Access Control Model for Virtual Objects (Shadows) Communication for AWS Internet of Things

Asma Alshehri, James Benson, Farhan Patwa and Ravi Sandhu
Institute for Cyber Security (ICS),
Center for Security and Privacy Enhanced Cloud Computing (C-SPECC), and
Department of Computer Science, University of Texas at San Antonio, San Antonio, Texas, US
{nmt366,james.benson,farhan.patwa,ravi.sandhu}@utsa.edu

ABSTRACT

The concept of Internet of Things (IoT) has received considerable attention and development in recent years. There have been significant studies on access control models for IoT in academia, while companies have already deployed several cloud-enabled IoT platforms. However, there is no consensus on a formal access control model for cloud-enabled IoT. The access-control oriented (ACO) architecture was recently proposed for cloud-enabled IoT, with virtual objects (VOs) and cloud services in the middle layers. Building upon ACO, operational and administrative access control models have been published for virtual object communication in cloud-enabled IoT illustrated by a use case of sensing speeding cars as a running example.

In this paper, we study AWS IoT as a major commercial cloud-IoT platform and investigate its suitability for implementing the afore-mentioned academic models of ACO and VO communication control. While AWS IoT has a notion of digital shadows closely analogous to VOs, it lacks explicit capability for VO communication and thereby for VO communication control. Thus there is a significant mismatch between AWS IoT and these academic models. The principal contribution of this paper is to reconcile this mismatch by showing how to use the mechanisms of AWS IoT to effectively implement VO communication models. To this end, we develop an access control model for virtual objects (shadows) communication in AWS IoT called AWS-IoT-ACMVO. We develop a proof-of-concept implementation of the speeding cars use case in AWS IoT under guidance of this model, and provide selected performance measurements. We conclude with a discussion of possible alternate implementations of this use case in AWS IoT.

KEYWORDS

Security; Access Control; Internet of Things (IoT); AWS IoT; IoT Architecture; Devices; Virtual Objects; ACL; RBAC; ABAC;

ACM Reference Format:
Asma Alshehri, James Benson, Farhan Patwa and Ravi Sandhu. 2018. Access Control Model for Virtual Objects (Shadows) Communication for AWS Internet of Things. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3176258.3176328

CODASPY '18, March 19–21, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5632-9/18/03…$15.00
https://doi.org/10.1145/3176258.3176328

1 INTRODUCTION

The Internet of Things (IoT) raises new security challenges, which require significant revisions and enhancements of existing security solutions, including access control systems. Recently an access-control oriented architecture (ACO) [2] for cloud-enabled IoT has been developed, which includes four layers: an object layer, a virtual object (VO) layer, a cloud services layer, and an application layer (see Section 2.1). The ACO recognizes the need for communication control within each layer and across adjacent layers, as well as the need for data access control at the cloud services and application layers. Multiple and diverse access control models are required at various points in this architecture, which must collectively enforce over-arching access control policies reflecting the complexity of cloud-enabled IoT. Towards this end, a set of access control models for VO communications has been published [3], referred to as ACO-IoT-ACMsVO. These models are developed in two layers: operational models and administrative models. Also, the style of communication among VO is based upon publish/subscribe topic-based communication interaction scheme. The ACO-IoT-ACMsVO models are illustrated by a use case of sensing speeding cars as a running example [3] which we will utilize in this paper (see Section 2.2).

The principal goal of this paper is to reconcile the afore-mentioned academic models with a major commercial cloud-IoT platform, viz., AWS IoT. While AWS IoT has a notion of digital shadows closely analogous to VOs, it lacks explicit capability for VO communication and thereby for VO communication control. Thus, there is a significant mismatch between AWS IoT and these academic models. Nevertheless, as we will show, it is possible to use AWS IoT mechanisms to effectively realize and control VO communications. This demonstrates on one hand that academic models developed independent of AWS IoT can be enforced using this commercially significant platform. It also suggests enhancements to AWS IoT that would be beneficial to facilitate such enforcement. We believe that in the rapidly developing ecosystems of cloud, IoT and their intersection, it is crucial to place academic work within major industry developments. This is the primary motivation for this research.

The rest of the paper is organized as follows. First, we review the ACO architecture for cloud-enabled IoT, the published ACO-IoT-ACMsVO [3] access control models for VO communication within ACO, and the general access control model for AWS IoT called AWS-IoTAC [5] in Section 2. Then, within AWS IoT, we develop an access control model for virtual object communication (AWS-IoT-ACMVO)

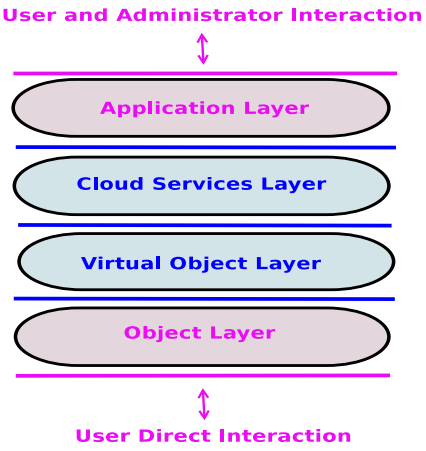

Figure 1: ACO Architecture for Cloud-Enabled IoT

in Section 3. Section 4 discusses the use case of ACO-IoT-ACMsVO within the AWS-IoT-ACMVO model. Section 5 discusses proof-of-concept implementations of the use case in two scenarios in AWS IoT platform. Selected performance aspects of our implementation are described in Section 6. A discussion of some issues of AWS IoT and possible enhancements are explained in Section 7. Finally, Section 8 concludes the paper.

2 BACKGROUND

2.1 ACO Architecture

The Access Control Oriented (ACO) Architecture for IoT was proposed in [2], consistent with various published IoT architectures [1, 6, 7, 9–14]. ACO architecture comprises four layers: an object layer, a virtual object layer, a cloud services layer, and an application layer, as shown in Figure 1. We briefly discuss each layer below.

The object layer comprises heterogeneous physical objects such as sensors, actuators, cameras, cellphones, etc. Users can directly communicate with objects by pressing a button, changing a device, powering on an object, etc. Also, objects can communicate directly to each other through communication technologies, or indirectly through virtual objects.

A virtual object represents the persistent current status of a physical object, when the two are connected. Otherwise, a virtual object could represent the last received state, desired future state, or both. A virtual object can have a subset of a physical object's services, all of a physical object's services, or a subset of a physical object's services. Virtual objects can uniformly communicate with each other regardless of heterogeneity and locality in the object layer. Virtual to physical object association can be one-to-one, many-to-one, one-to-many, or many-to-one [2].

The cloud services layer assists in storing and processing the collected data. This data can be used intelligently for smart monitoring and actuation, and it can be visualized in ways that are more meaningful for users. Thus, policymakers (or administrators) can utilize the visualized data to help them to modify or add policies that are kept in the cloud, so the communication and access between applications and objects are managed through the cloud. In addition, multiple IoT clouds can also communicate with each other, ranging from only providing services and information at a local level to collaborating with other connected IoTs in order to share information at a broad level and pursue common goals.

