the vendors for confirmation and helped them fix the problems. Our efforts to help the vendors were well received and acknowledged through their bug bounty programs. For PoC experiments that only affect our own devices/accounts, we implemented full end-to-end PoC attacks. We are also reporting the issues to affected GitHub project owners.

4 LOGICAL ENCODING AND FORMAL VERIFICATION OF IOT ACCESS POLICIES

Given the semantic and logical complexity, manually reasoning about security of *IoT Policies* is difficult and error-prone, especially in large policies with multiple correlated statements, operators, and multiple wildcards. To help IoT manufacturers automatically reason about *IoT Policies*, identify the flaws, and conveniently fix the issues, in this section, we elaborate on the design and implementation of *P-Verifier*, a tool capable of formally defining and verifying security properties for cloud-based IoT access policies.

4.1 Overview

Our security analysis (§ 3) suggests that *IoT Policies* developed by IoT manufacturers failed to soundly protect intended IoT resources and block requests from unintended parties. To reason about security flaws in *IoT Policies*, our innovative approach is built on the generic framework of ZELKOVA [54] (the state-of-the-art for verifying the AWS-general IAM policies) and addresses the unique semantic and logical challenges in *IoT Policies* (see design below and our thorough, end-to-end comparison with ZELKOVA in § 5). *P-Verifier* takes as input *IoT Policies* and develops formal models that fully, and semantically represent IoT policies; the models are encoded with Satisfiability Modulo Theories (SMT) formulas, and we leverage the state-of-the-art SMT solvers Z3 and CVC4 to reason about the formulas with respect to a set of generalized security properties

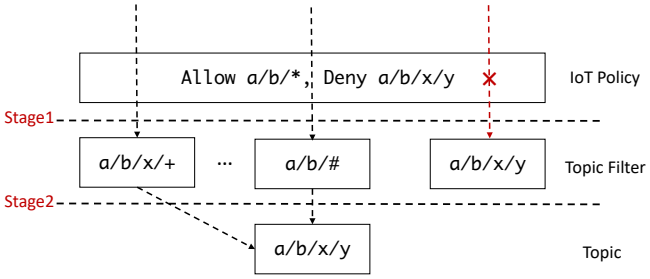

Figure 9: The two-stage trust management for IoT resources on the cloud

and report security flaws in the policies if any. We elaborate on the design as follows and leave the implementation in Appendix § .5.

Capabilities of *P-Verifier*. *P-Verifier* reasons about all possible IoT-related permissions allowed by an *IoT Policy* to verify properties (properties are specified in the same policy language). While the properties to be verified can be specified by developers for specific needs, *P-Verifier* provides three built-in, generalized checks to help eliminate *all* four types of flaws reported earlier (§ 3): (1) whether an *IoT Policy* soundly excludes a permission as expected — **Check 1**, e.g., preventing the principle/user from accessing a target IoT resource; (2) whether an *IoT Policy* is less-or-equally permissive than a reference policy (a policy stating a security property or an upper bound of permissiveness) — **Check 2**; (3) whether multiple independent policies (intended to be assigned to independent users) share permissions — **Check 3**.

Challenges for verifying IoT policies. To enable the above checks in IoT, *P-Verifier* has to address a few key challenges. In particular, *IoT Policies* on the cloud have a unique two-stage trust management (TM) [51, 57, 60, 89, 94] mechanism (Figure 9). Given a request (a dotted vertical arrow above Stage 1 in the figure), in Stage 1 (policy enforcement stage), the cloud's IoT policy engine inspects the resource string in the request (e.g., `a/b/#`) based on the *IoT Policy*. Inspection at this stage is based on the string-acceptor automaton model [47]: each policy essentially specifies an automaton that determines whether a request string (e.g., `a/b/x/+`, `a/b/#`, and `a/b/x/y`) is allowed. A resource request string that has passed Stage 1 goes through a resource reference stage (Stage 2); for example, subscribing to `a/b/x/+` or `a/b/#` (an MQTT topic with + or # is an MQTT topic filter, see § 2) can actually subscribe to MQTT topic *a/b/x/y*, for which the *IoT Policy* in Figure 9 intends to deny access. Note that the resource reference is done by IoT message brokers such as MQTT brokers [1, 83]. The string-acceptor automaton model specified by the policy does not effectively restrict requested resource strings like `a/b/x/+` and `a/b/#`, which will actually access the topic *a/b/x/y*, in violation of the security expectation.

Notably, prior approaches [53, 54, 61] could be adapted to model Stage 1 automaton and reason about the allowed request strings: the policy could be encoded as a logical formula or deterministic finite automaton (DFA) and each satisfiable assignment represents a unique request string that is accepted by the policy DFA and thus can pass the policy check. Prior logical-reasoning that ensures a policy to exclude a resource string will certainly deny any requests to that resource. However, prior security guarantee of policy verification is *invalidated* in IoT because an IoT-synonym (e.g., a/b/x/+)

```
(( allow,
   action    :   iot:Subscribe,
   resource  :   topicfilter/a/b/*),
 ( deny,
   action    :   iot:Subscribe,
   resource  :   topicfilter/a/b/x/y))
```

(a) Policy X

```
(( allow,
   action    :   iot:Subscribe,
   resource  :   topicfilter/a/b/#))
```

(b) Policy Y

Figure 10: IoT policy X and Y (snippets)

Resource: a/b/x/y

```
#, +/#, +/+/#, +/+/+/#, +/+/+/+, +/+/+/y, +/+/x/#,
+/+/x/+, +/+/x/y, +/b/# , +/b/+/#, +/b/+/+, +/b/+/y,
+/b/x/#, +/b/x/+, +/b/x/y, a/#, a/+/#, a/+/+/#,
a/+/+/+, a/+/+/y, a/+/x/#, a/+/x/+, a/+/x/y, a/b/#,
a/b/+/#,  a/b/+/+, a/b/+/y, a/b/x/#, a/b/x/+, a/b/x/y
```

Figure 11: The set of IS strings, denoted as $ISes(\text{“}a/b/x/y\text{”})$, that can refer to the IoT resource a/b/x/y.

can be the request string, which itself in nature is a DFA representing multiple resources (in prior models [53, 54, 61], a request string represents a unique resource and thus is not a DFA in nature). However, the policy check in Stage 1 (to identify whether a string or IoT-synonym is accepted by the policy DFA) cannot directly take the IoT-synonym as a DFA (thus representing the full semantics of the IoT-synonym) because the IoT-synonym's DFA is based on an alphabet $\Sigma_{IoT-synn}$ different from the alphabet Σ_{policy} of the policy DFA (i.e., $\Sigma_{IoT-synn} = \Sigma_{policy} - \{+, \#\}$). Hence, likely based on state-of-the-art approaches [53, 54, 61], AWS IoT takes the request string (even if being an IoT-synonym) as a literal string (compared to taking it as a DFA) to compare with the policy DFA and decides whether the request string is allowed or not. Such an approach of security verification based on prior models/techniques is inadequate for handling IoT semantics, essentially leading to the security risks.

