Checking Information Flow in Cloud-based IoT Access Control Policies

Lorenzo Ceragioli, Letterio Galletta, Edoardo Lunati
IMT School for Advanced Studies Lucca
Lucca, Italy
Emails: lorenzo.ceragioli@imtlucca.it, letterio.galletta@imtlucca.it, edoardo.lunati@imtlucca.it

Artifact Reviewed
10.5281/zenodo.21699813

***Abstract*—Many cloud providers for IoT technologies offer access control mechanisms whose proper configuration is critical for security. However, verifying permissions in isolation is insufficient in a setting where devices have different levels of trust or are compartmentalised in various subsystems. This work analyses IoT access control policies to identify potential security vulnerabilities from unwanted information flow between devices. To this end, we formally model AWS IoT Core's components and define an information flow graph to capture the communication among devices permitted by the access control policies. We build a finite representation of the graph by leveraging an SMT solver, thus enabling the verification of information flow between devices. We implement our approach in a tool called IoT:Poker, and assess it on a realistic scenario and several real-world policies.**

I. Introduction

Cloud computing has emerged as a cornerstone for Internet of Things (IoT) deployments, offering tailored infrastructure and platform services: numerous cloud providers offer specialised Platform-as-a-Service and Infrastructure-as-a-Service solutions for IoT applications, e.g., AWS IoT Core [1] and Azure IoT Hub [2]. These services allow developers to offload security and deployment responsibilities onto cloud providers.

Regarding security, many providers supply developers with languages to define their access control policies (*IoT policies*). An IoT policy is a specification of what resources an IoT device can access, which actions it can perform, and under which conditions (e.g., publish an MQTT [3] message – a type of action from a popular IoT messaging protocol, see Section II). Properly configured policies are essential for the security of IoT systems. Indeed, previous research has highlighted the susceptibility of IoT policies to misconfigurations and the significant risks they may cause [4]. However, preventing unauthorised access is necessary but not sufficient to ensure security in a setting where devices have a different degree of confidentiality, trust and of robustness against attacks. Therefore, developers resort to compartmentalisation and information flow control to increase the security level of their systems. To protect integrity, they assign integrity levels to devices and prevent any direct or transitive dependency that would let lower-integrity devices influence higher-integrity ones ("no read down, no write up"). Similarly, to enforce confidentiality, they assign confidentiality levels to devices and disallow interactions that would allow devices to learn from higher levels ("no read up") or leak to lower levels ("no write down"). Ensuring both integrity and confidentiality requires isolating devices and removing unwanted dependencies. We say that there is an information flow between devices d_1 and d_n if some devices $d_2, d_3 \ldots, d_{n-1}$ exist such that the device d_i sends a message that is received by d_{i+1} [5]–[9].

This paper addresses the problem of checking information flow in IoT access policies, namely, protecting critical devices from untrusted devices and thereby enforcing security properties, including confidentiality and integrity. We provide a formal model that characterises how access to resources in an IoT-Cloud deployment is regulated via policies. Then, we introduce a novel verification procedure to verify that a policy enjoys such information-flow properties and implement it in a tool called IoT:Poker. Formal verification techniques have been proven effective in detecting misconfigurations and verifying security properties in access control policies at various levels. Indeed, prior work [4], [10]–[13] studied the problem of permission misconfiguration in both cloud-based and IoT access policies, focusing on AWS Identity and Access Management (IAM) and AWS IoT Core. However, such works analyse policies in isolation to check the granted permissions and do not consider possible interactions among devices, so the possible information flow (see Section VIII). Here, we fill this gap by providing a triple contribution:

(i) We introduce a formal model of IoT policies, specifically targeting AWS IAM policies and their evaluation within the AWS IoT Core infrastructure.
(ii) Given a set of IoT policies, we analyse them and build an *information flow graph* to capture the possible communication interaction between devices allowed by the policy. The information flow graph captures the counterintuitive behaviour arising from the interplay of wildcard characters and variables, which are resolved at different times during request evaluation and may lead to *wildcard injection attacks*. Checking an information flow thus reduces to reachability in this graph. We compute a finite symbolic version of the graph through an SMT solver to enable practical verification, leveraging the specific constraints of AWS policies to efficiently decide the satisfiability of formulas with regular languages and string manipulation.
(iii) Finally, we implement our verification mechanism in the

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_34

tool IoT:POKER, available online [14]. We evaluate the tool's effectiveness in detecting misconfigurations and unintended information flows using a case study designed to capture a typical Build Automation System. Although synthetic, the case study reflects the main structural constraints and operational patterns of real IoT systems. In addition, we assess the tool's performance by randomly synthesising a network of devices and associating them with real-world IoT policies available online [15]: we show that IoT:POKER scales well in practice as the network grows.

Plan of the paper: In Section II, we introduce the MQTT protocol and AWS IoT Core. Section III present our running example. Section IV formalises AWS IoT Core's main components. In Section V, we describe how to build the information flow graph from a set of policies. Section VI presents our tool and its experimental evaluation, Section VII discusses the assumptions and limitations of our model, and how to extend our proposal to additional mechanisms available in IAM policies. In Sections VIII and IX, we compare our approach with the literature and draw some conclusions. The extended version of this work [16] contains a formal model of device-broker interaction, the proofs of our formal development, and some complementary evaluation results.

II. BACKGROUND

Here, we describe the MQTT protocol and the AWS IoT policy language by focusing only on the main components, which are sufficient for our formal development. Possible extensions and advanced features are discussed in Section VII. We also partially adapt the syntax for brevity, e.g. by omitting the Amazon Resource Name (ARN) from IoT-AWS policies.

MQTT Protocol: Message Queuing Telemetry Transport (MQTT) [3] is a lightweight client/server protocol relying on a publish/subscribe communication model, designed for environments where resources and bandwidth are constrained and limited, such as machine-to-machine and IoT contexts. In the publish/subscribe communication model, the entities sending messages (*the publishers*) and the ones receiving them (*the subscribers*) indirectly interact through the infrastructure provided by a third component (*the broker*), which is the only one responsible for delivering messages. In more detail, clients send `CONNECT` requests to establish a connection with the broker, specifying their chosen client id. After connection, clients communicate via *topics:* virtual communication channels managed by the broker. In practice, topics are hierarchically-structured strings, organized into *topic levels* through forward-slash characters '/'. For example, the topic `floor1/room1/temp` can be used to communicate the sampled temperature in the given room and floor of a building. The broker performs pattern-matching to filter and forward incoming messages to interested clients. A client can subscribe to a given set of topics by sending a `SUBSCRIBE` request to the broker, specifying a *topic filter*, namely a restricted regular expression denoting the set of topics it is interested in. Topic filters can use the single-level '+' or the multi-level '#' wildcard characters: '+' replaces any single topic level, whereas '#' at the end of a topic filter represents any sequence of topic levels. For example, the topic filter `floor1/+/temp` allows a client to receive temperature values from any room on the first floor of a building. For publishing on a given topic, clients send a `PUBLISH` message specifying the topic and a payload; the broker then propagates the payload to all clients subscribed to that topic. For example, a temperature sensor in `room1` can publish on the topic `floor1/room1/temp` to inform all the clients interested in that temperature value.

AWS IoT Policies: Amazon Web Services (AWS) [17] provides cloud services and a communication infrastructure on which IoT developers can deploy their devices, known as AWS IoT Core [1]. Among the various technologies provided by AWS IoT Core, we find the MQTT protocol: the cloud acts as the MQTT broker, while providing security mechanisms via the implementation of authentication and access control. Devices may connect to the broker only if they successfully authenticate through a certificate or via other Amazon authentication services [18], [19]; moreover, even when a client is authenticated, access to the MQTT infrastructure is controlled via a collection of access control policies, which restricts the available topics that any given client may use.

IoT Core allows the definition of different access rights, which correspond to MQTT actions. Here, we only consider the main ones: (i) `iot:Connect` to connect to the cloud infrastructure; (ii) `iot:Publish` to publish on a certain topic; (iii) `iot:Subscribe` to subscribe to some topic filter; (iv) `iot:Receive` to receive the messages published on the specified topics. Note that clients need both the `iot:Receive` and `iot:Subscribe` access rights to receive messages on a given topic. The access rights are specified in a policy (a JSON document) consisting of a set of *policy statements* (access control rules). Each statement describes whether a specific action, e.g. `iot:Publish`, is permitted or not (`Allow` or `Deny`) on a given resource (topic, topic filter, client id). Policy developers can use wildcard characters '?' and '∗' to represent any single character and any sequence of characters, respectively. For example, the following policy:

```
{ "Effect": "Allow",
  "Action": "iot:Subscribe",
  "Resource": [ "floor1/*/temp", "floor1/room1/*" ]},
{ "Effect": "Allow",
  "Action": "iot:Receive",
  "Resource": [ "floor1/*/temp", "floor1/room1/*" ]}
```

grants a client permission to subscribe and receive messages on topics concerning the temperature of any room on the first floor and any sensor in `room1` of the first floor. Moreover, policy statements may use variables, such as `${iot:ClientId}`, which are substituted for attributes associated with the certificate provided during connection. Permissions are then evaluated on the resulting variable-free policy by following a *default-deny* strategy: access is granted if there is at least a statement allowing the requested action and no statement denying it; in any other case, access is rejected.