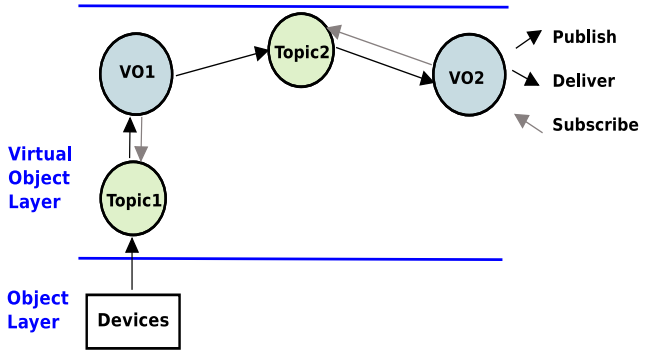

Figure 2: The Publish/Subscribe Topic-Based Scheme in the ACO-IoT-ACMsVO

The application layer is the topmost layer of the proposed ACO IoT architecture and offers an interface through which users can easily communicate with objects and visualize the analyzed information. Administrators can also interact with applications to generate policies or to update/add policies based on the obtained information. Moreover, configuring and managing the communication of objects and virtual objects is organized by administrators through applications. General users and administrators can remotely communicate with IoT objects and virtual objects only through applications.

2.2 Access Control Models for VO Communication in Cloud-Enabled IoT

The ACO architecture emphasizes the need for communication control and data access control within and across ACO layers [2]. One of the communication points that needs to be controlled is virtual object communication. Alshehri and Sandhu used the ACO architecture for IoT to propose access control models for virtual object communication (ACO-IoT-ACMsVO) [3]. The developed access control models are in two layers: operational models and administrative models. The current dominant access control models, viz., access control lists (ACLs), capability lists, and role-based access control (RBAC), are formally defined in both operational and administrative models for VO communication. Also, attribute-based access control (ABAC) models are proposed, because ABAC encompasses the benefits of previous traditional models, as well as brings new features appropriate for dynamic and open environments such as IoT.

The ACO-IoT-ACMsVO models are developed utilizing publish/subscribe communication interaction scheme. This scheme is appropriate for large-scale distributed interactions such as the IoT. The basic implementation style of publish/subscribe paradigm is topic-based scheme. The topic-based scheme is comparable to the idea of groups, where producers (publishers) publish data to a topic and consumers (subscribers) become members of a topic (a group) [4, 8]. Figure 2 shows the general idea of publish/subscribe topic-based scheme that is used in ACO-IoT-ACMsVO.

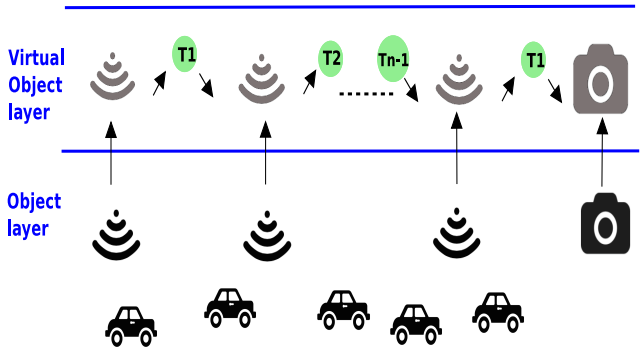

Figure 3: The Sensing Speeding Cars Use Case within ACO Architecture [3]

The operational AC models specified in [3] focus in placing the control on both VO side and topic (T) side to authorize VO to VO communications via topics. In other words, the operational AC models address the following questions. Which VOs are allowed to publish or send a subscription request to a topic? Which VOs should a topic forward data to? Which topic should VOs publish to or send a subscription request to? Which topics should VOs receive data from? This dual scheme permits unauthorized actions to be denied at the earliest possible moment, rather than postponing the decisions until later. Therefore, the decision of the VO communication in Figure 2 will be upon both of VO and T access control list and capability list (ACL-Cap Operational Model) or upon both of VO and T attributes (ABAC Operational Model).

A use case of sensing speeding cars is employed in [3] as a running example. Figure 3 shows a simplified picture of the employed use case. One car is recognized to be speeding if two sensors within a specified distance sense the speed to be over limit, and a camera will report pictures of the over-limit cars. Physical sensors and the camera identify cars by attached RFIDs and push collected data (e.g. RFID and Speed) to their virtual objects where more powerful computations and communication could happen. The use case assumes that sensors can only communicate within the virtual object layer, and they cannot communication directly with each other.

2.3 The General Access Control Model for AWS-IoT (AWS-IoTAC)

Bhatt et al [5] study AWS IoT as a major commercial cloud-IoT platform, and develop a formal access control model for AWS-IoT called AWS-IoTAC. This access control model is an extension of AWS access control (AWSAC) model previously developed by Zhang et al [15] for AWS access control in general.

AWS-IoTAC comprises all the components and relations of AWSAC with modified or extra set of components and relations related to the AWS IoT service. The main component of AWSAC are Accounts (A), Users (U), Groups (G), Roles (R), Services (S), Object Types (OT), and Operations (OP). The additional components in AWS-IoTAC, which are related to AWS IoT service, are Certs (C), IoT Objects (IO), IoT Operations (IOP), Rules (Ru), Virtual Permission Assignment (VPA), and Devices (D). The functionality of these entities and their relationship to each other is formally described in [5].

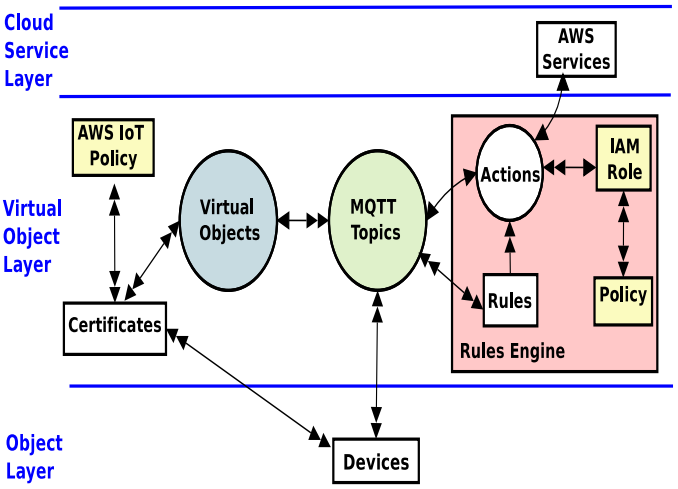

Figure 4: The Components of AWS-IoT-ACMVO

3 THE AWS-IOT-ACMVO MODEL FOR AWS IOT SHADOWS COMMUNICATION

In this section, based on our extensive exploration of AWS IoT platform, its documentation, and our implemented use cases, we propose an access control model for virtual objects (shadows) communication called AWS-IoT-ACMVO as an abstracted view of AWS IoT capabilities. Figure 4 shows the major components of this model, viz., certificates, AWS IoT policies, virtual objects (device shadows), Message Queuing Telemetry Transport (MQTT) topics, and rules engine and its action. The details of their functionalities are discussed below.

AWS IoT uses X.509 certificates as an identity credential for devices authentication [5]. Certificates can be either an AWS IoT generated certificate or a certificate signed by a AWS IoT registered external certification authority. Generally, one certificate can be given to many **devices**, but it is recommended that each device has a unique certificate to enable fine-grained device management. Figure 4 shows that each certificate can be given to more than one device, and each device can have multiple certificates (the arrow with double end means a multiplicity). However, every time a device connects it can only activate one certificate.