Further, to model/encode the full permissions of a policy (e.g., to reason about permissions between policies to fulfill Check 2 and 3), we should precisely model the two-stage TM process (Figure 9) to reason about all resources actually allowed by the cloud in each of the consecutive stages. Notably, like prior works [53, 54, 61], we can logically model the policy as a DFA encoded as an SMT formula f (based on theories of strings, regular expressions, see § 4.2), denoting all "literal" request strings accepted by Stage 1. We consider such a DFA as a Stage 1 DFA. However, to reason about actual permissions (resources allowed to access), we should subsequently encode and reason about Stage 2 DFA (IoT synonyms' DFAs), which effectively runs on a reduced alphabet than the policy DFA (see above); for example, allowed strings by Stage 1 include a string $a/b/x/+$ (an IoT-synonym of a/b/x/y), which we should decompose in Stage 2 to fully encode all MQTT topics it can reference/access, including $a/b/x/a, a/b/x/b, a/b/x/c, ..., a/b/x/aa, ...$ Here, without finding out and modeling all such IoT-synonym strings accepted by Stage 1 DFA and decomposing their semantics, one is not able to reason about all permissions allowed by the policy, which has not been done in prior works (also see our thorough, end-to-end comparison with state-of-the-art tools in §5).

```
(( deny,
   action    :   iot:Subscribe,
   resource  :   topicfilter/a/b/x/y))
```

Security property *p1* for policy X

```
(( allow,
   action    :   iot:Subscribe,
   resource  :   topicfilter/a/b/+))
```

Security property *p2* for policy Y

Figure 12: The security properties p_1 and p_2

$$F_{x0} : a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}a/b/*\text{”}$$
$$F_{x1} : a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}a/b/x/y\text{”}$$
$$F_x : F_{x0} \wedge \neg F_{X1}$$

Figure 13: Basic SMT encoding of policy X

$$F_y : a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}a/b/\#\text{”}$$

Figure 14: Basic SMT encoding of policy Y

4.2 Logical Encoding of IoT Access Policies

This section elaborates on *P-Verifier*'s semantic-based logical encoding of *IoT Policies* — a prerequisite to soundly verify the security of *IoT Policies* using a theorem prover such as Z3. We first introduce a basic encoding built on the generic framework *ZELKOVA* [54] (the resulting SMT formulas lack full IoT semantics), and then a novel alphabet-reduced encoding based on the basic encoding to fully encode IoT resources.

Modeling and a basic encoding. An *IoT Policy* is a list of statements, and each statement consists of a tuple (Effect, Action, Resource) (see § 2). The encoding for a policy is a formula over two variables *a* and *r* denoting the action and resource. The permissions granted by the policy are encoded as all the permissions granted by the *allow* statements and not revoked by *deny* statements. Figure 10 illustrates a policy X and Figure 13 presents its SMT encoding: policy X grants access if and only if F_{x0} allows access and F_{x1} does not deny it: $F_{x0} \wedge \neg F_{x1}$. The formula F_x evaluates to true (for an assignment given a request) when the policy grants access. Similarly, Figure 14 shows the basic SMT encoding of policy Y.

Note that, with the wildcard * (see § 2), we are abusing the notation in F_{x0} to say $r = \text{“}a/b/*\text{”}$. Similar to *ZELKOVA*, *P-Verifier* actually uses regular expressions to encode the semantic of the cloud-general wildcards: with the traditional regular expression pattern, "." standards for any single character, and "*" is the Kleene star operator representing zero or more occurrences of the previous character set. Formula (3) illustrates the string encoding related to policy X and Y:

$$\begin{aligned} \text{“}a/b\text{”} &\mapsto Var\ equals\ \text{“}a/b\text{”} \\ \text{“}a/*\text{”} &\mapsto Var\ matches\ \text{“}a/.*\text{”} \\ \text{“}a/b/\#\text{”} &\mapsto Var\ matches\ \text{“}a/b/\#\text{”} \\ \text{“}a/b/+\text{”} &\mapsto Var\ matches\ \text{“}a/b/+\text{”} \end{aligned} \tag{3}$$

Alphabet-reduced encoding for IoT policies. The basic encoding in Formula (3) essentially yields the automaton defined on an alphabet Σ_{stg1} including characters + and #. As mentioned earlier (§ 4.1), on this alphabet, request strings with + or # character (MQTT topic filters in IoT contexts) only literally represent the strings, losing the semantics to represent many IoT resources that they actually reference/access (Figure 11). Hence, to encode all IoT

$$F_{p1} : \neg(a = \text{“}iot : Subscribe\text{”} \wedge r \in ISes(\text{“}a/b/x/y\text{”}))$$
$$F_{p2} : a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}a/b/+\text{”}$$

Figure 15: SMT encoding of security properties p_1 and p_2

resources actually allowed by the policy, we should further decompose Formula (3) and generate an automaton on a reduced alphabet Σ_{stg2}. To this end, *P-Verifier* has two steps. First, we decompose a string with wildcard $*$ (essentially an automaton, such as "a/*") to a set of automata based on the positions of +, # (the characters to remove from the alphabet) in the resulting automaton (see an alphabet-reducing algorithm in Appendix § .4):

$$\begin{aligned} \text{“}a/*\text{”} \mapsto\ & Var\ matches\ \text{“}a/+\text{”} \\ \cup\ & Var\ matches\ \text{“}a/\#\text{”} \\ \cup\ & Var\ matches\ \text{“}a/+/.*\text{”} \\ \cup\ & Var\ matches\ \text{“}a/+/\#\text{”} \\ & \ldots \end{aligned} \tag{4}$$

Second, for each generated automaton (the right side of each row in Formula (4) after $\cup$ or $\mapsto$), we further encode the character + and # based on Σ_{stg2}. Based on their semantics ("#" matching any number of levels within a topic and "+" matches one level [35]), we introduce two sets: (1) U_s, which is the set of UTF-8 symbols (denoted as U) excluding the set of MQTT-reserved symbols/characters, such as "#" and "+", and MQTT-excluded symbols such as the null character; (2) U_p, which is U_s excluding the MQTT topic separator "/". Formula (5) illustrates the alphabet-reduced SMT encoding for resource strings with # and +:

$$\begin{aligned} \text{“}a/b/\#\text{”} &\mapsto Var\ matches\ \text{“}a/b/U_s^*\text{”} \\ \text{“}a/b/+\text{”} &\mapsto Var\ matches\ \text{“}a/b/U_p^*\text{”} \end{aligned} \tag{5}$$

Note that, U_s^* with the Kleene star, denotes all strings over the alphabet U_s, which can include the empty string ϵ. Also, based on the MQTT specification [35], the filter $a/b/\#$ can match the topic a/b. However, some real-world vendors did not strictly follow the standard here (e.g., $a/b/\#$ does not match the topic a/b on AWS IoT). So we omitted a/b for $a/b/\#$ in Equation 5 in our current version.

Discussion. We handle/encode *policy variables* (see § 2) based on their semantics. A *policy variable* (populated by AWS IoT at runtime for each client) is usually used to represent an attribute of the client [12]. As discussed in the prior work [83], some *policy variables* are populated based on the untrusted information (e.g., the *${iot:ClientId}* is populated with any value that the client claims [83]); we simply encode such a *policy variable* as a wildcard "*". In contrast, the *policy variable* `${iot:Connection.Thing.ThingName}` is based on trusted, user-specific value maintained by AWS IoT. We encode such a *policy variable* with a random, unique string.

4.3 Formal Verification and Flaw Detection

With the logical encoding of *IoT Policies*, this section presents how *P-Verifier* enables the three checks (§ 4.1) with respect to three types of security properties. In general, the verification of *P-Verifier* includes a few major steps: (1) translate the target policy into an SMT formula f, (2) check the validity/satisfiability of f with the security property p as a constraint. If the result is not valid (or *satisfiable*, depending on the property), *P-Verifier* reports that the policy f violates the property p (e.g., at least some assignments for f, intuitively meaning some IoT requests, will violate the property).

Check 1. Taking Policy X as an example (Figure 10), *P-Verifier* will first translate the policy into an SMT formula F_x (Figure 13). The expected security property p_1 (Figure 12) — Type 1 property for excluding a permission — is translated into an SMT formula F_{p1}, stating that the policy is expected to deny access to one particular *topic* `a/b/x/y`. Then, *P-Verifier* leverages the SMT solver Z3 to check the satisfiability of

$$F_x \wedge \neg F_{p1} \tag{6}$$

A satisfiable assignment corresponds to an IoT request allowed by F_x while violating the property. Indeed, *P-Verifier* reports (1) F_x violates the security property and (2) the satisfiable assignment. Intuitively, for example, IoT requests to `a/b/+/+` or `a/b/#` are allowed by F_x, and can be used to access/subscribe to the *topic* restricted by property $p1$.