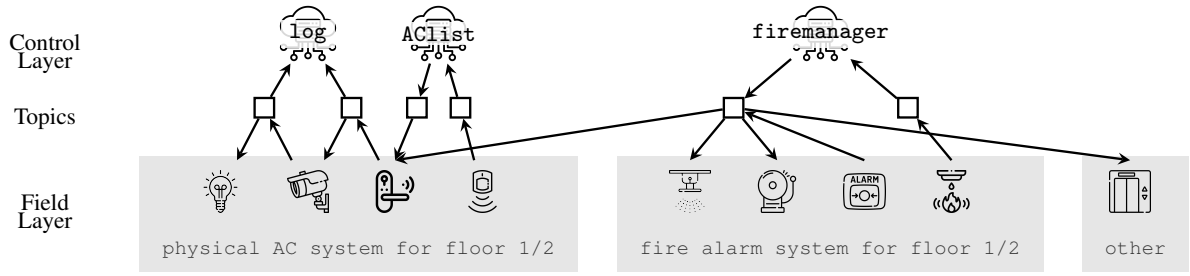

Figure 1. The Architecture of Our Building Automation System.

III. A RUNNING EXAMPLE: A SMART BUILDING

Assume a smart building with two floors, where several subsystems, including energy consumption, physical access control, heating, and fire alert, are all integrated. Our Building Automation System (BAS) is cloud-based, and it is organised in two layers (see Figure 1): the field layer contains off-the-shelf IoT devices that interact with the physical world via sensors and actuators, while the control layer consists of an MQTT broker and several cloud services that implement the communication infrastructure and the control logic. These services execute appropriate actions when triggered by events occurring in the field layer and provide operators with dashboards to monitor, configure, and control the system. Moreover, we group IoT devices in subsystems according to their functionalities. For example, smoke detectors are part of the fire alarm system, whereas badge readers are part of the physical access control system.

The physical access control system includes badge readers, smart locks, presence sensors, and light bulbs. Each floor of the building is expected to behave independently. Consider the first floor as an example. When the badge reader scans a bar code, it publishes a message on the topic `phAC/floor1/bdgReader1/check`. The cloud service `AClist` waits for messages on this topic and checks the code's validity in the message's payload. If the badge is authorised to open the door, `AClist` publishes a message on the topic `phAC/floor1/lock1/open`. When the smart lock `lock1` receives such a message, it unlocks itself and activates the presence sensor of the room by publishing on the topic `phAC/floor1/prsSens1/enable`. When the presence sensor `prsSens1` is active and detects someone inside the room, it publishes on the topic `phAC/floor1/dtdMovement/light1`. The light bulb `light1` is subscribed to this topic and switches itself on upon reception. The service `log` subscribes to all the topics mentioned above to keep a trace of such events.

The fire alarm subsystem consists of smoke sensors, alarm buttons, acoustic alarms, and water pumps. Its expected behaviour for the first floor follows. When a smoke sensor detects some smoke in its area, it publishes a message on the topic `fire/floor1/smokeLvl` that is received by the cloud service `fireMngr` subscribed to the related topic filter. If the level of detected smoke exceeds a given threshold, `fireMngr` publishes on the topic `fire/detected` to raise the fire alarm. The water pumps, acoustic alarms, doors, and elevators all subscribe to this topic. When they receive a message on it, the water pumps and acoustic alarms activate, the doors unlock, and the elevator travels to the ground floor, opens its doors, and halts. The alarm can also be raised by the fire alarm button, which publishes to the same topic.

The IoT policies for each device of the BAS system are available online [14]. We assume that the cloud infrastructure is trusted, while devices may be compromised and policies misconfigured. An attacker can gain full control of vulnerable devices (e.g., a light bulb) and use them to compromise other critical devices and obtain private information. The following sections present our formal model, our verification framework and the tool IoT:POKER, which detects policy misconfiguration and unintended information flows.

IV. A FORMAL MODEL OF IOT ACCESS POLICIES

We take AWS IoT Core as a reference implementation of an IoT-Cloud infrastructure, and we formalise a core of the AWS policy language and its semantics, focusing on the interplay between MQTT and the authorisation mechanism. In doing so, we clarify the counterintuitive semantics of AWS and MQTT wildcards and variable substitution in policy evaluation. Table I contains a summary of symbols and notation used here and in the next section to describe the formal model and its information flow.

Our presentation follows a bottom-up approach.

Definition 1 (Resource)**.** *A resource ρ is either a client id, a topic, or a topic filter. Let Λ be the alphabet of alphanumerical characters enriched with the forward-slash character '/'. Topics are strings over Λ, while client ids and topic filters are strings over $\Lambda \cup \{+, \#\}$.*

We denote with $\mathbb{I}$, $\mathbb{T}$, and $\mathbb{TF}$ the set of client ids, topics, and topic filters, respectively. Note that these sets have a non-empty intersection, so as to model features of the language like substitutions of client ids for `$clientId` variable and matching topics with topic filters. In the following, we see that MQTT wildcards are treated as common characters in client ids, and as special characters when occurring in topic filters, which allows *wildcard injection*. Indeed, although client ids and topic filters have the same syntax, a topic filter tf can be interpreted as a regular expression $tf^{\texttt{mqtt}}$ where the character '#' is interpreted as any sequence of characters in Λ, while the

Table I
SUMMARY OF SYMBOLS AND NOTATION USED IN THE PAPER.

Notation	Description
$\mathbb{I}$, $\mathbb{T}$, $\mathbb{TF}$	the set of client ids, topics, topic filters
$\underline{\texttt{\$clientId}}$	the variable $\${\texttt{iot} : \texttt{ClientId}}$ of AWS
$\mathcal{R}$	A set of resources ρ
$\mathbb{D}$	A set of devices d
Λ	Alphanumerical characters including '/'
$\mathcal{L}(\Lambda)$	Language of topics
$\Lambda \cup \{+, \#\}$	Alphabet of client ids and topic filters
$\mathcal{L}(\Lambda \cup \{+, \#\})$	Language of client ids and topic filters
$\varepsilon \in \{\texttt{Allow}, \texttt{Deny}\}$	Effect of a policy statement
$\alpha \in \{\texttt{Con}, \texttt{Pub}, \texttt{Rec}, \texttt{Sub}\}$	IoT actions
$s = (\varepsilon, \alpha, \mathcal{R})$	Policy statement (Definition 3)
(α, ρ)	Permission (Definition 5)
$\mathbb{C}$	A set of certificates
$\mathbb{P}$	A set of policies
$\varphi \colon \mathbb{C} \to \mathcal{P}(\mathbb{P})$	A map from certificates to policies
$k \colon \mathbb{D} \to \mathcal{P}(\mathbb{C})$	A map from devices to certificates
$(\mathbb{C}, \mathbb{P}, \varphi, \mathbb{D}, k)$	IoT Configuration (Definition 4)
$\mathbb{T}_c^I$, $\mathbb{T}_c^O$	Set of I/O topics (Definition 6)
$G = (\mathbb{C} \cup \mathbb{T}, E)$	Information-flow Graph (Definition 7)
$\psi_{(c,id,\alpha,\rho)}$	Permission predicate (Definition 8)
$\mathcal{O}_c(t)$, $\mathcal{I}_c(t)$	I/O Predicates (Definition 9)
$G_s = (N, E)$	Symbolic Information-flow graph (Definition 10)
$F_{c,c'}$	Topic formula (Definition 10)

character '+' as any sequence of characters different from '/' in Λ. According to MQTT rules, the substitution takes place only if '#' is the terminal character and '#' and '+' appear as topic levels, i.e. they are separated from other characters by '/'. The language of $tf^{\texttt{mqtt}}$, denoted as $\mathcal{L}(tf^{\texttt{mqtt}})$, contains the topics that are matched by the topic filter tf.

