FACT: Functionality-centric Access Control System for IoT Programming Frameworks

Sanghak Lee
POSTECH
uzbu89@postech.ac.kr

Jiwon Choi
POSTECH
wldnjs7@postech.ac.kr

Jihun Kim
POSTECH
jihun735@postech.ac.kr

Beumjin Cho
POSTECH
beumjincho@postech.ac.kr

Sangho Lee
Georgia Tech.
sangho@gatech.edu

Hanjun Kim
POSTECH
hanjun@postech.ac.kr

Jong Kim
Pohang University of Science and Technology (POSTECH)
jkim@postech.ac.kr

ABSTRACT

Improvement in the security and availability is important for the success of the Internet of Things (IoT). Given that recent IoT devices are likely to have multiple functionalities and support third-party applications, this goal becomes challenging to achieve. Through an in-depth investigation of existing IoT frameworks, we focused on two inherent security flaws in their design caused by their device-centric approaches: (1) coarse-grained access control and (2) lack of resource isolation. Because of the coarse-grained access control, IoT devices suffer from over-privileged applications. Furthermore, the lack of resource isolation allows the possibility of Denial-of-Service attacks.

In this paper, we propose a functionality-centric approach to managing IoT devices, called FACT, which has two design goals, namely, the principle of least privilege and the availability in terms of device functionalities. FACT isolates each functionality of the device using Linux Containers and grants a subject the privilege to access for each required functionality. We provide the overall framework and detailed working procedures between components that constitute FACT. We built a prototype of FACT on IoTivity and show that it accomplishes secure and efficient linkages between applications and functionalities of IoT devices through analysis and experiments.

CCS CONCEPTS

• **Security and privacy** → **Trust frameworks**; **Access control**; Denial-of-service attacks;

SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA
 ACM ISBN 978-1-4503-4702-0/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3078861.3078864

KEYWORDS

Internet of Things; Functionality-centric; Access control; Over-privileged application; Denial-of-Service

ACM Reference format:
Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. 2017. FACT: Functionality-centric Access Control System for IoT Programming Frameworks. In *Proceedings of SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA, ,* 12 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078864

1 INTRODUCTION

The Internet of Things (IoT) has emerged as a leading technology, and the scale of its expansion is overwhelming. It is only a matter of time until connecting every device to the Internet would become natural. The International Data Corporation (IDC) forecasts that 28 billion IoT devices (or things)[1] will be installed by 2020 [19]. Many companies (e.g., Google, Samsung, and Qualcomm) establish their frameworks and standards (e.g., AllJoyn, Android Things, IoTivity, and SmartThings), and release various IoT devices to dominate the IoT market [27].

Security is one of the most important requirements of the IoT systems. Among the 184 requirements issued by Internet of Things Architecture (IoT-A), 52 requirements (roughly 30%) are relevant to security [28]. In practice, however, IoT devices hardly support existing security mechanisms originally designed for servers, personal computers, or even smartphones, because of not only their constrained resources but also their diverse features (e.g., smart TVs with payment and social networking supports). Accordingly, IoT devices are becoming easy and attractive attack targets [8]. Thus, security mechanisms that consider the features of IoT devices have become imperative.

Furthermore, we observe the following two trends in the recent IoT developments. First, the number of supporting

[1]We use the terms 'device' and 'thing' interchangeably throughout this paper.

functionalities of IoT devices are increasing. Second, IoT devices tend to interact with third party applications. For example, a healthcare IoT device can have many sensors (e.g., pace and pulse sensors) and interact with diverse third party applications to for logging or alarming. Previous IoT frameworks adopt a device-centric approach that gives a user or an application either *all or no* permissions to use IoT devices because they usually have single or a few functionalities. Without selectively controlling accesses to each functionality (i.e., without satisfying the principal of least privilege), IoT devices would suffer from over-privilege and Denial-of-Service (DoS) attacks performed by third-party applications. We need a new access control approach for IoT devices to come up with their recent development trends providing and controlling *multiple* functionalities.

In this paper, we first conduct a case study to know how two popular IoT frameworks, SmartThings and IoTivity, implement access control mechanisms. We confirm that they suffer from the over-privilege and DoS problems we mentioned since their access control mechanisms are based on a device-centric approach. Motivated by the case study results, we propose FACT, a functionality-centric access control system to manage IoT devices securely.