The challenge is to reason about and report *all* property-violating assignments, so the manufacturer can soundly fix the policy (e.g., by specifying literally all the strings in *deny* statements). Our insight is that, an assignment that violates a Type 1 property is essentially an *IS* of the target resource to deny access to. To address the challenge, *P-Verifier* encodes F_{p1} to include the set of all *IS*es of the target resource, as illustrated in Figure 15. Note that $ISes(\text{"}a/b/x/y\text{"})$ denotes the set of all *IS*es of the *topic* "a/b/x/y", which can be easily enumerated (Figure 11) based on MQTT semantics (Appendix § .3).

Check 2. Take policy Y (Figure 10) as an example with its SMT encoding F_y (Figure 14) and a property p_2 (Figure 12), SMT-encoded as F_{p2} (Figure 15) stating a reference/upper-bound permission — Type 2 property — which is to allow access (subscribe) to `a/b/+`. To check if policy Y is less-or-equally permissive than the property F_{p2}, *P-Verifier* uses Z3 to prove formula f:

$$F_y \Rightarrow F_{p2} \tag{7}$$

If the result was *valid*, it indicates that policy Y is less-or-equally permissive than F_{p2}. Actually, the result is *not valid*, and indeed F_y is more permissive than the upper-bound permission. Intuitively, subscribing to *topic* `a/b/#` as allowed by policy Y effectively subscribes to more resources such as the *topic* `a/b/c/d` than the reference F_{p2}. For improved usability, *P-Verifier* reports examples such as `a/b/c/d` to help IoT manufacturers easily understand the problem. This is achieved by solving the formula $\neg f$, and Z3 will report "satisfiable" and an example such as the above.

Check 3. *P-Verifier* can also check whether two policies share permissions (e.g., that independent users share permissions indicates a security risk, see Flaw 4). Take the policies X and Y, encoded as F_x and F_y respectively, as an example: *P-Verifier* leverages Z3 to solve the formula

$$F_x \wedge F_y \tag{8}$$

A result "unsat" (unsatisfiable) indicates that there are no shared permissions between the two policies; a result "sat" (satisfiable) means the two policies have overlapping permissions (with shared IoT resources). To make the results more informative, in our implementation, *P-Verifier* decomposes the policy and takes out each "Allow" statement to check the overlapping with the other policy. This helps IoT vendors easily identify which part of a complex policy has the problem.

Usability discussion. As mentioned above, the outputs of *P-Verifier* are easy to understand and actionable for IoT vendors to fix the policies. Further, *P-Verifier* features convenience for constructing security properties. Specifically, Type 1 property (denying certain resources/permissions) can be easily obtained from the target policy to check (e.g., the property $p1$ for policy X). Also, IoT vendors can integrate *P-Verifier* for necessary security checks in the regular IoT lifecycle when there is a permission change. For example, with Check 3 vendors can avoid Flaw 4: the vendors can leverage *P-Verifier* to check whether an ex-user and the current user share permissions when a device is reset or the ex-user's permissions are revoked; unexpected, shared permissions indicates a security flaw.

Discussion of generality and extensibility. The design of *P-Verifier* enables its generality and potential extensibility to analyze and detect more than the flaws discussed in § 3. Above all, one critical capability offered by *P-Verifier* is the full-semantic encoding and modeling of an IoT policy (representing the complete permissions and resources allowed by the policy), a fundamental prerequisite for developing sophisticated or flaw-specific detection logic. Based on the modeling, our three check capabilities above can be used to verify a variety of high-level security properties.

For example, to manage an IoT lock in a house/organization, suppose the lock owner/administrator is assigned the policy z1 (Figure 16) and the less-privileged users (e.g., employees in the organization or guest users such as an Airbnb/rental guest in the house) is assigned the policy z2 (Figure 16). In such a scenario, one may need to ensure a few high-level security properties: (1) ensure unrestricted public access (e.g., access by guests) to the MQTT topic `deviceId/highpriv/reset` is not allowed since sending commands to this topic can factory-reset the device (least-privilege [7, 34]); (2) ensure the guests' permission is less than (i.e., being a subset of) the owner/administrator permission (least-privilege); (3) ensure the owner/administrator can publish to and subscribe to MQTT topic `deviceId/highpriv/reset` (availability property); (4) ensure the guests can publish to and subscribe to MQTT topic `deviceId/lowpriv/open` (availability property). These properties are described in the policy language in Appendix Figure 21 (pr1 to pr4).

$$\begin{aligned} R1 &= \neg F_{pr1} \wedge F_{z2} \\ R2 &= F_{z2} \Rightarrow F_{pr2} \\ R3 &= F_{pr3} \Rightarrow F_{z1} \\ R4 &= F_{pr4} \Rightarrow F_{z2} \end{aligned} \tag{9}$$

We can leverage *P-Verifier* to verify these high-level security properties. Specifically, using exactly the same encoding approach as described in § 4.2, we first encode the two policies as F_{z1} and F_{z2} and four properties as $F_{pr1}, F_{pr2}, F_{pr3}, F_{pr4}$ (also see the full details of the encoding in equation (10) and equation (11) in Appendix subsection .6 in our long version paper), and then leverage the Check 1 and Check 2 to reason about these formulas and thus verify the properties (see formulas (9)). More specifically, for property pr1, we directly leverage Check 1 to check whether policy z2 excludes the target permission (by reasoning about whether formula $R1$ in equation (9) is satisfiable, similar to equation (6). For pr2, we directly leverage Check 2 to check whether permission of policy z2 is a subset of policy z1's permission (i.e., $R2$ in equation (9)) is valid); for pr3, we can leverage Check 2 and verify whether policy z1 is more or equally permissive than pr3 which encodes a lower-bound permission (i.e., $R3$ is valid); pr4 is similar to pr3. We provide more examples for checking other security properties in Appendix §.7.

5 EVALUATION

We evaluated *P-Verifier* for its high effectiveness and low performance overhead. To enable a thorough evaluation, we introduce *IoT-Policy Bench* comprised of 706 policies, presenting a new test suite that is designed to evaluate *IoT Policy* analysis tools. Our experiment based on *IoT-Policy Bench* shows that *P-Verifier* significantly outperforms the prior tools in precision and coverage on AWS IoT policies. We released *IoT-Policy Bench* online [43].

Policy z1

```
(( allow,
   action   :  iot:Publish,
   resource :  topic/deviceId/*),
 ( allow,
   action   :  iot:Subscribe,
   resource :  topicfilter/deviceId/*))
```

Policy z2

```
(( allow,
   action   :  iot:Publish,
   resource :  topic/deviceId/lowpriv/*),
 ( allow,
   action   :  iot:Subscribe,
   resource : topicfilter/deviceId/lowpriv/*))
```

Figure 16: Example IoT policies to manage an IoT lock

5.1 New Benchmark: *IoT-Policy Bench*

IoT-Policy Bench (*IPB*) includes both secure and flawed *IoT Policies* that aim to best cover the error space (§ 3) in real-world design/development of *IoT Policies*. *IPB* covers all types of flaws in *IoT Policies* discussed in § 3. For example, related to Flaw 1, *IPB* includes a set of policies that failed to soundly excludes an expected permission. For diversity, *IoT-Policy Bench* includes (1) 146 hand-crafted policies by our domain experts (44 flawed, 102 secure), (2) 560 policies we gathered from open-source IoT projects on GitHub (§ 3.4), and (3) 10 flawed IoT policies likely used by real vendors (based on real access-control behaviors of their products, see § 3.4). Our hand-crafted policies are generated by mutating MQTT-topics (i.e., adding/using +/#/*) and statements (i.e., adding/removing) based on policies from GitHub and those likely used by real devices.