Example 1. *The topic filter* $\texttt{floor1}/+/\texttt{temp}$ *matches the topic* $\texttt{floor1/room10/temp}$ *because it is recognized by the regular expression* $\texttt{floor1}/[\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}]^*/\texttt{temp}$*, where* $[\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}]$ *is any alphanumerical character.*

In a policy, resources are specified by strings called resource expressions, which may contain AWS wildcards to compactly represent a set of resources, and the variable $\${\texttt{iot} : \texttt{ClientId}}$ (that we abbreviate as $\underline{\texttt{\$clientId}}$).

Definition 2 (Resource Expression)**.** *A resource expression* $\texttt{Re}$ *for topics, topic filters, or client ids, is a string built on the same alphabet of the given kind of resource extended with the variable* $\underline{\texttt{\$clientId}}$ *and with the AWS wildcards* '?' *and* '$*$'*. A resource expression is* ground *if it does not contain* $\underline{\texttt{\$clientId}}$*.*

Given a resource expression $\texttt{Re}$ and a string v, we write $\texttt{Re}[v/\underline{\texttt{\$clientId}}]$ for the resource expression where every occurrence of $\underline{\texttt{\$clientId}}$ is replaced with v. For example, $\texttt{dtdMovement?}/\underline{\texttt{\$clientId}}[\#/\underline{\texttt{\$clientId}}]$ is $\texttt{dtdMovement?}/\#$. A

```
{ "Effect":"Allow",
  "Action":"iot:Connect",
  "Resource":"*" },
{ "Effect":"Allow",
  "Action":"iot:Receive",
  "Resource":"*" },
{ "Effect":"Allow",
  "Action":"iot:Subscribe",
  "Resource":
      "phyAC/floor?/dtdMovement/${iot:ClientId}" }
```

Figure 2. The policy of the light bulbs in the BAS.

ground resource expression $\texttt{Re}$ can be interpreted as a regular expression $\texttt{Re}^{\texttt{aws}}$ where '?' stands for any character in Λ extended with the MQTT wildcards, and '$*$' for any sequence of such characters. The language $\mathcal{L}(\texttt{Re}^{\texttt{aws}})$ contains the resources that are matched by the resource expression $\texttt{Re}$.

Example 2. *After the device connects with client id* $\texttt{light1}$*, the topic filter* $tf = \texttt{phyAC/floor1/dtdMovement/light1}$ *is a resource matched by the expression* $\texttt{Re} = \texttt{phyAC/floor?/dtdMovement/}\underline{\texttt{\$clientId}}$*, which uses the AWS wildcard '?'. Indeed,* $tf \in \mathcal{L}(\texttt{Re}[\texttt{light1}/\underline{\texttt{\$clientId}}]^{\texttt{aws}})$ *since* $\texttt{Re}[\texttt{light1}/\underline{\texttt{\$clientId}}]$ *can be seen as the regular expression* $\texttt{phyAC/floor}[\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}/+\#]/\texttt{dtdMovement/light1}$.

We now define IoT policies: a policy establishes which client ids can connect and which topics they can publish and/or subscribe to.

Definition 3 (Policy)**.** *A policy* P *is a set of statements* $s = (\varepsilon, \alpha, \mathcal{R})$ *where*

- $\varepsilon \in \{\texttt{Allow}, \texttt{Deny}\}$ *is the effect of the statement;*
- $\alpha \in \{\texttt{Con}, \texttt{Pub}, \texttt{Rec}, \texttt{Sub}\}$ *is an IoT action;*
- $\mathcal{R}$ *is a set of resource expressions.*

The definition of ground and of substitution is naturally extended to policies.

Example 3. *Figure 2 shows the policy of a light bulb in our running example, using the resource expression of Example 2.*

To configure IoT Core, developers define certificates to identify the devices authorised to connect to the broker. Devices are associated with their certificates, and each certificate is associated with its policies. Formally:

Definition 4 (Configuration)**.** *A configuration is a 5-tuple* $(\mathbb{C}, \mathbb{P}, \varphi, \mathbb{D}, k)$ *where*

- $\mathbb{C}$ *is a set of certificates;*
- $\mathbb{P}$ *is a set of IoT policies;*
- $\varphi \colon \mathbb{C} \to \mathcal{P}(\mathbb{P})$ *associates certificates with their policies;*
- $\mathbb{D}$ *is a set of devices;*
- $k \colon \mathbb{D} \to \mathcal{P}(\mathbb{C})$ *associates devices with their certificates.*

We write $\Phi(c) = \bigcup_{P \in \varphi(c)} P$ for the union of c's policies.

Example 4. *Consider the BAS of Section III, the certificates of both the light bulbs are associated with the policy of Figure 2, i.e.,* $\varphi(c_{\texttt{light1}}) = \varphi(c_{\texttt{light2}})$.

We now introduce how permission requests are evaluated. Intuitively, the broker interrogates the access control system to check if the action requested by the message is allowed. To do that, the access control system checks whether there exists a policy statement s in P forbidding the request, i.e., with the same action of the request, with effect `Deny`, and with the resource ρ matched by a resource expression in $\mathcal{R}$ (after expanding AWS wildcards). If a match is found, the request is rejected; otherwise, the system looks for a statement s that explicitly allows the request. If there is at least one match, the request is accepted. If no matching statement is found, the request is denied (*implicit deny*).

Definition 5 (Permission)**.** *A permission is a pair (α, ρ) where α is an IoT action to be performed over a resource ρ. A permission (α, ρ) is granted by the* ground *policy P, in symbols $P \vdash\sim (\alpha, \rho)$, if and only if both the following hold:*

- $\{(\texttt{Allow}, \alpha, \{\texttt{Re}\} \cup \mathcal{R})\}$ *exists in P with $\rho \in \mathcal{L}(\texttt{Re}^{\texttt{aws}})$;*
- *for all $(\texttt{Deny}, \alpha, \mathcal{R}) \in P$ and $\texttt{Re} \in \mathcal{R}$, $\rho \notin \mathcal{L}(\texttt{Re}^{\texttt{aws}})$.*

We can now detail how an attacker can break desired security properties by exploiting policy misconfigurations. In more detail, we assume that the cloud provider (thus, the broker and authorisation mechanisms) is trusted and safe from tampering. On the contrary, devices may be compromised by attackers, and IoT policies may be misconfigured and subverted. In addition, we assume that the attacker knows the configuration of the system, and it can take full control of a device d, thus gaining the capability of authenticating with all the certificates in $k(d)$. The compromised device d ignores its original scope, and it maliciously selects both the client id to connect with and the topics over which to communicate. The attacker's objective is to compromise either *confidentiality* or *integrity* of the system. When breaking confidentiality, the attacker gains private information, e.g. directly from some sensor d, or indirectly from other devices d' receiving messages from d. Conversely, the integrity of a device d is compromised when its behaviour is influenced by the attacker, possibly leveraging an intermediate device that propagates the attacker's messages.

We conclude by clarifying how the *wildcard injection attack* works. MQTT wildcards are not considered as reserved characters when used in client ids, but they are substituted verbatim for $\underline{\texttt{\$clientId}}$. During connection, attackers can use a string containing a wildcard as their client id, thus injecting it into the policies of the broker, possibly bypassing `Deny` rules and breaking security guarantees. For example, consider a certificate c and a device d with $c \in k(d)$, and assume that:

$$\Phi(c) = \{(\texttt{Allow}, \texttt{Con}, \{*\}), (\texttt{Deny}, \texttt{Con}, \{\texttt{private}\}), (\texttt{Allow}, \texttt{Rec}, \{*\}), (\texttt{Allow}, \texttt{Sub}, \{/\underline{\texttt{\$clientId}}\})\}$$

The policy should allow every device to receive on its topic (named after its client id), while forbidding them from using the topic `/private`, which is reserved for protected information. At first glance, eavesdropping seems impossible. However, assume d connects to the broker by identifying as $\#$. Since $\# \in \mathcal{L}(*^{\texttt{aws}})$, the connection permission is granted, and the last statement of $\Phi(c)$ is instantiated as $(\texttt{Allow}, \texttt{Sub}, \{/\#\})$. Note that the instantiated policy allows d to subscribe to the topic filter $/\#$, where the character '$\#$' of the client id is interpreted as an MQTT wildcard. Thus, the device can receive all the messages published on the topic `/private`.

In the next section, we make this formal by giving a graph representation of the possible paths along which messages can propagate through devices and topics.