Once a certificate is generated, there are two AWS IoT entities that need to be attached to the certificate in order to authenticate and authorize AWS IoT devices, which desire to communicate with virtual objects (device shadows), viz., **AWS IoT policy** and **virtual objects**. An AWS IoT policy is a JSON document that is attached to a certificate for authorization purpose. It comprises one or more policy statements, each of which specifies effect, action, resources, and optional condition. An action is an operation that can be granted or denied to a resource as determined by the effect value. Actions can be MQTT policy actions or thing shadow policy actions. The MQTT policy actions are the operations that deal with connecting, sending, or receiving data, which are iot:Connect, iot:Publish, iot:Subscribe, and iot:Receive. On the other hand, thing shadow policy actions deal with permissions to handle virtual objects (device shadows), which are iot:DeleteThingShadow, iot:GetThingShadow,

and iot:UpdateThingShadow. Figure 4 shows that each AWS IoT policy can be attached to more than one certificate, and each certificate can attach multiple AWS IoT policies. Generally, the AWS IoT policy is attached to a certificate to authorize any kind of actions (e.g. iot:Publish and iot:GetThingShadow) to devices that hold that certificate (and its private key).

Virtual objects (device shadows) also need to be attached to a certificate as a resource that a device is fully or partially authorized to access. A virtual object can be given more than one certificate, and a certificate can attach to more than one device. Figure 4 shows the many-to-many relationship between certificates and virtual objects. A virtual object is also a JSON document that stores information about the current state of a connected device and the desired future state of the connected device (there is no recent or historical state). One of the benefits of the device shadow is that its information can be used to set or get the state of its device, even if the device is not connected. In general, a device that holds a certificate with attached policies and virtual objects has the rights to communicate and access to the attached virtual objects (one virtual object at each connection) based on the attached policies.

In AWS IoT, applications cannot directly update or retrieve data of devices. Virtual objects in AWS work as an intermediate point of communication among applications and physical devices. The only way for applications or devices to interact with a virtual object is to communicate with its **MQTT topics**. In other words, MQTT topics of a virtual object allow applications and devices to get, update, or delete the state information of the virtual object (device shadow) by publishing or subscribing to its MQTT topics. The name of each MQTT topic begin with $\$aws/things/thingName/shadow/\#$, where the thingName is the name of a virtual object, and the symbol # could be one of the thingName MQTT topics that can be used to interact with the thingName. There are reserved thingName MQTT topics for each virtual object that can be used to publish or subscribe to the virtual object. In order to send a request to a thingName (a virtual object), we only can use /update, /get, or /delete as thingName MQTT topics of that thingName. While /update/accepted, /update/reject, /update/delta, /update/documents, /get/accepted, /get/rejected, /delete/accepted, and /delete/rejected MQTT topics are used by the thingName itself to publish an acknowledgement about accepting or rejecting the received request. Generally, AWS IoT service generates reserved MQTT topics for each created virtual object. The reserved MQTT topics is the only way to communicate with the created virtual object. Figure 4 shows that each virtual object has specific reserved MQTT topics, and each reserved MQTT topic is only related to one virtual object. Moreover, each device can communicate with more than one MQTT topic as long as it has an authorized certificate, and each MQTT topic can be used by more than one device if devices are authorized.

A powerful mechanism in AWS IoT is that a message sent to an MQTT topic can be recognized and analyzed by a rule. **Rules** provide processing for the arrived messages to MQTT topics and enable interactions with various AWS services. A rule consists of a rule name, optional description, SQL statement, SQL version, and one or more actions. The SQL statement is used to filter received messages to MQTT topics, and then the rule engine forwards it to AWS services or republishes it to other MQTT topics by using the action field specified in the rule. There are fixed AWS actions that

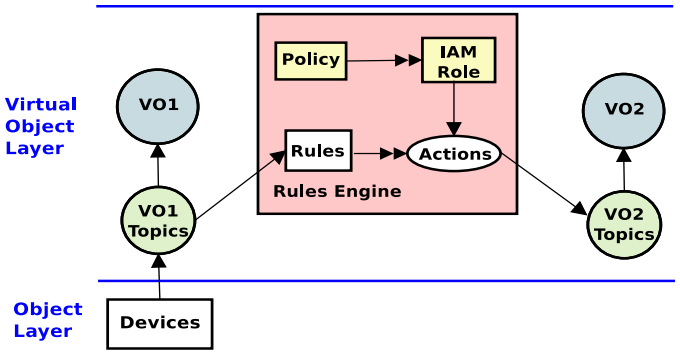

Figure 5: The Rules Engine as a Communication Channel in AWS-IoT-ACMVO

can be selected, such as inserting a message into a DynamoDB table, invoking a Lambda function, and republishing messages to AWS IoT topics. Thus, rules that are attached to MQTT topics provide ways for virtual objects to interact with AWS services or republish the received messages to other MQTT topics (reserved or unreserved). Figure 4 shows that each rule can be triggered by more than one topic, and each topic can trigger more than one rule. Also, when a rule is triggered, one or multiple actions can be executed.

When rules forward the published messages to another AWS service, such as AWS Lambda, the authorization to access the other service and the actions of other service can be controlled via AWS identity and access management **(IAM) role**. Each IAM role is attached with at least one **policy** that grants permissions to access resources specified in the action of the rule or to control actions toward the received data. For example, when an Amazon SNS rule is created, an IAM role will be attached to that SNS rule to authorize access to SNS resources. The attached role will have policies that allow actions (e.g. sns:Publish) toward specific resources in Amazon SNS. Similarly, when a lambda rule is created, an IAM role will be attached to the lambda function. This attached IAM role will have policies that authorize actions (e.g. iot:Publish, iot:GetThingShadow) toward specific resources in AWS Lambda. Thus, we can see that IAM role and its attached policies are a part of the AWS IoT rule definition to control actions. Figure 4 shows that each action of a rule can only attach one IAM role, but each IAM role can be used by many rule actions. Also, one IAM role can attach many policies, and one policy can be attached to many IAM roles.

4 ISSUES IN ENFORCING ACO-IOT-ACMSVO WITHIN AWS-IOT-ACMVO

AWS IoT does not support direct communication among VOs, because a VO is only allowed to communicate directly with its reserved topics. The AWS-IoT-ACMVO model is one way to effect VOs communication via rules within AWS IoT. AWS-IoT-ACMVO keeps the transient data within the virtual object layer without persistent storage, while only data about actual speeding cars is propagated to the higher layers. Thus, the privacy of data can be preserved. All components of VOs communication that contribute in this communication are shown in Figure 4. Figure 5 show how

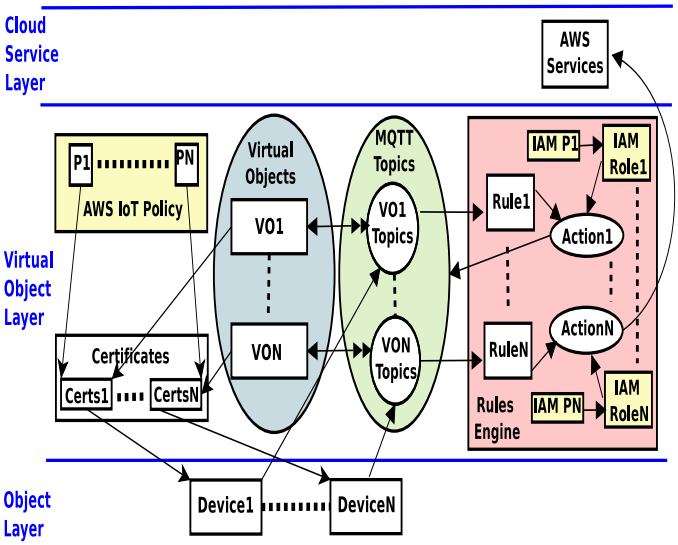

Figure 6: The Sensing Speeding Cars Use Case within AWS-IoT-ACMVO

the rules engine of AWS IoT serves as a communication channel between VOs in the AWS-IoT-ACMVO model. The ACO-IoT-ACMsVO academic model assumes the communication regime shown in Figure 2 where the communication channel between two VOs is a shared topic to which VO1 publishes and VO2 subscribes. The rules engine enables a similar effect to be achieved in AWS IoT as shown in Figure 5.