For each policy in *IPB*, we leverage multiple domain experts to manually construct its expected security property (similar to Figure 12) and come up with a ground truth. Specifically, the security properties can be constructed based on the scenarios in which the IoT policies can be used. For example, a set of policies in *IPB* gathered from GitHub and in our manually crafted policies include statements to deny access to certain resources (similar to Figure 10), and we can take their security properties as excluding a certain resource/permission (corresponding to our Check 1, § 4.1). As another example, in the real world, two IoT policies may be assigned separately to two separate IoT owners or two independent groups of users; a security property corresponding to our Check 3 (§ 4.1) can be that the two policies do not have overlapped permissions or shared resources to access.

Ground truth composition. As an established ground truth, *IPB* includes 242 IoT policies with Flaw 1 and 40 IoT policies that are secure regarding this flaw. Regarding Flaw 2, *IPB* includes 62 IAM policies with the flaw and 46 secure IAM policies. Last, *IPB* includes 87 IoT policies with Flaw 3 accompanied with 179 confirmed secure IoT policies, and 11 IoT policies with Flaw 4 accompanied with 39

secure IoT policies. Note that Flaw 2 concerns IAM policies used in IoT contexts and all other flaws only concern IoT policies. The entire *IPB* including all ground truth information is released online [43].

5.2 Effectiveness

We ran *P-Verifier* on *IoT-Policy Bench* showing high precision and coverage (zero false positive and false negative). Notably, although real-world manufacturers often did not release their IoT policies, *IoT-Policy Bench* includes our manually crafted snippets of IoT policies for ten IoT manufacturers (e.g., Govee, Belkin, Molekule) based on the vulnerabilities confirmed thanks to our white-hat reports to the manufacturers (following standard responsible disclosure practices [78]) and communication with them. *P-Verifier* shows zero false positive/negative on the entire dataset (released online [43]).

Comparison with ZELKOVA/industry tools. Up to our knowledge, AWS IoT Defender (Defender) [13] (backed by ZELKOVA [54] based on Defender's white paper [45]) is the state-of-the-art and only tool known to be capable of security verification for *IoT Policies*. *P-Verifier* significantly outperforms Defender for the precision and coverage based on functionalities described in their user manuals [13] and our thorough, end-to-end experiment on *IPB* (detailed below). Note that AWS did not release the source code of ZELKOVA and we may not directly compare it with *P-Verifier*.

In our comparison with Defender/ZELKOVA, for each of the flaws separately (Flaw 1 to Flaw 4, see § 3), we assessed the tools' false positive rate (FPR) and false negative rate (FNR) for verifying IoT policies with and without the flaw. If the tool raised false alarms for an actually secure policy (based on our ground truth, see § 5.1), that indicated a false positive. If the tool failed to raise alarms for an IoT policy that is indeed flawed, this indicated a false negative. The detailed results are shown in Appendix Table 2. In particular, while *P-Verifier* shows high precision and detection coverage (i.e., zero false positive and zero false negative), Defender suffers from a low coverage (with FNR up to 100%, see Appendix Table 2) for its inadequate capability to analyze and identify the flaws introduced in this paper.

More specifically, regarding Flaw 1, Defender (with 21.1% FNR) is not capable of analyzing whether a policy specifies sufficient "IoT synonyms" of a resource to deny (in the "deny" statement); the flawed IoT policies that Defender typically reported were those with relatively simple patterns of wildcard "*" (see Figure 18 in Appendix). In the meantime, Defender did not take user-provided security properties as inputs (e.g., deny the access to resource x/y/z as supported by *P-Verifier*), and hence, the detection semantic supported by Defender is much more coarse-grained, mainly focusing on identifying whether the policy in general overly uses "*" in specifying the resource fields. Note that *IPB* includes such simple policies from GitHub, whose over-permission could thus be detected by Defender. Also due to the lack of support for policy-specific security properties and semantic/logic, Defender cannot handle Flaw 4 (with 100% FNR): our *P-Verifier* can compare whether two policies unwittingly share permissions/resources, while Defender can only check policies individually. Regarding Flaw 2, Defender is comparable with *P-Verifier* because the detection needed here is to identify overly permissive IAM policies (e.g., Figure 5), without complicated IoT semantic analysis needed in handling Flaw 1. Regarding Flaw 3, Defender (37.9% FNR) failed to identify overly permissive policies that have a policy-specific or vendor-specific prefix prior to "*" such as "dc/" in NetVue's policy (Figure 8). Additionally, Defender did not properly analyze the semantics of certain policy variables, such as "${iot:Connection.Thing.ThingName}", which effectively will be populated at runtime to be a trusted, user-/client-specific value (§ 3.3), and thus is not overly permissive (see an example in Figure 19 in Appendix). Defender showed false positives in such cases related to Flaw 3. Overall, Defender underperforms *P-Verifier* for these properties due to two major disadvantages: (1) it lacked the sophisticated semantics/logic analysis offered by *P-Verifier*; (2) it did not take expected security properties as inputs.

5.3 Performance Overhead

We evaluated the performance overhead of *P-Verifier* running on the above dataset using a workstation with 3.40GHz Intel i7-6700 CPU, 15.6GB memory and 931.5GB hard drive. With an average of ten runs, *P-Verifier* took 1.15 seconds and 43MB memory at most to fully verify a policy (with Z3 as the prover). We did not observe unusually complicated (with multiple wildcards in multiple levels of the topics) policies that cannot be finished within 120 seconds (the time threshold of *P-Verifier*) in our dataset and after a thorough search of 560 open-source AWS-IoT-based IoT projects on GitHub.

We further evaluated *P-Verifier* using more complex policies (beyond the 2048-non-white-char limit of AWS IoT [11, 40]). For each individual *check* (Check 1 to 3), we reuse the policy X or Y in the original example (§ 4.1) and increase the policy complexity in two separate settings: (1) increasing the number of "allow" statements (each added statement having the similar complexity, i.e., the same string length 20 and wildcard number 1), (2) increasing string length (statement number and wildcard number remain unchanged). Appendix Figure 22 and Figure 23 show the result of the two settings respectively (based on the average results of 20 trials). Specifically, for Setting 1 of each *check*: we observed an approximately linear correlation between the execution time and statement number (in our implementation, *P-Verifier* reasons about each added "allow" statement separately and thus is capable of reporting all individual statements that violate the property — a fine-grained reporting actionable for IoT vendors); for Setting 2, the efficiency of analyzing longer strings depends largely on the performance of the underlying solver (Z3 in our experiment). The performance overhead is based on end-to-end execution time (including the time to read/encode the policy and reason with the property) and the generated (more complex) statements and strings yielded "unsat" result with the security properties (we released all generated policies online [43]). Further, we increased the wildcard number based on policy X and Y (statement number and string length remain unchanged) and observed a result similar to what was reported in [54]: when the number of "*" wildcards is more than 5, Z3 may not terminate (e.g., for the query "is `topic/*/a/*/a/b/y` less or equally permissive than `topic/*/a/*/a/*/y`"). Still, 99% of policies with less than 6 wildcards (600 policies in total, released online [43]) are solved within 0.6 seconds. Note that in all 560 IoT policies we found on GitHub, we did not observe policy statements with more than 3 wildcards in 1 string resource (policy with many wildcards is hard for human to write/understand and thus can violate the programming principles [32]).

6 DISCUSSION

Although our study primarily focuses on the *IoT Policies* on AWS IoT, the problems we identified can generally affect device manufacturers on other IoT clouds (e.g., Azure, Tuya, Alibaba). We indeed found that IoT policies on Azure IoT [38] also suffered from the Flaw 1 (§ 3.1), sharing the same fundamental challenge with AWS IoT as discussed in § 3.1. To fundamentally address the problems, we expect multiple-level efforts: (1) improved verification techniques and new level of formal guarantees (such as those offered by P-Verifier), (2) improved, more open design of IoT clouds, (3) improved developer guides/awareness.