V. Characterising Information Flow

We characterise the information flow permitted by an IoT configuration, i.e. which devices can influence each other. The underlying idea is to build a graph whose nodes represent devices and topics: there is an arc from a device d to a topic t if a policy associated with the device allows it to publish on t; vice versa there is an arc from t to d if a policy allows d to subscribe and receive messages on t. Then, we present a symbolic representation of this graph, enabling the practical verification of security properties.

A. Information Flow Graph

It is convenient to introduce some auxiliary definitions to facilitate the construction of our graphs. For each certificate c, we define $\mathbb{T}_c^I$ as the set of topics from which the devices connected and authenticated using certificate c are permitted to receive messages. Similarly, $\mathbb{T}_c^O$ collects topics to which c may publish messages.

Definition 6 (I/O Sets)**.** *Given a configuration, the input and output sets of certificate c are the sets of topics $\mathbb{T}_c^I$ and $\mathbb{T}_c^O$:*

$$\begin{aligned}\mathbb{T}_c^I = \{t \mid\ & \Phi(c)[\,^{id}/_{\underline{\texttt{\$clientId}}}] \vdash\sim (\texttt{Con}, id),\\ & \Phi(c)[\,^{id}/_{\underline{\texttt{\$clientId}}}] \vdash\sim (\texttt{Rec}, t),\\ & \Phi(c)[\,^{id}/_{\underline{\texttt{\$clientId}}}] \vdash\sim (\texttt{Sub}, tf)\\ & \text{and } t \in \mathcal{L}(tf^{\texttt{mqtt}}),\ \textit{for some}\ id\ \textit{and}\ tf\}\\ \mathbb{T}_c^O = \{t \mid\ & \Phi(c)[\,^{id}/_{\underline{\texttt{\$clientId}}}] \vdash\sim (\texttt{Con}, id)\\ & \textit{and}\ \Phi(c)[\,^{id}/_{\underline{\texttt{\$clientId}}}] \vdash\sim (\texttt{Pub}, t),\ \textit{for some}\ id\}\end{aligned}$$

Roughly speaking, the conditions on the first set require c to be able to connect with client id id, to subscribe to the topic filter tf, and that topic t belongs to the language of tf. The conditions on the second set, instead, require that c can connect with client id id and can publish over t.

Example 5. *Consider the light bulb* `light2` *and the topic* $t =$ `phAC/floor1/dtdMovement/light1`*. It holds that* $t \in \mathbb{T}^I_{c_{\texttt{light2}}}$ *because: (i) the light bulb can connect to the broker with client id $\#$, and the substitution gives the policy*

$$P' = \{(\texttt{Allow}, \texttt{Con}, \{*\}), (\texttt{Allow}, \texttt{Rec}, \{*\}), (\texttt{Allow}, \texttt{Sub}, \{\texttt{phAC/floor?/dtdMovement/\#}\})\}$$

(ii) the ground policy allows the client to receive messages on t and to subscribe to the topic filter $tf =$ `phAC/floor1/dtdMovement/#`*; and (iii) the topic t is a string in the language of tf when considering MQTT wildcards. Note that this is a wildcard injection attack: the second light bulb receives private information intended for another device.*

We now define information flow graphs of configurations:

Definition 7 (Information Flow Graph)**.** *Given a configuration, its information flow graph is a bipartite directed graph* $G = (\mathbb{D} \cup \mathbb{T}, E)$ *with* $\mathbb{D}$ *the set of devices,* $\mathbb{T}$ *of all possible topics, and* $E \subseteq (\mathbb{D} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{D})$ *of the arcs defined as*

$$E = \{(d,t) \mid \exists c \in k(d) \text{ s.t. } t \in \mathbb{T}_c^O\} \cup \{(t,d) \mid \exists c \in k(d) \text{ s.t. } t \in \mathbb{T}_c^I\}$$

An arc of G represents a communication permitted by the configuration. Following a path of G, we can track all the possible device interactions, including indirect ones. We say that there is a (possible) information flow from d to d' if d' is reachable from d in the information flow graph, meaning that information produced by d may affect d'.

Example 6. *Assume that you want the information on* `lock1` *to be private to the physical access control of the first floor in our running example. This requirement can be checked by inspecting the information flow graph of the configuration in [14]. Indeed, the following path is a violation* `lock1` $\rightarrow t_1 \rightarrow$ `prsSens1` $\rightarrow t_2 \rightarrow$ `light2` *where* t_1 = `phAC/floor1/prsSens1/enable` *and* t_2 = `phAC/floor1/dtdMovement/light1`.

As a final remark, recall that we do *not* identify a client with its client id. This is because only the association with the certificate is authenticated by the Cloud, whereas the client id is freely selected by the device. A compromised device might exploit this flexibility by using multiple ids or MQTT wildcards to maximise the attack surface.

In the extended version of this work [16], we assess the adequacy of the information flow graph in capturing how information propagates in IoT systems. We give an operational semantics describing the effects of MQTT requests on message transmission between devices and on the broker state. Then, we characterise information flow as sequences of messages that may arise in some feasible execution, and we prove it consistent with the paths of the information flow graph built from the configuration.

B. Symbolic Information Flow Graph

In general, the number of nodes and arcs of the information flow graph grows exponentially with respect to the length of topics (256 bytes of UTF-8 characters in the current implementation of AWS IoT Core). Clearly, building such a graph is impractical. To address this issue, we build a symbolic version where sets of topics are represented by logical formulas.

We define predicates representing the conditions under which a permission may be granted during some execution.

Definition 8 (Permission Predicate)**.** *Given a 4-tuple* (c, id, α, ρ) *where* c *is a certificate,* α *is an IoT action, and* ρ *is a resource, we let the* permission predicate $\psi_{(c,id,\alpha,\rho)}$ *be defined as follows, where* P *is* $\Phi(c)[^{id}/_{\texttt{\$clientId}}]$*:*

$$\psi_{(c,id,\alpha,\rho)} := \Big(\bigvee_{(\texttt{Allow},\alpha,\mathcal{R}) \in P} \bigvee_{\texttt{Re} \in \mathcal{R}} \rho \in \mathcal{L}(\texttt{Re}^{\texttt{aws}}) \Big) \wedge \neg \Big(\bigvee_{(\texttt{Deny},\alpha,\mathcal{R}) \in P} \bigvee_{\texttt{Re} \in \mathcal{R}} \rho \in \mathcal{L}(\texttt{Re}^{\texttt{aws}}) \Big)$$

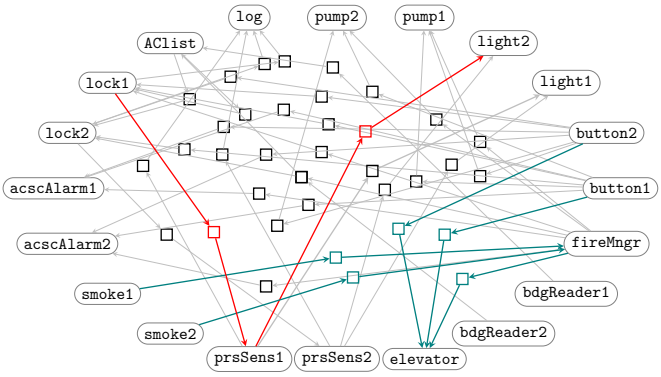

Figure 3. Information Flow Graph for the Building Automation System.

We introduce the auxiliary predicates $\mathcal{I}_c(t)$ and $\mathcal{O}_c(t)$, mimicking input and output sets in a symbolic setting.

Definition 9 (I/O Predicates)**.** *Given a configuration, the input and output predicates* $\mathcal{I}_c$ *and* $\mathcal{O}_c$ *of a certificate* c *are:*

$$\mathcal{I}_c(t) = \exists\, id, tf.\, \psi_{(c,id,\texttt{Sub},tf)} \wedge \psi_{(c,id,\texttt{Rec},t)} \wedge \psi_{(c,id,\texttt{Con},id)} \wedge t \in \mathcal{L}(tf^{\texttt{mqtt}});$$
$$\mathcal{O}_c(t) = \exists\, id\, .\psi_{(c,id,\texttt{Pub},t)} \wedge \psi_{(c,id,\texttt{Con},id)}$$

Intuitively, the symbolic version of the information flow graph has a node for each device, and an additional node for each pair of certificates that can communicate. More precisely, for each pair of certificates c and c', we build a logic formula $F_{c,c'}$ that is true if and only if c can publish a message that can be read by (devices authenticated with) c'. If the formula is valid, we add a node $F_{c,c'}$, and we connect it with an arc from d to $F_{c,c'}$ for each device such that $c \in k(d)$. Similarly, we add an arc from $F_{c,c'}$ to d' if $c' \in k(d')$. (see the next section for the algorithm to compute the symbolic graph).