The control points to authorize VOs communication via topics in ACO-IoT-ACMsVO is placed on both VO side and topic (T) side [3]. For example, in case of ACL-Cap operational model, a VO can have a right to publish to a topic only if the topic in the capability list of the VO and the VO is in the access control list of T. In case of the ABAC operational model, a publish permission will be authorized if the topic is within VO-Publish attribute values and the VO is within T-Publish attribute values. The subscription right is similarly authorized on both sides.

In case of AWS-IoT-ACMVO, the control point to authorize reserved topics of a VO to communicate with other reserved topics of another VO is placed in rules engine. For example, when a data arrived to a reserved topic of a VO, a lambda rule will trigger a lambda function as an action if the Select Clause and Where Clause in the SQL statement of the lambda rule evaluate to true. When the lambda function is triggered, an attached IAM role with lambda function will comply with a coupled policy or policies to authorize AWS-IoT to access to the lambda function and authorize the lambda function to execute actions with the received data. IAM role policy could authorize lambda function to forward data to other reserved/unreserved topics. Thus, other topics will receive data as long as it is in an appropriate format without checking where the data came from or rejecting the received data, and as a result, the received data will be forwarded to subscribers. So, a question like "which resources should a topic receives data from?" is only controlled via IAM role that is attached within an action of a rule, and topics has no control over what they receive.

We investigated applying the use case of sensing speeding cars, which is employed in ACO-IoT-ACMsVO, within AWS-IoT-ACMVO. But as we discussed above, the communication style, access control points, access control models are not precisely alike. Although, AWSnIoT does not support direct VOs communication, we were able to develop AWS-IoT-ACMVO that allows effect VOs communication. So, the use case of sensing speeding cars within ACO architecture in Figure 3 can be applied and enforced within AWS IoT as shown in Figure 6. The details of configuration, scenario, and authorization policy are discussed in the following section.

5 A USE CASE: THE SENSING SPEEDING CARS WITHIN AWS-IOT-ACMVO

In this section, we present two scenarios of the use case of sensing cars speed. The two scenarios will have number of sensors and a camera in the physical layer. All devices on the physical layer will push collected data to their virtual objects (shadows). In our scenarios, we focus on the communication among virtual objects and how this communication can be controlled.

5.1 Sensing the Speed of One Car

We will discuss the configuration and the scenario of our simple use case as follow.

5.1.1 Setup and Configuration. In this simple scenario, we will have two physical sensors and one physical camera each with one virtual object connected to it. Figure 7 shows the connected devices, virtual objects (shadows), certificates, AWS IoT policy, rules, actions and their IAM roles, and AWS services.

First, we create one virtual object for each physical object using AWS IoT management console and attach one X.509 certificate for each virtual object. For each one certificate, we attached an AWS IoT policy. Certificates are copied into their corresponding physical objects to allow authentication and authorization of physical objects when they communicate with the corresponding virtual objects. In other words, the attached AWS IoT policy authorizes specific actions (connect and publish) for physical objects. When certificates are given to the corresponding physical objects, they are accompanied by the private key of the certificate and an AWS root CA certificate.

We simulated sensors and camera physical objects using AWS SDK for JavaScript (Node.js). There is an attached rule for each MQTT update topic $\$aws/things/Sensor_i/shadow/update$ that triggers a Lambda function. Lambda functions are responsible about republishing the coming reported data that arrived to a virtual object ($Virtual\ Sensor_i$ or $Virtual\ Camera$) from its corresponding physical object ($Sensor_i$ or $Camera$) to the next virtual object ($Virtual\ Sensor_{(i+1)}$ or $Virtual\ Camera$) as shown in Figure 7. Also, each Lambda function is attached with IAM role that authorizes AWS IoT to access AWS and AWS IoT resources and services. The IAM role also controls Lambda function operations, such as republishing data to another topic or getting the current state of a shadow.

5.1.2 Senario. $Sensor_1$ sends RFID and Speed of the over speeding car as a *reported* message to $Virtual\ Sensor_1$ ($VS1$) by publishing to $Sensor_1$ MQTT update topic $\$aws/things/Sensor_1/shadow/update$. $Rule_1$ that is attached with the $Sensor_1$ MQTT update topic will