Wildcards in access policies. Mainstream policy languages (such as XACML [48], APPEL [36], AWS IAM policy [31], Azure policy [22], Kubernetes API [33]) and access control policies [19, 23, 26, 27] are designed to support and leverage wildcards. Despite a set of known kinds of security risks with wildcards [59, 82] (e.g., over permissiveness due to careless developers), wildcards are necessary in describing resources at scale, e.g., in production/cloud environment with hundreds or millions of resources [26, 61], bringing significant efficiency in development and maintenance compared to, for example, exhaustively listing all individual resources. More specifically, in the IoT context, wildcards such as + and # are an inherent part of the MQTT protocol, which help easily describe and access multiple/many MQTT topics of a certain user/device/organization. For example, making a single MQTT request to subscribe to deviceID/# effectively subscribes to deviceID/cmd, deviceID/states, deviceID/configure, and more, compared to sending many requests for individual resources; this is important for IoT devices which may work on resource-constrained devices in low-bandwidth or unreliable networks [83, 92]. Hence, wildcards still appear to be part of the state-of-the-practice [19, 23, 26, 27] to develop access policies, and can be hard to completely get rid of. Notably, state-of-the-practices have warned developers to avoid a set of known bad paradigms/practices in using wildcards [3, 9, 15, 41] (e.g., to avoid coarse-grained * for an unbounded range of resources, and use better crafted regular expressions for a finer-grained range of resources). Further, recent efforts from industry and academia [54, 61, 71] have aimed at helping developers construct secure policies supporting wildcards while elevating the security assurance with new formal guarantees. Our work makes new contributions at least in this line of efforts (by identifying new kinds of risks such as the IoT-synonym and formally verifying the policies to provide security guarantees against the new risks).

7 RELATED WORK

Security of cloud-based IoT policies. Under the generic term of "IoT policies," prior works extensively the security, safety and privacy implications of different IoT policies on diverse platforms. [55, 64, 68, 69, 73, 93, 97, 100] studied IoT Trigger-Action platforms (TAP), which suffered from over-privilege recipes (TAP apps), inter-rule vulnerabilities, and privacy implications. On IoT application platforms which support third-party applications (e.g., SmartThings apps [72, 98]), prior works [62, 63, 72, 84, 98] studied their coarse-grained access control model and security and privacy of IoT apps. Voice-Controlled Platforms were also extensively studied [65, 76, 79, 87, 91, 95, 99, 104–106]. [107] studied access control of the emerging "Mobile-as-a-Gateway" IoT paradigm. [83] lightly discussed issues related to wildcards, e.g., vendors missed restricting wildcards at all. We systematically discussed wildcard-related issues using model-based approaches and the fundamental challenges to deny/verify IoT-synonyms for DFA. The prior work [6, 83] concerning IoT messaging protocols only marginally relates to our focus.

Formal methods for analyzing access control policies. Prior works proposed methods to formally reason about access control policies [53, 54, 61, 66, 77, 82, 101]. In the literature, policy languages have been studied [28, 56, 67, 70, 74, 74, 75, 77, 80, 85, 86, 88, 88, 89]. Most related is ZELKOVA [54, 61], which presents the state-of-the-art, being a generic framework to formalize/model and analyze policies on public clouds. *P-Verifier* is built on ZELKOVA while addressing fundamental, new challenges in the IoT context, such as how to reason about *IS*es, fully encode IoT semantics and provide usable, actionable reasoning results for IoT vendors.

8 CONCLUSION

We performed a systematic study on the security of cloud-based IoT access policies. Our research shows that the complexity in the IoT semantics and enforcement logic of *IoT Policy* leaves tremendous space for device manufacturers to program a flawed *IoT Policy*, introducing convoluted logic flaws which are non-trivial to reason about. The problems are general and pervasive, and serious. To help manufacturers identify *IoT Policy* flaws, we introduce *P-Verifier*, a novel formal verification tool that can automatically verify cloud-based *IoT Policies* and is highly effective and efficient. Our work will contribute to elevating security assurance of modern IoT deployments for the cloud-based IoT infrastructure.

ACKNOWLEDGMENTS

Luyi Xing is supported in part by NSF CNS-2145675, CCF-2124225, and Indiana University's FRSP-SF, REF, and IAS. Ze Jin, Yiwei Fang, and Qixu Liu are supported in part by the Youth Innovation Promotion Association CAS (No.2019163), the Strategic Priority Research Program of Chinese Academy of Sciences (No. XDC02040100), the Key Laboratory of Network Assessment Technology at Chinese Academy of Sciences and Beijing Key Laboratory of Network security and Protection Technology. Yan Jia is supported in part by National Natural Science Foundation of China (No. 62102198) and China Postdoctoral Science Foundation (No. 2021M691673).

REFERENCES

[1] 2019. MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.
[2] 2019. Publish–subscribe pattern. https://en.wikipedia.org/wiki/Publish-subscribe_pattern. Accessed: 2019-07.
[3] 2020. Security Best Practices for Amazon S3. https://techcommunity.microsoft.com/t5/azure-architecture-blog/azure-policy-prevent-the-use-of-wildcard-for-source-in-azure/ba-p/1783844.
[4] 2021. Air Purifier official page. https://molekule.com/.
[5] 2021. Molekule Air Mini Receives FDA clearance to destroy viruses and bacteria. https://molekule.science/molekule-air-mini-receives-fda-510k-clearance/.
[6] 2021. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/wan-qinying
[7] 2021. Security best practices in AWS IoT Core. https://docs.aws.amazon.com/iot/latest/developerguide/security-best-practices.html.
[8] 2022. Automata theory . https://en.wikipedia.org/wiki/Automata_theory.
[9] 2022. Avoiding wildcard permissions in IAM policies. https://docs.aws.amazon.com/lambda/latest/operatorguide/wildcard-permissions-iam.html.
[10] 2022. AWS IoT Core. https://aws.amazon.com/en/iot-core/.