Definition 10 (Symbolic Information Flow Graph)**.** *The symbolic information flow graph* G_s *of a configuration is* (N, E)*:*

$$N = \mathbb{D} \cup \{F_{c,c'} \mid c, c' \in \mathbb{C} \text{ and } F_{c,c'} \text{ is true}\}$$
$$E = \{(d, F_{c,c'}) \mid c \in k(d)\} \cup \{(F_{c,c'}, d') \mid c' \in k(d')\}$$

where for each c *and* c'*,* $F_{c,c'} = \exists t. \mathcal{O}_c(t) \wedge \mathcal{I}_{c'}(t)$ *is a logic formula over the (first-order) theory of regular expressions.*

We say that there is a (possible) symbolic information flow from d to d' if d' is reachable from d in the symbolic information flow graph G_s.

Example 7. *Figure 3 contains the symbolic information flow of our BAS running example. The red path is* `lock1` $\rightarrow F_{c_{\texttt{lock1}},c_{\texttt{prsSens1}}} \rightarrow$ `prsSens1` $\rightarrow F_{c_{\texttt{prsSens1}},c_{\texttt{light2}}} \rightarrow$ `light2`*. The violating path of Example 6 is one of its instances where the topics* t_1 *and* t_2 *satisfy the formula* $F_{c_{\texttt{lock1}},c_{\texttt{prsSens1}}}$ *and* $F_{c_{\texttt{prsSens1}},c_{\texttt{light2}}}$*.*

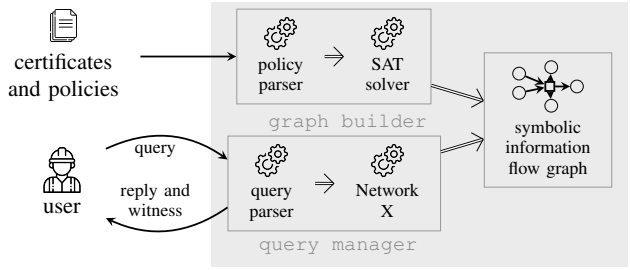

Figure 4. Overview of IoT:Poker.

The following theorem establishes that the symbolic information flow graph provides a sound and complete representation of the information flow graph which in turn is proved to match the information flow permitted by the configuration according to the operational semantics of the MQTT broker given in the extended version of this work [16].

Theorem 1. *Given a configuration, there is an information flow from d to d' if and only if there is a symbolic information flow from d to d'.*

The next section presents the implementation of our tool for checking information flow in IoT systems. However, we highlight a crucial point: the symbolic information flow graph cannot be computed solely through operations on regular languages (as one might initially assume, given that both AWS and MQTT wildcards can be described using regular language operators). The issue arises from the substitution of the variable $\underline{\texttt{\$clientId}}$ within the policy rules that complicates the task. In particular, $\mathbb{T}_c^I$ and $\mathbb{T}_c^O$ are not always regular languages. For example, consider a certificate c with

$$\Phi(c) = \{(\texttt{Allow}, \texttt{Con}, *), (\texttt{Allow}, \texttt{Pub}, \underline{\texttt{\$clientId}}/\underline{\texttt{\$clientId}})\}.$$

The output set of topics over which c can publish $\mathbb{T}_c^O = \{\omega/\omega \mid \omega \in (\Lambda \cup \{/\})^*\}$ is *not* a regular language, as it contradicts the pumping lemma [20].

VI. Implementation and Evaluation

We implemented our verification approach in IoT:Poker [14], an open-source AWS IoT-policy analyzer written in Python. Its architecture is shown in Figure 4: the tool takes a configuration as input, parses the policies using the library Policy Universe [21], and uses the SMT solver cvc5 [22] to construct the symbolic information-flow graph. Finally, a query manager answers users' requests by checking reachability in the graph via the NetworkX library [23].

The following subsections first describe how the symbolic information flow graph is built; then clarify which queries can be performed on the resulting graph; and finally, evaluate the effectiveness and scalability of IoT:Poker.

A. Building the graph

Algorithm 1 describes the procedure to build the graph. Given a configuration $(\mathbb{C}, \mathbb{P}, \varphi, \mathbb{D}, k)$, we start from a symbolic graph that has a node for each device $d \in \mathbb{D}$ and no arcs. Then,

Algorithm 1 Information Flow Graph Construction.

Input: a configuration $(\mathbb{C}, \mathbb{P}, \varphi, \mathbb{D}, k)$
Output: its symbolic information flow graph

$(N, E) \leftarrow (\mathbb{D}, \emptyset)$
for all $(c, c') \in \mathbb{C} \times \mathbb{C}$ **do**
  build $F_{c,c'}$ as in Definition 10
  check its satisfiability through Algorithm 2
  if $F_{c,c'}$ is satisfiable **then**
    $N \leftarrow N \cup \{F_{c,c'}\}$
    for all d such that $c \in k(d)$ **do**
      $E \leftarrow E \cup \{(d, F_{c,c'})\}$
    for all d' such that $c' \in k(d')$ **do**
      $E \leftarrow E \cup \{(F_{c,c'}, d')\}$
return (N, E)

Algorithm 2 Witness for the predicate $F_{c,c'}$

Input: two certificates c, c'
Output: a witness $w = (id, id', t, tf)$ or None

$S \leftarrow \psi_{(c,id,\texttt{Con},id)} \wedge \psi_{(c',id',\texttt{Con},id')} \wedge \psi_{(c,id,\texttt{Pub},t)}$
  $\wedge\ \psi_{(c',id',\texttt{Rec},t)} \wedge \psi_{(c',id',\texttt{Sub},tf)}$
if SMT(S) is unsat **then**
  return None ▷ Early Fail
$S_e \leftarrow S \wedge (t = tf)$
if SMT(S_e) is sat **then**
  return $w \leftarrow (id, id', t, tf)$ ▷ Early Success
for $n = 1, \ldots, 8$ **do**
  $S_h \leftarrow S \wedge t = t_1/t_2/\ldots/t_n$
  $S'_h \leftarrow tf \in \mathcal{L}((t_1|+)/\ldots/(t_n|+))$
  for $m = 1, \ldots, n-1$ **do**
    $S'_h \leftarrow S'_h \vee tf \in \mathcal{L}((t_1|+)/\ldots/(t_m|+)/\#)$
  if SMT$(S_h \wedge S'_h)$ is sat **then**
    return $w \leftarrow (id, id', t, tf)$
return None

we iterate over each pair of certificates c and c' of $\mathbb{C}$, building the predicate $F_{c,c'}$ of Definition 10, encoding it into an SMT formula (via Skolemization), and invoking the solver: if the formula is satisfiable, then we add a node for $F_{c,c'}$, and we connect it with all the devices associated with c (incoming arc) and c' (outgoing arc).

The core challenge is efficiently deciding the SMT problem, which amounts to finding two client ids (id, id'), a topic (t) and a topic filter (tf) that satisfy the formula $\Psi(id, id', t, tf)$ below (which encodes $\mathcal{O}_c$ and $\mathcal{I}_{c'}$), or proving it unsatisfiable.

$$\Psi(id, id', t, tf) = \psi_{(c,id,\texttt{Con},id)} \wedge \psi_{(c',id',\texttt{Con},id')} \wedge$$
$$\psi_{(c,id,\texttt{Pub},t)} \wedge \psi_{(c',id',\texttt{Rec},t)} \wedge \psi_{(c',id',\texttt{Sub},tf)} \wedge t \in \mathcal{L}(tf^{\texttt{mqtt}})$$

The predicates can be encoded by resorting to the strings and regular expressions theories, but the presence of two different kinds of wildcards (AWS and MQTT) in the policies makes the problem challenging in practice: the solver would take hours and hours of computation to check the satisfiability of a single predicate $F_{c,c'}$, if a naive encoding of the formula is used. For

solving this problem, we implement an effective strategy to invoke the solver in Algorithm 2, which is divided into three stages. The first two stages are heuristics for early failure and success, while the last one is a complete decision procedure, optimised for dealing with the potency of wildcards by taking advantage of the peculiar structure of the $F_{c,c'}$ formulas.