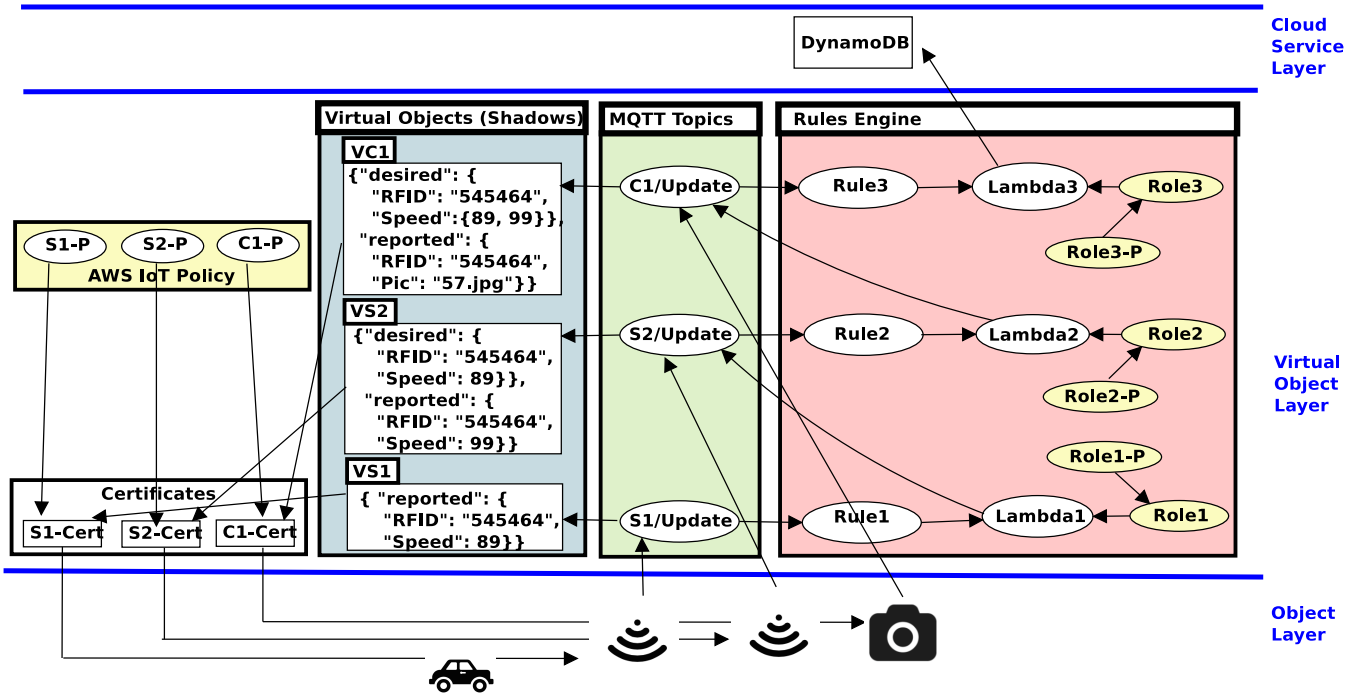

Figure 7: A Simple Use Case of Sensing the Speed of One Car

trigger $Lambda_1$ *function* every time data arrived to MQTT update topic of $VS1$. $Lambda_1$ *function* republishes the arrived data to *Virtual* $Sensor_2$ ($VS2$) with a *desired* tag. Figure 7 shows that the *reported* RFID and Speed to $VS1$ is republished to $VS2$ as *desired* state by $Lambda_1$ *function*.

$Sensor_2$ also sends RFID and Speed of the over speeding car as a *reported* message to $VS2$ by publishing to the following MQTT update topic: $\$aws/things/Sensor_2/shadow/update$. $Rule_2$ is going to trigger $Lambda_2$ *function* every time data arrived to MQTT update topic of $VS2$. $Lambda_2$ *function* check if the coming data is with *reported* tag, it compares the saved *desired* RFID with the coming *reported* RFID from $Sensor_2$. If the two RFIDs are matched, $Lambda_2$ *function* combines the two speeds and one RFID and publish it with a *desired* tag to the *Virtual Camera* ($VC1$). Figure 7 shows that the *reported* RFID matches the *desired* RFID in *Virtual* $Sensor_2$. Thus, $VC1$ will receive from $Lambda_2$ *function* two speeds that are reported from $Sensor_1$ and $Sensor_2$ for the same RFID.

Camera also sends RFIDs and pictures (Pic) of the passed cars as a *reported* message to *Virtual Camera* by publishing to MQTT update topic $\$aws/things/Camera/shadow/update$. $Rule_3$ is going to trigger $Lambda_3$ *function* every time data arrived to MQTT update topic of $VC1$. $Lambda_3$ *function* check if the coming data is with a *reported* tag, it compares the saved coming *desired* RFID from $Sensor_2$ with the coming *reported* RFID from *Camera*. If the two RFIDs are matched, $Lambda_3$ *function* combines the RFID, Speeds, and Pic and store them to the Amazon DynamoDB. Figure 7 shows that the *reported* RFID matches the *desired* RFID in $VC1$. Thus, the combined RFID, Speeds, and Pic will be stored in in the Amazon DynamoDB.

```
{ "Version": "2012-10-17",
 "Statement":
 [
   { "Effect": "Allow",
     "Action": [ "iot:Connect" ],
     "Resource": ["arn:aws:iot:us-west-2:760000000000:
      client/Sensor2"]
   },
   { "Effect": "Allow",
     "Action": [ "iot:Publish"],
     "Resource": [ "arn:aws:iot:us-west-2:76000000000:
       topic/$aws/things/Sensor2/shadow/update"]
   }
 ]
}
```