[11] 2022. AWS IoT Core endpoints and quotas. https://docs.aws.amazon.com/general/latest/gr/iot-core.html#security-limits/.
[12] 2022. AWS IoT Core policy variables - AWS IoT Core. https://docs.aws.amazon.com/iot/latest/developerguide/iot-policy-variables.html.
[13] 2022. AWS IoT Defender. https://docs.aws.amazon.com/iot/latest/developerguide/device-defender.html.
[14] 2022. AWS IoT official documentation about thing registry. https://docs.aws.amazon.com/iot/latest/developerguide/thing-registry.html.
[15] 2022. AWS IoT policies overly permissive. https://docs.aws.amazon.com/iot/latest/developerguide/audit-chk-iot-policy-permissive.html.
[16] 2022. AWS IoT Policy actions. https://docs.aws.amazon.com/iot/latest/developerguide/iot-policy-actions.html.
[17] 2022. AWS Publish/Subscribe IoT policy examples. https://docs.aws.amazon.com/iot/latest/developerguide/pub-sub-policy.html.
[18] 2022. AWS python SDK boto3 IoT api attach policy. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/iot.html#IoT.Client.attach_policy.
[19] 2022. AWS S3 Policy Example. https://docs.aws.amazon.com/AmazonS3/latest/userguide/example-policies-s3.html.
[20] 2022. AWS SDK for Android - 2.22.1. https://aws-amplify.github.io/aws-sdk-android/docs/reference/.
[21] 2022. Azure IoT Hub. https://azure.microsoft.com/en-us/services/iot-hub/.
[22] 2022. Azure Policy definition structure. https://docs.microsoft.com/en-us/azure/governance/policy/concepts/definition-structure.
[23] 2022. Azure Policy Example. https://github.com/Azure/azure-policy.
[24] 2022. Biobeat official page. https://www.bio-beat.com/.
[25] 2022. Broiking official page. https://broilkingbbq.com/.
[26] 2022. Controlling access to a bucket with user policies. https://docs.aws.amazon.com/AmazonS3/latest/userguide/walkthrough1.html.
[27] 2022. Dynamodb Policy Example. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_examples_dynamodb_specific-table.html.
[28] 2022. eXtensible Access Control Markup Language (XACML) Version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.
[29] 2022. Failing Security|CISA. https://us-cert.cisa.gov/bsi/articles/knowledge/principles/failing-securely.
[30] 2022. Github official page. https://github.com.
[31] 2022. IAM JSON policy elements: Resource. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_resource.html.
[32] 2022. KISS Principle. https://en.wikipedia.org/wiki/KISS_principle.
[33] 2022. Kubernetes' ABAC access control. https://kubernetes.io/docs/reference/access-authn-authz/abac/.
[34] 2022. Least privilege. https://en.wikipedia.org/wiki/Principle_of_least_privilege.
[35] 2022. MQTT Version 3.1.1 specification. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.
[36] 2022. P3P Language. https://www.w3.org/TR/P3P-preferences/.
[37] 2022. Policies and permissions in IAM. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.
[38] 2022. Publish and subscribe with Azure IoT Edge | Microsoft Docs . https://docs.microsoft.com/en-us/azure/iot-edge/how-to-publish-subscribe?view=iotedge-2020-11.
[39] 2022. Rules for AWS IoT - AWS IoT Core. https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html.
[40] 2022. Scaling authorization policies with AWS IoT Core. https://aws.amazon.com/blogs/iot/scaling-authorization-policies-with-aws-iot-core/.
[41] 2022. Security Best Practices for Amazon S3. https://docs.aws.amazon.com/AmazonS3/latest/userguide/security-best-practices.html.
[42] 2022. Sun-Pro google play store page. https://play.google.com/store/apps/details?id=com.SunProtection.
[43] 2022. Supporting website for P-Verifier. https://sites.google.com/view/p-verify/home.
[44] 2022. Tuya IoT Cloud. https://www.tuya.com/.
[45] 2022. Using provable security to enhance IoT – An industry differentiator. https://docs.aws.amazon.com/whitepapers/latest/securing-iot-with-aws/using-provable-security-to-enhance-iot-an-industry-differentiator.html.
[46] 2022. What is Amazon CloudWatch Logs? https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html.
[47] 2022. Wikipedia page Deterministic finite automaton. https://en.wikipedia.org/wiki/Deterministic_finite_automaton.
[48] 2022. XACML policy language OASIS standard. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.
[49] 2022. Z3 String Constraint Solver. https://z3string.github.io/.
[50] 2022. Z3Py. https://ericpony.github.io/z3py-tutorial/guide-examples.htm.
[51] Ava Ahadipour and Martin Schanzenbach. 2017. A Survey on Authorization in Distributed Systems: Information Storage, Data Retrieval and Trust Evaluation. In *2017 IEEE Trustcom/BigDataSE/ICESS*. 1016–1023. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.346
[52] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1362–1380. https://doi.org/10.1109/SP.2019.00013
[53] John Backes, Ulises Berrueco, Tyler Bray, Daniel Brim, Byron Cook, Andrew Gacek, Ranjit Jhala, Kasper Luckow, Sean McLaughlin, Madhav Menon, et al. 2020. Stratified abstraction of access control policies. In *International Conference on Computer Aided Verification*. Springer, 165–176.
[54] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8602994
[55] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If this then what?: Controlling flows in IoT apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1102–1119.
[56] Moritz Y. Becker and Peter Sewell. 2004. Cassandra: Flexible Trust Management, Applied to Electronic Health Records. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 139–154. https://doi.org/10.1109/CSFW.2004.7
[57] E. Bertino, E. Ferrari, and A. Squicciarini. 2004. Trust negotiations: concepts, systems, and languages. *Computing in Science Engineering* 6, 4 (2004), 27–34.
[58] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. 2017. Access control model for AWS internet of things. In *International Conference on Network and System Security*. Springer, 721–736.
[59] Sandeep Bhatt and Prasad Rao. 2008. *Enhancements to the vantage firewall analyzer*. Technical Report. Citeseer.
[60] M. Blaze, J. Feigenbaum, and J. Lacy. 1996. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*. 164–173. https://doi.org/10.1109/SECPRI.1996.502679
[61] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 281–291. https://doi.org/10.1145/3368089.3409728
[62] Z Berkay Celik, Leonardo Babun, Amit K Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. *arXiv preprint arXiv:1802.08307* (2018).
[63] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 147–158.
[64] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT.. In *NDSS*.
[65] Long Cheng, Christin Wilson, Song Liao, Jeffrey Young, Daniel Dong, and Hongxin Hu. 2020. Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1699–1716.
[66] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2020. Cross-app interference threats in smart homes: Categorization, detection and handling. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 411–423.
[67] J. DeTreville. 2002. Binder, a logic-based security language. In *Proceedings 2002 IEEE Symposium on Security and Privacy*. 105–113. https://doi.org/10.1109/SECPRI.2002.1004365
[68] Wenbo Ding and Hongxin Hu. 2018. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 832–846.
[69] Wenbo Ding, Hongxin Hu, and Long Cheng. 2021. IOTSAFE: Enforcing Safety and Security Policy with Real IoT Physical Interaction Discovery. (2021).
[70] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2006. Specifying and Reasoning About Dynamic Access-Control Policies. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4130)*, Ulrich Furbach and Natarajan Shankar (Eds.). Springer, 632–646.
[71] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying permissiveness of access control policies. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1805–1817.
[72] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 636–654.
[73] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *22nd Network*

and Distributed Security Symposium (NDSS 2018).

[74] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. 2005. Verification and change-impact analysis of access-control policies. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 196–205.

[75] Dimitar P. Guelev, Mark Ryan, and Pierre Yves Schobbens. 2004. Model-Checking Access Control Policies. In *Information Security*, Kan Zhang and Yuliang Zheng (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 219–230.

[76] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. 2020. SkillExplorer: Understanding the Behavior of Skills in Large Scale. In *29th USENIX Security Symposium (USENIX Security 20)*. 2649–2666.

[77] William T Hallahan, Ennan Zhai, and Ruzica Piskac. 2017. Automated repair by example for firewalls. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 220–229.

[78] Allen D Householder, Garret Wassermann, Art Manion, and Chris King. 2017. *The cert guide to coordinated vulnerability disclosure.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Pittsburgh United States.

[79] Hang Hu, Limin Yang, Shihan Lin, and Gang Wang. 2020. A Case Study of the Security Vetting Process of Smart-home Assistant Applications. In *2020 IEEE Security and Privacy Workshops, SP Workshops, San Francisco, CA, USA, May 21, 2020*. IEEE, 76–81. https://doi.org/10.1109/SPW50608.2020.00029

[80] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *Int. J. Softw. Tools Technol. Transf.* 10, 6 (2008), 503–520. https://doi.org/10.1007/s10009-008-0087-9

[81] Andrew Hunt. 1900. *The pragmatic programmer.* Pearson Education India.

[82] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. Automated analysis and debugging of network connectivity policies. *Microsoft Research* (2014), 1–11.

[83] Yan Jia, Luyi Xing, Yuhang Mao, Dongfang Zhao, XiaoFeng Wang, Shangru Zhao, and Yuqing Zhang. 2020. Burglars' IoT paradise: Understanding and mitigating security risks of general messaging protocols on IoT clouds. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 465–481.

[84] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.

[85] T. Jim. 2001. SD3: a trust management system with certified evaluation. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001.* 106–115.

[86] G. Kolaczek. 2003. Specification and verification of constraints in role based access control. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* 190–195. https://doi.org/10.1109/ENABL.2003.1231406

[87] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. 2018. Skill Squatting Attacks on Amazon Alexa. In *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 33–47.