Our heuristics are based on checking weakened and strengthened versions of Ψ, where the constraint $t \in \mathcal{L}(tf^{\texttt{mqtt}})$ is approximated by simpler conditions. In the first stage, the algorithm considers a set of conditions that are necessary but not sufficient for guaranteeing communication between c and c'. The constraint $t \in \mathcal{L}(tf^{\texttt{mqtt}})$ is removed and the solver is asked to simply determine if there exists some assignment that satisfies at least the $\psi_{\_}$ predicates of Ψ. If this is not the case, then communication is trivially impossible, and an *early fail* occurs (line 4). Note that these conditions are not sufficient for communication because we are imposing no relation between the topic filter tf and the topic t. The second stage checks a condition that is sufficient but not necessary for communication: whether c' can subscribe to the same topic t on which c can publish (line 7). Basically, we ignore the MQTT wildcards and run the solver on a version of Ψ where $t = tf$ substitutes $t \in \mathcal{L}(tf^{\texttt{mqtt}})$. Although simple, these two heuristics greatly improve our tool's performance, solving most real-world cases.

If neither an early failure nor a solution is found, we adopt the complete resolution strategy (*hard strategy*). This third stage considers Ψ as it is, also taking MQTT wildcard characters into account. Finding an efficient SMT encoding of $t \in \mathcal{L}(tf^{\texttt{mqtt}})$ is made particularly challenging by the fact that tf is not a constant value, but it is rather dynamically computed as a solution of the other constraints. To address this problem, we focus on the peculiar format of IoT resources, exploiting the hierarchical structure of topics $t = t_1/t_2/\ldots/t_n$ and topic filters $tf = tf_1\,/\,tf_2\,/\ldots/\,tf_m$, which have a maximum depth of 8 in AWS. The algorithm essentially creates a disjunction of constraints encoding alternative formats for t and tf, and solves it with a specific search strategy. In more detail, we start from the assertions of line 2 about the $\psi_{\_}$ predicates of Ψ, and we add the condition of line 9 stating that t has n levels. Finally, we impose $t \in \mathcal{L}(tf^{\texttt{mqtt}})$ by requiring t and tf to satisfy one of the following:

- the number of levels of t and tf coincides (i.e., $n = m$) and, tf_i matches t_i for each level i, i.e., $tf_i = t_i$ or $tf_i = +$ (line 10, recall that $+$ matches any single level of the topic);
- tf has no more levels than t (i.e., $m \leq n$), tf_i matches t_i for all $i < m$, and the last level of the topic filter is $tf_m = \#$ (line 12, recall that $\#$ matches any remaining sequence of levels $t_m/t_{m+1}/\ldots/t_n$ of the topic t).

For example, the fist case tells us that the topic $t = \texttt{phAC/floor1/prsSens1/enable}$ is included in $\mathcal{L}(tf_1^{\texttt{mqtt}})$ with $tf_1 = \texttt{phAC/+/prsSens1/+}$, because each layer of tf_1 is either equal to the corresponding one of t or it is the wildcard character $+$, which matches both `floor1` and `enable` (as well as any other alphanumerical string). Moreover, the second case tells us that the topic above is also included in $\mathcal{L}(tf_2^{\texttt{mqtt}})$ with $tf_2 = \texttt{phAC/\#}$, because the wildcard character $\#$ matches `floor1/prsSens1/enable` (as well as any other string over the alphabet of alphanumerical characters enriched with the forward-slash character '/').

Our optimisations allow Algorithm 2 to effectively check the satisfiability of the formula $F_{c,c'}$, as shown by our experiments below. An upper bound to the complexity of computing the symbolic graph is given by the following theorem:

Theorem 2 (Complexity)**.** *Let $(\mathbb{C}, \mathbb{P}, \varphi, \mathbb{D}, k)$ be a configuration, and let $f(n)$ be the cost for deciding the satisfiability of quantifier-free predicates of size n using regular expressions and string theories. Then, the time required by Algorithm 1 is at most $O(|\mathbb{C}|^2 \cdot (|\mathbb{D}| + |\overline{P}| \cdot |\overline{\mathcal{R}}| + f(|\overline{P}| \cdot |\overline{\mathcal{R}}|)))$, where $|\overline{P}|$ and $|\overline{\mathcal{R}}|$ are the size of the largest policy and resource expression.*

Note that f is at least exponential, independently of the used theories, as the problem subsumes SAT, for which only exponential solutions are known. However, the exponential cost is only on the size of policies and on the number of resource expressions in statements, while the algorithm is polynomial on the size of the network (our experimental results are consistent with this estimation). Noticeably, the size of the network is expected to increase more rapidly than the complexity of the individual policies in practice (as far as policies are adequately designed).

B. Query Manager

IoT:POKER allows users to define security labels, i.e. names $\texttt{s}, \texttt{s}', \texttt{s}'', \ldots$ for sets of devices that play similar roles in the IoT system. After that, IoT:POKER can check four kinds of queries on IoT systems. The first verifies the existence of a permitted information flow, while the others correspond to the security properties of integrity, confidentiality, and isolation. The query $\texttt{flow}(\texttt{s}, \texttt{s}')$ checks whether there is a permitted information flow from a device labelled with $\texttt{s}$ to a device labelled with $\texttt{s}'$. The query $\texttt{onlyAffects}(\texttt{s}, \texttt{s}')$ checks integrity, i.e. it requires that only a device labelled with $\texttt{s}$ can influence the behaviour of a critical device labelled with $\texttt{s}'$. The query $\texttt{onlyKnows}(\texttt{s}, \texttt{s}')$ checks confidentiality, i.e. it requires that only devices labelled $\texttt{s}$ can receive private information produced by devices labelled $\texttt{s}'$. Finally, the query $\texttt{isolated}(\texttt{s}, \texttt{s}')$ requires that s-labelled devices and s'-labelled devices cannot interact.

By Theorem 1, query evaluation reduces to reachability in the symbolic graph: each query can be solved by standard search algorithms, with worst-case performance linearly proportional to the number of edges of the graph. More specifically, $\texttt{flow}(\texttt{s}, \texttt{s}')$ (and $\texttt{onlyKnows}(\texttt{s}', \texttt{s})$) are evaluated by visiting the graph from the nodes associated with s-labelled devices, and checking if at least one of (respectively, all) the visited nodes corresponds to an s'-labelled device. The same approach applies to $\texttt{onlyAffects}(\texttt{s}, \texttt{s}')$, but with the edges reversed. Finally, $\texttt{isolated}(\texttt{s}, \texttt{s}')$ requires two traversals of the graph for checking that s cannot reach s' and vice-versa.

Table II
ANALYSING THE BAS WITH IOT:POKER.

Query	Result	Counterexample
`isolated(phyAC1, elevator)`	✓	
`onlyAffects(fire, elevator)`	✓	
`onlyKnows(phyAC1, lock1)`	✗	`lock1; prsSens1; light2`

Table III
IOT:POKER SCALABILITY OVER REAL-WORLD CONFIGURATIONS.

		nodes	hard strategies	building time (s)			query time (s)
		avg.	avg.	min.	avg.	max.	avg.
certificates	20	264.6	1.8	0.42	2.87	12.00	0.0060
	40	973.0	4.5	2.30	6.28	16.55	0.0086
	60	2211.2	5.6	2.93	9.22	23.39	0.0108
	80	3903.9	13.6	5.93	16.34	29.08	0.0136
	100	6098.1	16.7	7.50	19.46	36.55	0.0167
	120	8764.2	21.5	13.58	24.69	41.49	0.0198
	140	12080.1	28.7	18.61	31.06	46.63	0.0244
	160	15415.9	29.6	20.03	33.01	44.43	0.0280
	180	19571.7	37.0	26.61	37.87	52.79	0.0318
	200	24209.9	45.8	31.62	46.06	57.34	0.0349
	220	29066.2	50.5	32.39	50.93	62.02	0.0393
	240	34732.6	59.0	42.11	57.68	62.41	0.0430
	258	40019.0	65.0	62.35	62.67	63.53	0.0473

C. Effectiveness Evaluation

We first evaluate the effectiveness of IOT:POKER using the case study of Section III. Due to the lack of open-source configurations for multi-device IoT systems, we constructed a representative BAS that captures the structural rules, constraints, and communication patterns of typical IoT scenarios.

We assume that each device is configured with a known IoT policy. The complete configuration is available in the online repository [14] and includes 17 devices, 20 certificates, and 15 IoT policies, each of which consists of approximately 20 lines of code. Our threat model assumes that the cloud infrastructure is trusted, whereas devices may be vulnerable and IoT policies may be misconfigured. An attacker may gain full control of a vulnerable device, such as a light bulb, and maliciously select the topics to which it publishes or subscribes, exploiting overly permissive policies to compromise critical devices or access sensitive information. Security labels identify the subsystem to which each device or cloud service belongs. For example, `phyAC1` labels the devices involved in the physical access control of the first floor, such as `bdgReader1`, together with the related cloud services `log` and `AClist`. Similarly, `fire` labels the fire-alarm manager, the alarm buttons, and the smoke sensors on both floors.