Figure 8: *S2-P* **that is Attached to** *S2-Cert*

5.1.3 Authorization policy. There is an AWS IoT policy attached with each certificate to authorize specific actions for physical objects. For example, $Sensor_1$ are only allowed to connect and publish to $VS1$ in order to send the collected RFID and Speed of the over speed cars. Thus, the AWS IoT *S1-P* and $VS1$ are attached with *S1-Cert* which is copied to $Sensor_1$. The policy states that connect and publish actions are allowed to the specified resources, which is $VS1$ (the shadow of $Sensor_1$). Similarly, the AWS IoT *S2-P* in Figure 8 and $VS2$ will be attached to *S2-Cert*, which is copied to $Sensor_2$, and the AWS IoT *C1-P* and $VC1$ will be attached to *C1-Cert*, which is copied to $Camera_1$. AWS IoT defines policy variables, which can be used in AWS IoT policies within the resource or condition block. The basic variable *IoT* : *ClientID* can be used to generate a policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {  "Effect": "Allow",
       "Action": "iot:GetThingShadow",
       "Resource": "arn:aws:iot:us-west-2:760000000000:
        thing/Sensor2"
    },
    {  "Effect": "Allow",
       "Action": "iot:Publish",
       "Resource": "arn:aws:iot:us-west-2:760000000000:
        topic/$aws/things/Camera/shadow/update"
    }
  ]
}
```

Figure 9: *$Role_2$ Policy* **that is Attached to** *$Role_2$*

that can be attached to all certificates. However, certificates is not coupled with an ID of physical sensor that should connect and publish to the attached shadows, so malicious sensor could change their ID to connect and publish to any other MQTT update topic. Therefore, we preferred to specify and hard-coded one different policy for each certificate as shown in Figure 7, and each AWS IoT policy is similar to *S2-P* shown in Figure 8 but with different sensor names.

Also, There is an IAM role attached to each Lambda function to authorize it accessing to AWS services and AWS IoT resources. For example, $Role_1$ is attached to $Lambda_1$ to authorize it publishing to the update topic of $VS2$. Also, $Role_2$ is attached to $Lambda_2$ to authorize it getting the *desired* state of $VS2$ and publishing to the MQTT update topic of $VC1$. Figure 9 shows the IAM $Role_2$ *Policy* that is attached to $Role_2$. Also, $Role_3$ is attached to $Lambda_3$ to authorize it getting the *desired* state of $VC1$ and publishing to Amazon DynamoDB.

5.2 Sensing the Speed of Multiple Cars

The previous simple use case introduces the basic idea of implementing and controlling the virtual object communication within AWS IoT. However, in realty there is a need to track multiple cars, where different cars pass a sensor at a time. A VO (shadow) in AWS IoT has different reserved topics that are used by the VO to subscribe to them. So, any time a sensor publish a new list of RFIDs/Speeds, the old list is deleted and a new one is saved. However, our use case with multiple cars needs to keep track of the historical data (old and new RFIDs).

In this use case, for every VO corresponding to a physical object, we propose to have another relative VO that works as storage of historical data. The only way to push or get data from the VO storage is by using a lambda function that is triggered by publishing data from a sensor to the MQTT update topic of the corresponding VO. Figure 10 shows sensors ($S1, S2, ..., Sn, C1$) and their corresponding virtual objects ($VS1, VS2, ..., VSn, VC1$) and the storage for each of them ($VS1S, VS2S, ..., VSnS, VC1S$).

5.2.1 Setup and Configuration. As pervious simple use case, we create one virtual object and one virtual object storage for our physical objects and attached one X.509 certificate for each virtual object. Certificates that are attached with AWS IoT policies are copied into their corresponding physical objects. The AWS IoT policy states that sensors and the Camera are only allowed to connect and publish to the corresponding VO (similar to the mentioned policy in Figure 8).

We simulated Sensors and the Camera using AWS SDK for JavaScript (Node.js). Lambda functions are triggered by rules that are attached with MQTT update topics of VOs. For example, $Lambda_1$ is triggered by rule1 that is attached to MQTT update topic of VS1. In general, Lambda functions are responsible about the complex computations, such as getting the stored data, comparing and consolidating the coming and the stored data, and republishing data to the current storage or next VO. Figure 10 describes the functionality of each lambda function.

5.2.2 Senario. $Sensor_1$ sends a list of RFIDs/Speeds of over speeding cars as a *reported* message to $VS1$ by publishing to $VS1$ MQTT update topic. $Rule_1$ triggers $Lambda_1$ $function$ when a published request arrived to MQTT update topic. $Lambda_1$ $function$ will consider the reported RFIDs/Speeds as a suspicious list, that will be handled as described in Figure 10.

$Sensor_2$, ..., $Sensor_i$, .., $Sensor_n$ also send a list of RFIDs/Speeds of over speeding cars as a *reported* message to their corresponding VO by publishing to VO_i MQTT update topic, where $2 \leq i \leq n$. The reported RFIDs/Speeds from physical objects is considered as a suspicious list (stored in $VSiS$ under reported tag) beside the suspicious list that is coming from a previous VO (stored in $VSiS$ under desired tag with RFID1). The matched RFIDs in both of the suspicious lists will be stored as SavePic list (stored in $VSiS$ under desired tag with RFID2). $Rule_i$ triggers $Lambda_i$ $function$ when a published request arrived to MQTT update topic of VS_i. $Lambda_i$ $function$ will deal with the arrived data as suspicious lists and handle it to generate the SavePic list as descried in Figure 10. Note that $Lambda_3$ to $Lambda_{(n-1)}$ will do the same computations.

Camera sends RFIDs and pictures (Pic) of the passed cars as a *reported* message to $Virtual\ Camera$ ($VC1$) by publishing to MQTT update topic of $VC1$. $Rule_{(n+1)}$ triggers $Lambda_{(n+1)}$ $function$ when a published request arrived to $VC1$ MQTT update topic. $Lambda_{(n+1)}$ $function$ will deal with the coming data as descried in Figure 10.

5.2.3 Authorization policy. As pervious simple use case, we will have an AWS IoT *Policy* that is attached with *S1-Cert*, .., *Sn-Cert*, *C1-Cert*. The policy state that physical objects can only connect to their corresponding VO and publish to MQTT update topic of the VO. Figure 8 is an example of an attached AWS IoT policy that authorizes physical objects to connect and publish.

Also, The IAM roles are attached to Lambda functions. For example, $Role_1$ is attached to $Lambda_1$ to authorize it publishing to the update topic of $VS2$. $Role_2$ is attached to $Lambda_2$ to authorize it getting the data of the storage of $VS2$ and publishing only to the update topic of the $VS2S$ and $VS3$. $Role_{(n+1)}$ is attached to $Lambda_{(n+1)}$ to authorize it getting the data of the storage of $VC1$ and publishing only to its storage and then to the Amazon DynamoDB. Figure 11 shows the $Role_5$ *Policy* of the IAM $Role_2$ that is attached to $Lambda_5$ to authorize it getting the data that is saved in the storage of $VS5$ ($n = 5$ in our implementation) and publishing only to the update topic of the $VS5S$ and to the update topic of $VC1$.

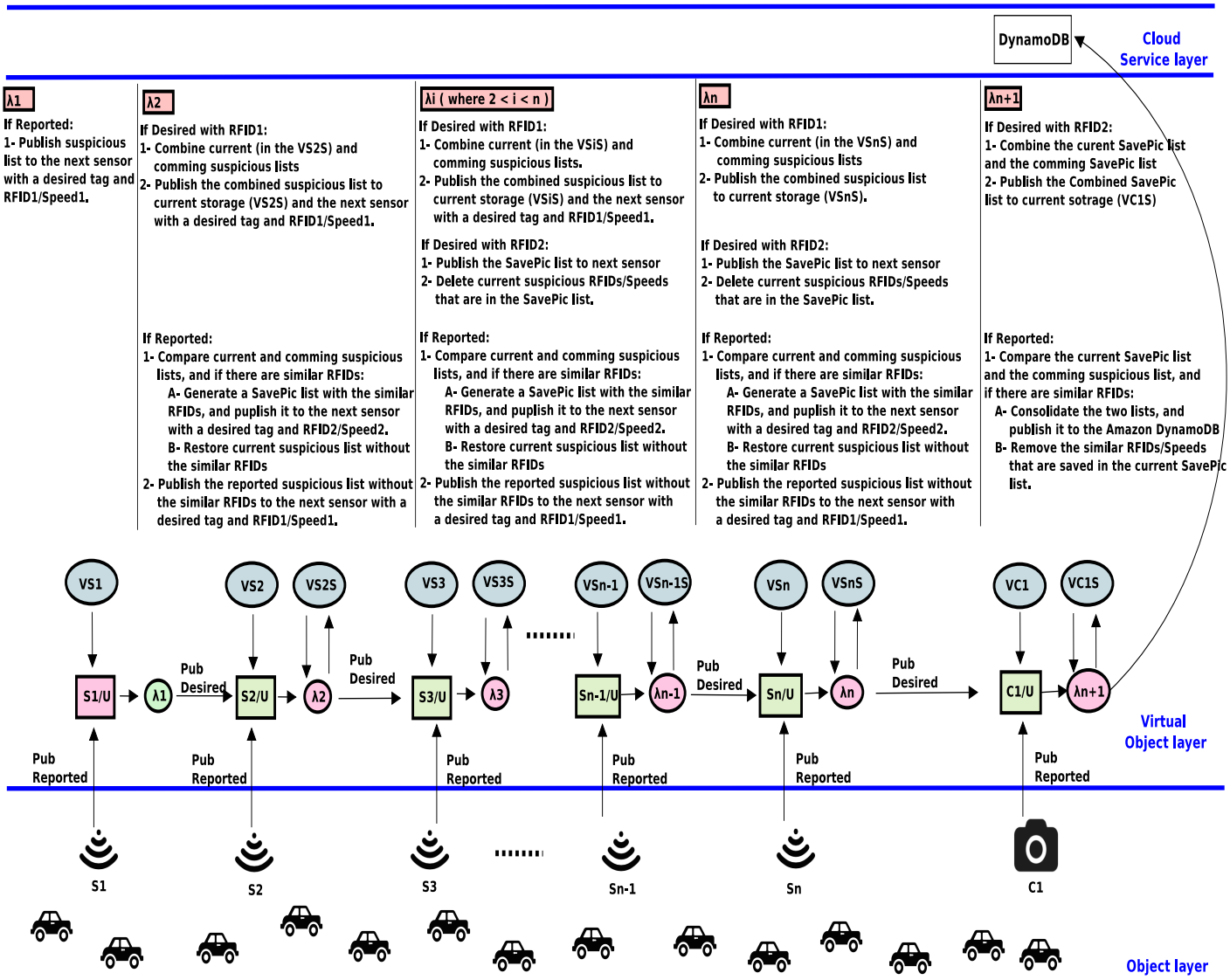

Figure 10: A Use Case of Sensing the Speed of Multiple Cars