[88] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. 2003. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.* 6, 1 (2003), 128–171. https://doi.org/10.1145/605434.605438

[89] Ninghui Li and John C. Mitchell. 2003. Datalog with Constraints: A Foundation for Trust Management Languages. In *Practical Aspects of Declarative Languages*, Veronica Dahl and Philip Wadler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–73.

[90] Ninghui Li, Mahesh V Tripunitara, and Ziad Bizri. 2007. On mutually exclusive roles and separation-of-duty. *ACM Transactions on Information and System Security (TISSEC)* 10, 2 (2007), 5–es.

[91] Song Liao, Christin Wilson, Long Cheng, Hongxin Hu, and Huixing Deng. 2020. Measuring the Effectiveness of Privacy Policies for Voice Assistant Applications. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 856–869.

[92] Jorge E Luzuriaga, Miguel Perez, Pablo Boronat, Juan Carlos Cano, Carlos Calafate, and Pietro Manzoni. 2015. A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE, 931–936.

[93] Chandrakana Nandi and Michael D. Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, Toby C. Murray and Deian Stefan (Eds.). ACM, 97–102.

[94] K.E. Seamons, M. Winslett, Ting Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and Lina Yu. 2002. Requirements for policy languages for trust negotiation. In *Proceedings Third International Workshop on Policies for Distributed Systems and Networks*. 68–79. https://doi.org/10.1109/POLICY.2002.1011295

[95] Faysal Hossain Shezan, Hang Hu, Gang Wang, and Yuan Tian. 2020. VerHealth: Vetting Medical Voice Applications through Policy Enforcement. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4 (2020), 153:1–153:21. https://doi.org/10.1145/3432233

[96] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. 2012. *Computer security: principles and practice.* Pearson Education Upper Saddle River, NJ, USA.

[97] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web (WWW)*. 1501–1510.

[98] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 361–378. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tian

[99] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. 2015. Cocaine Noodles: Exploiting the Gap between Human and Machine Speech Recognition. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.

[100] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1439–1453.

[101] Moosa Yahyazadeh, Proyash Podder, Endadul Hoque, and Omar Chowdhury. 2019. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*. 61–72.

[102] Yuqing Zhang Yan Jia, Luyi Xing. 2019. Sneak into Your Room: Security Holes in the Integration and Management of Messaging Protocols on Commercial IoT Clouds. Accessed: 2020-08.

[103] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, XiaoFeng Wang, and Yuqing Zhang. 2020. Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1183–1200.

[104] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, XiaoFeng Wang, and Carl A. Gunter. 2018. CommanderSong: A Systematic Approach for Practical Adversarial Voice Recognition. In *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 49–64.

[105] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. 2019. Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1381–1396.

[106] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. 2019. Life after Speech Recognition: Fuzzing Semantic Misinterpretation for Voice Assistant Applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[107] Xinan Zhou, Luyi Xing Jiale Guan, and Zhi Qian. 2022. Perils and Mitigation of Security Risks of Cooperation in Mobile-as-a-Gateway IoT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.

[108] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1296–1310.

APPENDIX

.1 Additional Technical Details for A Few Flawed IoT Policies

Issues with the IoT policy of Govee smart plugs. We found two problems with the Govee policy (Figure 4): (1) The filtering of wildcards specified in the policy is incomplete, and being complete is practically difficult for the manufacturer. Specifically, a malicious Govee user can still subscribe to the topic `+/`, and effectively subscribes to multiple topics of Govee such as `LWT/`, allowing the adversary to receive sensitive messages (e.g., device mac, device name) of all Govee plug users. (2) Once a malicious user obtains the topic *tpc* (or device ID) of the victim's device, subscribing to *tpc* or `GD/[md5 of device ID]` satisfies the policy specification. Notably, obtaining IoT device information such as device ID and its topic is entirely practical, for example, an Airbnb/hotel guest, home visitor, or virtually any user who had at least temporary access to the device — this is also evidenced by the prior research [83, 103]. In our PoC experiment, we confirmed that a (guest) user who ever uses the Govee device can easily obtain the device ID and MQTT topic from the traffic or internal program states of his Govee mobile app.

.2 Discussion of Relation between IAM Policy and IoT Policy

Due to the complicated logical relations between IoT policies and the Cognito and IAM policies (§ 3.2), it is natural to ask why AWS IoT needs separate IoT policies (while other AWS services such as S3, EC2 just leverage the IAM policy). Although never documented by AWS, we found a few key reasons. First, AWS IoT supports three types of identity principles for device or client authentication: (1) X.509 certificate, (2) IAM users, groups, and roles, and (3) Cognito identities. The X.509 certificate is an AWS IoT feature (typically used by IoT devices for authentication to AWS IoT) and is not supported by Cognito/IAM for authentication purposes. That is, Cognito cannot maintain the IAM policy for an X.509 certificate based identity. Second, IAM policies are not designed to support the potentially huge number of IoT end-users of an IoT manufacturer. In particular, under an IoT manufacturer's AWS account, one cannot create more than one thousand different IAM policies [11], considering that the IoT manufacturer may actually want to assign different IAM policies for different users. In contrast, AWS IoT does not have such a limit and allows the IoT manufacturer to create different IoT policies for each IoT user.

```
(( allow,
   action    :   (iot:AttachPolicy,
                  iot:AttachPrinciplePolicy),
   resource  :     * ))
```

Figure 17: IAM policy that allow user to attach IoT policy

.3 Algorithm to Enumerate ISes

P-Verifier uses a simple algorithm to get all *IS*es of a *topic* (see § 4.2). A basic idea is to convert a string problem into a numerical sequence problem, and the syntax of topic ("/" is the delimiter) gives us great convenience. Take topic filter "a/b/x/y" as an example, *P-Verifier* splits the string by "/" and gets a string sequence `[a, b, x, y]`, then we know the length of the sequence is 4 and the basic numerical sequence is `[0, 1, 2, 3]` as subscript. Algorithm 1 shows how to get the subsets of a numerical sequence. Based on the obtained subset of all subscript numeric sequences, all synonyms containing "+" are obtained by replacing the corresponding position with "+". Based on all the ISes with "+", replace the corresponding position with "#" respectively, after string cutting, de-duplication, and merging, we can get all the ISes.

Table 1: Measurement of impact

Vendor	Device Type	Flaw Type	App Downloads	Security & Safety Impact[1]	Privacy Impact and Information Leak
Govee	plug, light	1	1000K+	C, M, F	device id, user id
Onelink	smoke detector	1, 2	10K+	M, F	device id user id, device status, wifi name, device mac, pmesh key
Beurer	air purifier	1	1K+	C, M, F	device id, in-door air quality, power, fan speed, temperature
Belkin WeMo	plug,light	1	1000K+	M, F	device id, device mac, device type, device serial number, device status
SwitchMate	plug, light	1	100K+	C, M	device mac, device version, device status
sun-pro[3]	awning	1, 2	500+	C, M, H	device id, device status, temperature
broil-king[4]	grill	1, 2	5K+	C, M, H	wifi name, lan ip, wan ip, wifi mac, barbecue temperature, fan status, motor status
biobeat	medical	1, 2	700+	C, M, H	Personal medical /health information[2] device id, device mac, app mac, mobile system version
Molekule	air purifier	1, 2	10K+	C, M, H	in-door air quality
NetVue	camera	3	100K+	C	N/A
singlecue	TV	1	5K+	C, M	alexa topic, device mac, control command, serial number
Hippokura	medical	1	5K+	C, M	user id, chat message
SwitchBot	plug	4	100K+	C, M	device id, hardware password
Dyson	air purifier, vacuum	4	1000K+	C, M	device status, control command, in-door air quality

[1] **Security & Safety Impact**: C: Control the device; M: Monitor device activities/status; F: Fake device messages to users; H: Control the vendor's AWS IoT developer account.
[2] **Personal health information leaked**: blood pressure, height, weight, age, calories, steps walked, sleep status (whether the user is sleeping).
[3,4] **Sun-pro and broil-king**: Both broil-king and sun-pro are developed by t2Fi, and they share the same endpoint.