We use IOT:POKER to analyse the IoT policies, reconstruct the permitted device interactions, and represent them in the information-flow graph shown in Figure 3. Table II reports the queries considered in our evaluation and the corresponding results produced by IOT:POKER. The first query evaluates an *isolation* property: the elevator must be completely isolated from devices in `phyAC1` subsystem. IOT:POKER verifies that no path violates this requirement: no path starts from the elevation, and the ones reaching it – highlighted in **teal** in Figure 3 – do not come from `phyAC1`. The second query evaluates the *integrity* of the elevator, whose behaviour should depend only on devices labelled `fire`. The property is satisfied: **teal** paths originate from the intended devices. The third query, `onlyKnows(phyAC1, lock1)`, evaluates *confidentiality*: information produced by the first-floor lock should be accessible only to devices and services labelled `phyAC1`. IOT:POKER detects a violation, as a compromised light bulb can receive messages intended for other light bulbs, allowing information originating from `lock1` to reach `light2` on the second floor. This unintended flow results from the overly permissive policy for the light bulbs, which is shown in Figure 2.

D. Scalability Evaluation

We experimentally evaluate the scalability of IOT:POKER on real-world configurations when the IoT configuration's size (number of certificates) grows, using cvc5 as our SMT solver. We conduct all our experiments below on a desktop machine with an i7-10700K processor (3.80GHz) and 32GB RAM, running Windows 11. We consider 258 real-world policies originally implemented for various IoT devices by different manufacturers, available online in the benchmark of the P-verifier tool [15]. Using these policies, we performed 30 experiments as follows: (i) we generate 13 configurations with an increasing number of devices from 20 to 258; (ii) each device is randomly associated with a distinct certificate, real-world policy (from the benchmark), and a fresh security label; (iii) we build the symbolic information flow graph of each configuration, measuring its computation time; (iv) we interrogate IOT:POKER with 1000 reachability queries between two random devices, and record the response time.

The results are in Table III, where the first column indicates the size of the IoT configuration (i.e., the number of certificates), the second reports the number of nodes of the graph (both devices and symbolic nodes representing sets of topics), the third one is the number of times the hard strategy was used to determine if two devices communicate, and the last two columns contain the time needed for building the graph and solving the queries. For each configuration size, the table reports the arithmetic mean of the results obtained by the configurations of that size, across the 30 experiments. Moreover, for the graph build time, which is the most expensive part of the computation, we also mention the fastest and slowest runs. When considering all the 258 real-world policies of the benchmark, IOT:POKER builds the symbolic graph in less than 64 seconds. Moreover, the time required to build the graph increases linearly with the number of times the algorithm resorts to the hard strategy. In more detail, the time needed to solve the SMT problem with the hard strategy is 0.2589 seconds on average (calculated across all 258 policies), and the worst case is 8.7353 seconds; early fail and success take 0.0220 and 0.0122 seconds on average instead, with worst cases 1.2852 and 0.0905 seconds, respectively. While running Algorithm 2 on all 258 policies, early failure occurs

90.4% of the time, early success 7.1%, and hard strategy 2.5%. The time needed for answering each slot of 1000 queries is less than 0.1 seconds on average. In conclusion, the algorithm scales well up to configurations with more than 250 devices: the one-time effort for building the graph is reasonable, and evaluating queries is almost immediate.

A version of IoT:POKER also exists, which employs Z3 [24] as SMT solver in place of cvc5. We repeated our experiments with Z3, using the same configurations generated for cvc5. The detailed experimental results are in the extended version of this work [16], from which it emerges that cvc5 performs significantly better than Z3.

VII. Discussion

The model presented in Section IV focuses on the core communication mechanisms of MQTT and the main components of AWS IoT Core policies. While this modeling choice keeps the formal treatment tractable, it omits certain protocol and platform features available in practice that we do not capture. For example, we do not consider wildcard matching in ARN components other than the resource field, such as the region, or the possibility of temporary credentials. In this section, we discuss how the formalization can be extended to capture and account for some of these features when verifying information-flow properties.

Among the advanced features of AWS IoT policies that we omitted, *things* and *conditions* are those that we can incorporate more naturally by extending our model. Policy developers can define named virtual devices, called *things*, in their configurations, which are associated with a set of custom attributes, e.g., geographical location, source IP address or domain name. Formally, a thing can be represented as a set of bindings from attribute names to strings which may be used in resource expressions and substituted with their values when the policy is instantiated, similarly to what is done for $\texttt{\$clientId}$. For example, in our BAS scenario of Section III one may define the attributes $\texttt{\$floor}$ and $\texttt{\$room}$, and add the resource expression $\texttt{floornum}\texttt{\$floor}/\texttt{roomnum}\texttt{\$room}/\texttt{light?}$ in the policy. As a result, two things that associate different values with the attributes above may publish on disjoint sets of topics, even if they share the same policy. Our formal model can be updated by considering an additional special case for things that are associated with connecting devices during certificate authentication, and a specific certificate is either for things or for common devices; hence, the kind of instantiation to use is always clear. When the broker receives a request r from a device, the policy instance governing the request is obtained by taking $\Phi(c)$ and applying the substitutions defined by the attributes of the authenticated device. Updating the definition of the information flow graph and the procedure for building it in Section V and VI is trivial, since things are inherently more constrained than common devices. While we kept the formal development as simple as possible, our tool is capable of handling configurations that use things, provided their bindings are specified in advance.

An additional feature of policy statements is the optional "Condition" field that further restricts the cases in which the statement is considered when evaluating a request. Conditioning expressions are built using a predefined finite set of operators, keys, and values [18]. Such conditions require only a mild extension of our model by enriching the policy statements in Definition 3 with suitable (decidable) predicates over requests. Then, Definitions 5, 6 and 8 are updated by considering these predicates in conjunction with $\rho \in \mathcal{L}(\texttt{Re}^{\texttt{aws}})$.

Finally, we argue that focusing on AWS IoT Core as a specific instance of IoT-Cloud deployments does not limit the generality of our methodology. On the one hand, focusing on a specific case enables us to concretely apply our verification approach by targeting real-world configurations. On the other hand, several MQTT brokers support authorization policies for publishing messages and subscribing to topics, e.g., Azure IoT Hub [25] and HiveMQ [26]. These policies are typically written in an RBAC policy language that supports wildcards to specify sets of topics. We are confident that our methodology can be easily adapted to these MQTT brokers with very few modifications in the technical aspects, e.g., encoding roles through suitable predicates or slightly adjusting the treatment of wildcards.

VIII. Related Work

Several papers have investigated the problem of specifying and verifying cloud-based access policies and information flow in IoT systems. However, to the best of our knowledge, no paper in the literature addresses the verification of information flow properties in cloud-based IoT access policies.

Backes et al. [10] proposed Zelkova, a tool that statically analyses policies by encoding them into SMT formulas and using off-the-shelf solvers to verify specific requirements. Zelkova can also compare different policies based on permission inclusion. Building on that, D'Antoni et al. [27] proposed an ad hoc SMT encodings for efficiently checking whether a policy is public (i.e. if the number of allowed IPs exceeds a given threshold). With a similar approach, Eiers et al. [11]–[13] use an SMT solver to quantify the permissiveness of policies and to reduce the number of allowed requests below some predefined threshold. Jin et al. [4] proposed P-Verifier, a tool to detect security flaws in policy configurations. They focus on specific flaws like wildcard injection that may result in the granting of unwanted permissions. Their verification mechanism relies on translating the policies into suitable SMT formulas and checking them with an off-the-shelf solver. Barnett et al. [28] proposed a modular formalization of the AWS authorization engine and a corresponding SMT-based analysis tool, called IAM-MULTIPOLICYANALYZER, for verifying properties pertaining to multiple policies of different types (identity-based, resource-based policies, service and resource control policies, permissions boundaries). The technical aspect of this work is similar to ours, namely determining the effect of combined policies. However, they investigate how multiple policies affect permitted operations when applied together, whereas we focus on simpler policies and check whether the

operations permitted to different entities allow communication. All these papers use an SMT solver to verify that a policy satisfies some requirements. However, unlike our work, they do not give a formal model of IoT systems, nor do they consider information flow among them. Since the properties they consider differ from ours, their verification mechanism is also different: they only analyse each policy in isolation to determine the permissions granted. Consequently, it is not straightforward to apply their methodology and tools to verify information-flow properties within a network of devices, as this requires consideration of several policies. In contrast, our verification mechanism considers all policies simultaneously and all possible device interactions to construct the information flow graph. This requires matching the (possibly infinite) sets of topics over which two devices may publish and subscribe, which is challenging because topics may depend on variables, like the client id, and may use wildcards.