```
{"Version": "2012-10-17",
  "Statement": [
    {  "Effect": "Allow",
       "Action": "iot:Publish",
       "Resource": "arn:aws:iot:us-west-2:760000000000:
        topic/$aws/things/Sensor5_Storage/shadow/update"
    },
    {  "Effect": "Allow",
       "Action": "iot:GetThingShadow",
       "Resource": "arn:aws:iot:us-west-2:769000000000:
        thing/Sensor5_Storage"
    },
    {  "Effect": "Allow",
       "Action": "iot:Publish",
       "Resource": "arn:aws:iot:us-west-2:769000000000:
        topic/$aws/things/Camera/shadow/update"}
  ]
}
```

Figure 11: $Role_5$ that is attached to $Lambda_5$

6 PERFORMANCE

Our scenario propagates the Suspicious list published by any sensor until the last virtual sensor, and it propagates the SavePic list from the moment of generation until the camera. The first possible generated Suspicious list starts from $Sensor_1$, and the first possible SavePic list starts when $Sensor_2$ publishes similar Suspicious list to the published Suspicious list by $Sensor_1$, so the SavePic list will be generated by $lambda_2$ $function$ that is triggered when $Sensor_2$ publishes to its virtual object. In this section, we calculate the time of propagating the Suspicious and the SavePic list to their final destination.

The use case with multiple sensors and cars is employed in computing the propagation time. we set the number of sensors to five. We used two AWS SDKs for JavaScript (Node.js) to subscribe to $Virtual\ Sensor_5\ Storage$ ($VS5S$) and $Virtual\ Camera_1\ Storage$

($VC1S$), so we can get an acknowledgement whenever the Suspicious and the SavePic list are reached. A bash script is written to run $Sensor_1$, start the timer, run $VS5S$, and end the timer whenever we get an acknowledgement from $VS5S$. Similarly, the bash script will run $Sensor_2$ (with smillar RFIDs of $Sensor_1$), start the timer, run $VC1S$, and end the timer whenever we get an acknowledgement from $VC1S$. Thus, we were able to calculate the propagation time of the Suspicious and the SavePic list to their final destination.

We run $Sensor_1$ that publish the Suspicious list with {1, 10, 20, 30, 40} RFIDS. For the Suspicious list with one RFID, we calculate the propagation average time of 10 times run. Thus, the propagation time of the Suspicious list with one RFID from S_1 until $VS5S$ in Figure 12, which is 5915 millisecond, is the average of 10 times run. Similarly, the propagation time of the Suspicious list with 10, 20, 30, 40 RFIDs from S_1 until $VS5S$, which is 6335, 7131, 7519, and 8109 millisecond, is also the average of 10 times run. However, we get rid off outliers, which is the time values that exceed 10000 or less than 3000 millisecond.

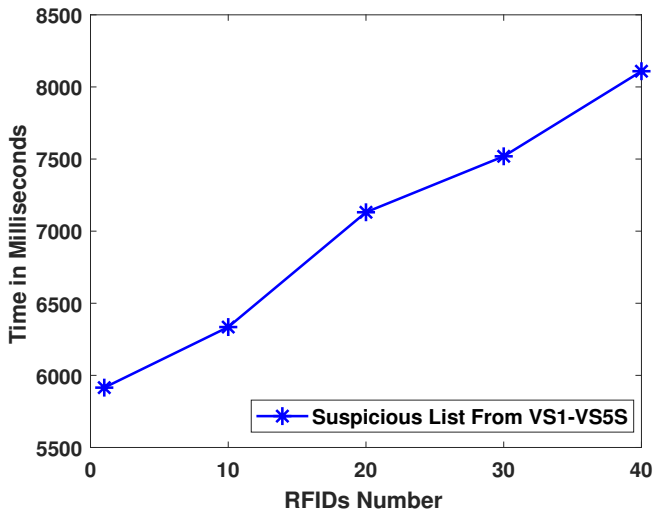

Figure 12: Propagation Time of Suspicious List from S_1 until $VS5S$

After a Suspicious list is published by S_1 and an acknowledgement is received from $VS5S$, $Sensor_2$ is also run to publish a Suspicious list, similar to the Suspicious list that is published by S_1, with {1, 10, 20, 30, 40} RFIDS. The propagation time of the SavePic list with {1, 10, 20, 30, 40} RFID from S_2 until $VC1S$ in Figure 13, which is 7774, 8100 , 8405, 8694 , 8851 millisecond, is the average of 10 times run. However, we get rid off outliers, which is the time values that exceed 14000 or less than 4000 millisecond.

The algorithms of our Lambda functions that we used within our use case in Figure 10 shows more computation and steps when a Lambda function gets the Suspicious list than when a Lambda function gets the SavePic list. However, our results in Figure 12 and 13 show that the propagation time of the Suspicious lists are less than the propagation time of the SavePic lists. This different is because of the larger payload of the SavePic list, which has two

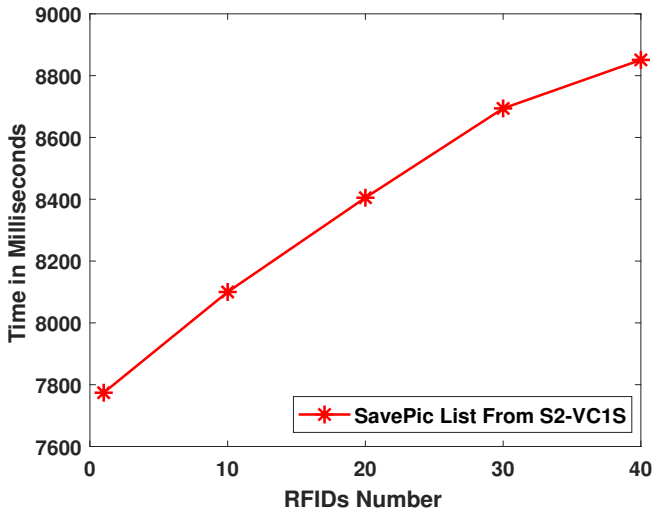

Figure 13: Propagation Time of SavePic List from S_2 until $VC1S$

speeds for each one RFID, than the Suspicious list, which has only one speed for each RFID.

7 DISCUSSION

AWS IoT does not have full capability to implement our use case that we employed in [3]. First, virtual objects (shadows) in AWS IoT can not communicate directly to each other. Since virtual objects can only subscribe to their reserved topics update, get, and delete. So, they receive data through their reserved topics. Also, virtual objects can publish to their reserved topics only whenever they receive data. As a result, publishing and subscribing of virtual objects is only to their reserved topics and direct publishing to unreserved topics or irrelevant topics is not applicable.

There are several indirect ways to allow virtual objects communication in AWS IoT. One way to allow two virtual objects to communication is to attach a rule with the update topic of first virtual object that triggers a republish action to the second virtual object update topic. The Republish ction can be also used to forward data to AWS services as shown in Figure 14. Another way is to attach a rule with a topic of first virtual object that trigger a lambda function, which can do complex computation, such as publishing data, getting data, and comparing data. Thus, lambda function can republish the received data to another topic, which could be the update topic of the second virtual object. We employed the second way in our use cases.

In addition to indirect communication among virtual objects, virtual objects in AWS IoT cannot keep track of old data. For example, if a new suspicious list is published to a virtual object, the current suspicious list will be deleted and the new one will be saved. However, our use case needs to combine the coming suspicious list from previous sensor and the current saved one. Since the process of deleting and saving list is very fast, triggering a lambda function that get the current suspicious list from virtual object and then

combine it with the coming one did not work. Thus, virtual objects in AWS IoT can not save old data.