.4 Algorithm of Alphabet-Reducing

As mentioned in § 4.1, the alphabet-reducing algorithm (Algorithm 2) generates a set of automata at a reduced alphabet by decomposing an automaton running on a more inclusive alphabet (see an example with formula 4).

Algorithm 1 Get subsets of numerical sequence

Require: *set* The basic numerical sequence

```
function GetSubSets(set)
    subsets = [[ ]]
    for var in set do
        cache = []
        for item in subsets do
            cache.add(item + [var])
        end for
        subsets.extends(cache)
    end for
    return subsets
end function
```

Algorithm 2 Alphabet-Reducing

```
next_in_set :: (Eq a) => a → [a] → a
next_in_set x (y : ys) = if x == y
    then head ys
    else next_in_set x ys
star_closure :: [Char] → [String]
star_closure as = iterate step ""
    where step s = case splitAt (length s − 1) s of
        (s', "") → s ++ fst_ch_s
        (s', [ch]) → if [ch] == last_ch_s
            then step s' ++ fst_ch_s
            else s' ++ [next_in_set ch as]
        last_ch_s = drop (length as − 1) as
        fst_ch_s = [head as]
check :: String → Bool
check s = and $ map mqtt_ban (zip s (tail s))
mqtt_ban :: (Char, Char) → Bool
mqtt_ban ('*','*') = False
mqtt_ban ('+', y) = y == '/'
mqtt_ban ('#', y) = False
mqtt_ban ('/', y) = True
mqtt_ban (x,'+') = x == '/'
mqtt_ban (x,'#') = x == '/'
mqtt_ban _ = True
alphabets = ['+','#','/','*']
mqtt_topics :: [String]
mqtt_topics = filter check (star_closure alphabets)
```

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect",
        "iot:Subscribe",
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

Figure 18: A policy with simple patterns of wildcard

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iot:Connect"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:client/${iot:Connection.Thing.ThingName}"
      ]
    },
    {
      "Action": [
        "iot:Subscribe"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:topicfilter/${iot:Connection.Thing.ThingName}/*"
      ]
    }
  ]
}
```

Figure 19: A simple policy not overly permissive

.5 Implementation of *P-Verifier*

We have implemented *P-Verifier* in Python with 961 lines of code. *P-Verifier* leverages the off-the-shelf SMT prover Z3. *P-Verifier* leverages the Z3Py APIs [50] (for Python applications) provided by Z3 to invoke the functionalities of Z3 (e.g., And/Or/Not, Regular Expression, Solver). We use Z3 with its String Constraint Solver[49]. Similar to *ZELKOVA*, in case the verification for a policy exceeds a threshold of time (120 seconds), *P-Verifier* also tries the solver CVC4 to re-run the task and reports an "unknown" result if no result is obtained within the same time threshold. Such a case will only happen for unusually complicated IoT policies with many levels of wildcards, which are rare (see the evaluation in § 5). *We released the full source code of P-Verifier online [43].*

Table 2: Comparison with AWS IoT Defender (backed by ZELKOVA [45])

		Flaw1	Flaw2	Flaw3	Flaw4
P-Verifier	FPR	0	0	0	0
	FNR	0	0	0	0
Defender	FPR	0	28.3%	15.6%	0
	FNR	21.1%	0	37.9%	100%

.6 Equations of encoding the lock policies and properties

$$
\begin{aligned}
F_x \doteq\ & (a = \text{“}iot : Subscribe\text{”} \wedge r\ matches\ \text{“}deviceId/.*\text{”}) \vee \\
& (a = \text{“}iot : Publish\text{”} \wedge r\ matches\ \text{“}deviceId/.*\text{”}) \\
F_y \doteq\ & ((a = \text{“}iot : Subscribe\text{”} \wedge r\ matches\ \text{“}deviceId/lowpriv/.*\text{”}) \vee \\
& (a = \text{“}iot : Publish\text{”} \wedge r\ matches\ \text{“}deviceId/lowpriv/.*\text{”}))
\end{aligned} \tag{10}
$$

Device Policy
((allow, action : iot:Connect, resource : client/device-*))
User Policy
((allow, action : iot:Connect, resource : client/user-*))

Figure 20: Example IoT policies for devices and users

$$
\begin{aligned}
F_{pr1} &\doteq \neg(a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}deviceId/highpriv/reset\text{”}) \\
F_{pr2} &\doteq (a = \text{“}iot : Subscribe\text{”} \wedge r \ matches \ \text{“}deviceId/.*\text{”}) \vee \\
&\quad (a = \text{“}iot : Publish\text{”} \wedge r \ matches \ \text{“}deviceId/.*\text{”}) \\
F_{pr3} &\doteq (a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}deviceId/highpriv/reset\text{”}) \vee \\
&\quad (a = \text{“}iot : Publish\text{”} \wedge r = \text{“}deviceId/highpriv/reset\text{”}) \\
F_{pr4} &\doteq (a = \text{“}iot : Subscribe\text{”} \wedge r = \text{“}deviceId/lowpriv/open\text{”}) \vee \\
&\quad (a = \text{“}iot : Publish\text{”} \wedge r = \text{“}deviceId/lowpriv/open\text{”})
\end{aligned} \tag{11}
$$

.7 A high-level scenario to avoid client ID conflict

To manage users and devices in an organization, suppose the devices and users are assigned policies in Figure 20. In this scenario, since the users and devices share the same MQTT broker, one should ensure a user client cannot use an MQTT Client ID that is conflicting with that of a device client [83]. Otherwise, the user (who can be malicious) can force the device offline based on the MQTT protocol [35], and then may fake device messages on behalf of the device [83].

We can leverage *P-Verifier* to verify this high-level security property. By using the same encoding approach as described in §4.2, we can encode the two policies as F_d and F_u, and then leverage the Check 3 to reason about if the resource field of the connect action — determining the Client IDs that are allowed to use — in the device policy shares permissions with that in the user policy (see equation (12)). If the two policies share resources (allowed Client IDs), a user can use the same Client ID as a device.

$$R = F_d \wedge F_u \tag{12}$$

Property pr1
((deny, action : iot:Subscribe, resource : topicfilter/deviceId/highpriv/reset))
Property pr2
Same as policy z1 in Figure 16
Property pr3
((allow, action : iot:Publish, resource : topic/deviceId/highpriv/reset), (allow, action : iot:Subscribe, resource : topicfilter/deviceId/highpriv/reset))
Property pr4
((allow, action : iot:Publish, resource : topic/deviceId/lowpriv/open), (allow, action : iot:Subscribe, resource : topicfilter/lowpriv/open))

Figure 21: Multiple security properties for the lock

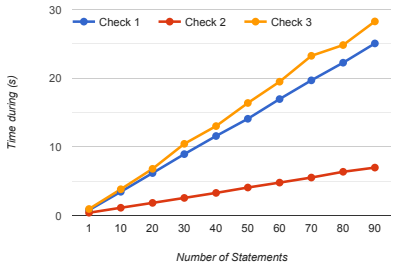

Figure 22: Performance evaluation for setting 1

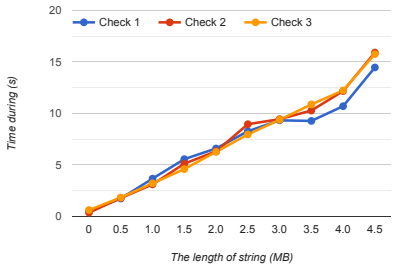

Figure 23: Performance evaluation for setting 2