Regarding information security in IoT systems, Bastys et al. [29] investigated the problem of securing IoT apps that use the If-This-Then-That paradigm. Similarly, Yu et al. [30] propose TAPFixer to detect and repair vulnerabilities in the setting of Home Automation. They perform model-checking on the Trigger-Action Programming (TAP) rules of the analysed system. Celik et al. [31] proposed IoTGuard, a dynamic, policy-based enforcement system that monitors the runtime behaviour of IoT apps to detect violations of safety and security policies. Mandalari et al. [32] propose another dynamic approach to automatically classify and block the non-essential network traffic of IoT devices. Some works [33], [34] propose process-algebra models that capture the core aspects of the behaviour of IoT systems, and use formal methods (control-flow analysis, type systems) to prove various security properties. All these papers share a goal similar to ours: ensuring that information flows within an IoT system complies with a given security policy. However, their approach is orthogonal to ours: they assume access to the app's code and take into account the interaction between sensors and actuators mediated by the environment, but they ignore access control policies. In contrast, we treat IoT devices as black boxes, over-approximating their behaviour based on the actions their IoT policies allow: we consider an attacker capable of taking full control of a compromised device, subverting its code. As a result, we do not need to re-analyse systems when a device is replaced or its behaviour changes due to an attack, as long as the access policy remains unchanged. Moreover, to streamline the discussion, we isolate the information flow permitted by the broker's access control policy; we leave to future work the analysis of scenarios involving indirect interactions between sensors and actuators. To this end, we anticipate no significant challenges in integrating our information flow graph with arcs obtained with the approaches discussed above.

IX. Conclusion

We addressed the problem of verifying information flow in IoT access policies. We focused on AWS IoT Core's main components as a specific instance of IoT-Cloud deployments. First, we introduced a formal model of IoT policies, then, we built an information flow graph to capture communication between IoT devices via MQTT topics. We reduce the problem of checking information flow between devices to a graph reachability problem. To enhance practical feasibility, we introduced a symbolic version of the graph, replacing topics with logic formulas to succinctly represent potentially infinite sets of topics. Finally, we implemented our verification mechanism in the IoT:Poker tool, we evaluated its effectiveness on a representative scenario and its scalability on a collection of real-world IoT policies.

Future work aims to improve our analysis and foster the adoption of IoT:Poker by practitioners. A first extension involves considering the typical dynamicity of IoT systems, where devices may join and leave. Then, we will implement a change-impact analysis to determine how a policy update affects the information flow and if it preserves the security requirements. Finally, we plan to consider scenarios where information flows occur through indirect interaction between devices' sensors and actuators.

Acknowledgements

This work has been partially supported by RDS PTR 25-27 CYBER 2.1 "Progetto Cybersecurity" WP3 LA 3.21 - CUP: D63C24001060001.

Data availability

The artifact(s) associated with this paper is/are available at Zenodo under the following DOI: 10.5281/zenodo.21699813.

References

[1] "Aws iot core," accessed on April 2024. [Online]. Available: https://aws.amazon.com/iot-core/

[2] "Microsoft azure iot hub," accessed on April 2024. [Online]. Available: https://azure.microsoft.com/products/iot-hub/

[3] "Mqtt version 3.1.1," accessed on April 2024. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[4] Z. Jin, L. Xing, Y. Fang, Y. Jia, B. Yuan, and Q. Liu, "P-verifier: Understanding and mitigating security risks in cloud-based iot access policies," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[5] M. Bugliesi, S. Calzavara, R. Focardi, and M. Squarcina, "Gran: Model checking grsecurity RBAC policies," in *25th IEEE Computer Security Foundations Symposium*, S. Chong, Ed., 2012.

[6] E. Uzun, V. Atluri, J. Vaidya, S. Sural, A. L. Ferrara, G. Parlato, and P. Madhusudan, "Security analysis for temporal role based access control," *J. Comput. Secur.*, vol. 22, no. 6, 2014.

[7] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka, "Verifying information flow goals in security-enhanced linux," *J. Comput. Secur.*, vol. 13, no. 1, 2005.

[8] B. S. Radhika, N. V. N. Kumar, R. K. Shyamasundar, and P. Vyas, "Consistency analysis and flow secure enforcement of selinux policies," *Comput. Secur.*, vol. 94, 2020.

[9] L. Ceragioli, L. Galletta, P. Degano, and D. Basin, "Specifying and verifying information flow control in selinux configurations," *ACM Trans. Priv. Secur.*, vol. 27, no. 4, 2024.

[10] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for AWS access policies using SMT," in *Formal Methods in Computer Aided Design*, 2018.

[11] W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince, and T. Bultan, "Quantifying permissiveness of access control policies," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022.

[12] ——, "Quacky: Quantitative access control permissiveness analyzer," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2023.
[13] W. Eiers, G. Sankaran, and T. Bultan, "Quantitative policy repair for access control on the cloud," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.
[14] "Checking Information Flow in Cloud-based IoT Access Control Policies (Supplementary Material)," doi: 10.5281/zenodo.21699813. [Online]. Available: https://github.com/edoxthebest/IoTPoker
[15] "P-verifier policy benchmark," accessed on June 2024. [Online]. Available: https://github.com/P-Verifier/P-Verifier/tree/master/policy_benchmark
[16] L. Ceragioli, L. Galletta, and E. Lunati, "Checking information flow in cloud-based iot access control policies (extended version)," 2026. [Online]. Available: https://arxiv.org/abs/2607.28088
[17] "Amazon web services," accessed on April 2024. [Online]. Available: https://aws.amazon.com/
[18] "Policies and permissions in iam," accessed on April 2024. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html
[19] "Amazon cognito," accessed on April 2024. [Online]. Available: https://aws.amazon.com/cognito/
[20] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.
[21] "Policyuniverse," accessed on June 2024. [Online]. Available: https://github.com/Netflix-Skunkworks/policyuniverse
[22] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 13243. Springer, 2022, pp. 415–442.
[23] "Networkx," accessed on June 2024. [Online]. Available: https://networkx.org/documentation/stable/index.html
[24] "The z3 theorem prover," accessed on June 2024. [Online]. Available: https://github.com/Z3Prover/z3
[25] "Access control for mqtt clients," accessed on July 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/event-grid/mqtt-access-control
[26] "Hivemq file role based access control extension," accessed on July 2025. [Online]. Available: https://github.com/hivemq/hivemq-file-rbac-extension
[27] L. D'Antoni, A. Gacek, A. Goel, D. Jovanović, R. G. Kıcı, D. Peebles, N. Rungta, Y. Sharoda, and C. Sung, "Projective model counting for ip addresses in access control policies," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 208–216.
[28] L. Barnett, L. D'Antoni, A. Goel, R. Kici, N. Rungta, M. Southern, and C. Sung, "Modeling the aws authorization engine," 2025. [Online]. Available: https://www.amazon.science/publications/modeling-the-aws-authorization-engine
[29] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what? controlling flows in iot apps," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.
[30] Y. Yu, Y. Xu, K. Huang, and J. Liu, "TAPFixer: Automatic detection and repair of home automation vulnerabilities based on negated-property reasoning," in *33rd USENIX Security Symposium*. USENIX Association, 2024.
[31] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot," in *NDSS*. The Internet Society, 2019.
[32] A. M. Mandalari, D. J. Dubois, R. Kolcun, M. T. Paracha, H. Haddadi, and D. R. Choffnes, "Blocking without breaking: Identification and mitigation of non-essential iot traffic," *Proc. Priv. Enhancing Technol.*, vol. 2021, no. 4, 2021.
[33] C. Bodei and L. Galletta, "Tracking Sensitive and Untrustworthy Data in IoT," in *Proceedings of the First Italian Conference on Cybersecurity (ITASEC17)*, ser. CEUR Workshop Proceedings, A. Armando, R. Baldoni, and R. Focardi, Eds., vol. 1816. CEUR-WS.org, 2017, pp. 38–52.
[34] M. Balliu, M. Merro, M. Pasqua, and M. Shcherbakov, "Friendly fire: Cross-app interactions in iot platforms," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, Apr. 2021.