There are several ways to keep track historical data in AWS IoT. One way to keep track historical data of a virtual object is to have another relative virtual object that works as storage. The only way to get or publish data to the relative virtual object is by allowing one lambda function to publish and get data from it. This lambda function is triggered whenever data is published to update topic of the virtual object. Thus, the *coming* suspicious list arrived to the update topic of a virtual and the *current* suspicious list that is saved in the virtual object can be combined and republished by the lambda function. We used this way in our use case to keep transit data within the virtual object layer, so the privacy of data can be reserved. Another way to reach the historical data of a virtual object is to trigger a republish action to AWS DynamoDB whenever data is published to update topic of the virtual object. Thus, authorized virtual objects can get the historical data from AWS DynamoDB as needed. However, our use case tends to keep the suspicious lists within the virtual object until at least two sensors report the speed of a car to be over limit. Figure 14 shows the way of republishing suspicious lists, which come from S_1 and S_2, to AWS DynamoDB in the cloud service layer. Then, $lambda_1$ is authorized to get all suspicious lists and check if there are duplicated RFIDs within the saved suspicious lists and also within the suspicious list coming from the camera. If so, this RFID is declared to be an over-limit car, and it is reported along with consolidate information from all suspicious lists (speed, picture).

Another issue with the AWS IoT is that virtual objects cannot do complex computation on the data they receive. They only save the recent published desired or reported data. Such a computation in our use case can not be implemented within only AWS IoT. Thus, since we need the transit data to be only within AWS IoT, we used AWS Lambda service to support doing the needed computation. Another way to do that is to send data to DynamoDB and allow an application or a third party to do the needed computation.

Moreover, in our use case, we could have up to n sensors. $Lambda_3$ to $Lambda_{(n-1)}$ functions are repetitive functions that can be triggered by the update topic of $VS3$ to $VS_{(n-1)}$. We can get rid of this repetition If we have only one Lambda that accept passing the name of the published sensor. However, triggering same lambda is not working within our use case, because to our knowledge there is no way to pass the name of the published sensor to a lambda function. Thus, repeated copies of $Lambda_{(n-1)}$ will be increased by increasing the number of sensors, which is n.

8 CONCLUSION

In this paper, we studied AWS IoT and developed the access control model for virtual objects (shadows) communication in AWS IoT (AWS-IoT-ACMVO). We used the AWS-IoT-ACMVO to implement two scenarios of the use case that is employed in ACO-IoT-ACMsVO: the simple use case of sensing the speed of one car with two sensors and the use case of sensing the speed of multiple cars with multiple sensors. By implementing these two scenarios using ACO-IoT-ACMsVO, we determined how to configure the policies and control virtual object communication of our proposed model. The time to propagate information about suspicious cars and over-limit

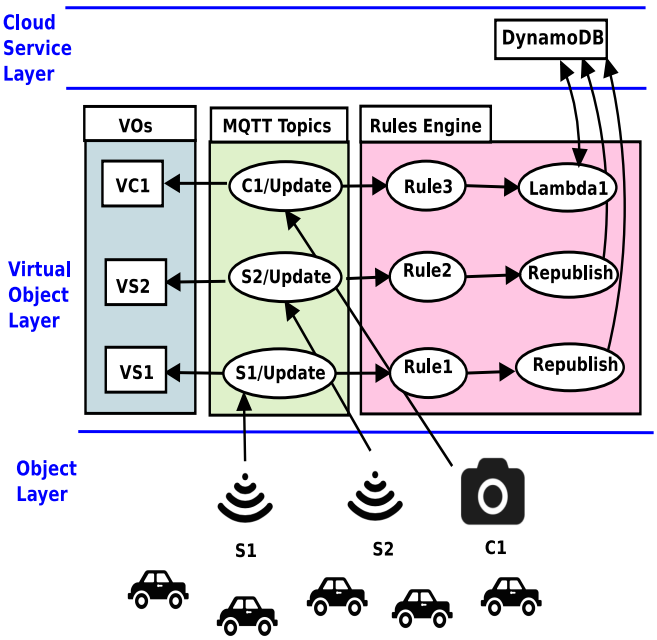

Figure 14: A Different Way of VOs Communication and Data Computation

cars through all virtual objects is measured and discussed. Finally, upon our study and implementation, we offered a discussion of AWS IoT issues and suggestions of enhancing VOs communication and their access control.

ACKNOWLEDGMENT

This research is partially supported by NSF CREST Grant HRD-1736209, NSF Grants CNS-1111925, CNS-1423481, CNS-1538418, and DoD ARL Grant W911NF-15-1-0518.

REFERENCES

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Comm. Surveys & Tutorials* 17, 4 (2015), 2347–2376.

[2] Asma Alshehri and Ravi Sandhu. 2016. Access Control Models for Cloud-Enabled Internet of Things: A Proposed Architecture and Research Agenda. In *the 2nd IEEE International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 530–538.

[3] Asma Alshehri and Ravi Sandhu. 2017. Access Control Models for Virtual Object Communication in Cloud-Enabled IoT. In *The 18th International Conference on Information Reuse and Integration (IRI)*. IEEE.

[4] Jean Bacon, David M Eyers, Jatinder Singh, and Peter R Pietzuch. 2008. Access control in publish/subscribe systems. In *the Second International Conference on Distributed Event-Based Systems*. ACM, 23–34.

[5] 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.

[6] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. 2014. On the integration of cloud computing and internet of things. In *IEEE Int. Conf. on Future Internet of Things and Cloud (FiCloud)*. 23–30.

[7] Li Da Xu, Wu He, and Shancang Li. 2014. Internet of things in industries: A survey. *IEEE Trans. on Indust. Informatics* 10, 4 (2014), 2233–2243.

[8] Patrick Th Eugster and et all. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[9] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and

future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645–1660.

[10] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. 2012. Future internet: the internet of things architecture, possible applications and key challenges. In *10th IEEE Int. Conf. on Frontiers of IT*. 257–260.

[11] Michele Nitti, Virginia Pilloni, Giuseppe Colistra, and Luigi Atzori. 2015. The Virtual Object as a Major Element of the Internet of Things: a Survey. *IEEE Communications Surveys & Tutorials* 18, 2 (2015), 1228–1240.

[12] Pritee Parwekar. 2011. From internet of things towards cloud of things. In *2nd IEEE Int. Conf. on Comp. and Comm. Tech.* 329–333.

[13] BB Prahlada Rao, Paval Saluia, Neetu Sharma, Ankit Mittal, and Shivay Veer Sharma. 2012. Cloud computing for Internet of Things and sensing based applications. In *Sixth IEEE Int. Conference on Sensing Technology (ICST)*. 374–380.

[14] Rodrigo Roman, Jianying Zhou, and Javier Lopez. 2013. On the features and challenges of security and privacy in distributed internet of things. *Computer Networks* 57, 10 (2013), 2266–2279.

[15] Yun Zhang, Farhan Patwa, and Ravi Sandhu. 2015. Community-based secure information and resource sharing in AWS public cloud. In *2015 IEEE International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 46–53.