In FACT, the basic unit of control and usage is not the device, but the *functionality*. This functionality-centric approach resolves the security problems of the existing IoT frameworks. First, FACT examines whether an application has the privilege to access the functionalities it requests to prevent unauthorized access. Users no longer have to worry about security problems derived from unprivileged applications. Second, FACT isolates each functionality of IoT devices to maximize the overall availability of the functionalities. Despite the attack of a malicious or compromised application to a functionality (e.g., DoS attack), the isolation prevents the attack from affecting the remaining functionalities such that the functionalities can be provided to other applications.

We implement a prototype of FACT on IoTivity, which is one of the popular open-source IoT frameworks. The evaluation results confirm that FACT satisfies our security goals with minimal overhead.

This work makes the following contributions:

- **Novel study.** To the best of our knowledge, FACT is the first functionality-centric approach to protecting IoT devices from third-party applications. Given that recent IoT devices are likely to have many functionalities and interact with third-party applications, our approach is necessary for fostering a secure IoT environment.
- **Fine-grained access control.** FACT prevents unauthorized applications from using any disallowed functionalities of IoT devices. Malicious or compromised applications are restricted from accessing the unauthorized functionalities of IoT devices.
- **Functionality Isolation:** FACT separates each functionality, which restrains the functionalities from affecting each other. Thus, resource exhaustion attacks on a functionality cannot harm the availability of the other functionalities.

This paper is organized as follows. Section 2 provides the investigation of existing IoT frameworks and their problems. Section 3 presents the security flaws in the existing IoT frameworks, threat models, and our design goals. Section 4 presents the overview and detailed working procedures of components that constitute the proposed FACT framework. We present the implementation and evaluation results in Section 5. Sections 6 and 7 provide discussion and related works. Finally, we conclude in Section 8.

2 EXISTING IOT FRAMEWORKS

In this section, we investigate how the existing IoT frameworks establish a connection between a host and an IoT device. We classify the IoT frameworks into two types: commercial IoT frameworks and open-source frameworks. For each type, we select the most dominant framework: SmartThings[2] and IoTivity[3]. We analyze the frameworks with respect to access control models and the basic units used for binding between the subject and the object, which is summarized in Table 1.

2.1 SmartThings

Overview. SmartThings is a commercial IoT framework that integrates heterogeneous IoT ecosystems. It supports around 170 IoT devices and communication protocols (e.g., Zigbee and Z-Wave). A simple form of the SmartThings architecture consists of *SmartThings hub*, *SmartApps*, and *Device handlers*.

- SmartThings hub: acts as a gateway for connected IoT devices by connecting devices directly to a home network router. The hub is compatible with diverse communication protocols such as Zigbee, Z-Wave, and IP-accessible devices.
- SmartApp: provides the interface that allows users to operate the functionalities of connected IoT devices. A user can download and use it on smartphones, called SmartThings Mobile.
- Device handler: represents the virtual wrapper of physical devices.

SmartThings supports a web-based programming environment where app developers and device vendors can implement SmartApps and Device handlers.

Capability. A capability in SmartThings is the basic unit of authorization. It consists of two elements, namely, a set of *attributes* and *commands*. *Attributes* represent the properties of a device. *Commands* are ways that a user can control the device. For example, a *door control* capability has a *door status* attribute and two commands, *open()* and *close()*. A SmartApp has to declare a capability to connect with a device. Then, the system scans for the Device handlers that support the requested capability and asks the user to select

[2]https://www.smartthings.com/
[3]https://www.iotivity.org/

Table 1: Summary of access control mechanisms in IoT frameworks

Framework	Access Control (AC)	Connection Unit	Binding Units	
			Subject	Object
SmartThings	Account-based AC	Capability	SmartApp	Device handler
IoTivity	Device-based AC	Resource	Client device	Resource

a device from the scanned devices for binding. In this model, we consider a capability as the basic unit of connection.

Access Control. SmartThings provides a hierarchical framework for its security. *Accounts* (i.e., SmartThings users) are at the top of the framework. Under *accounts*, there are *locations* such as an office or a home. In general, SmartThings hubs are located in these *locations*. Under *location*, there are *groups* that represent physical spaces such as rooms. Finally at the bottom, there are *devices* that belong to a certain *group*. Once a user logs into a SmartApp with his account and grants the location permission on the host device, the application automatically gains access to a specified device through a SmartThings hub, which is bound to a *location*.

Most of the SmartApps contain the capability lists defined in their code. Once a SmartApp is installed via SmartThings Mobile, it asks a user to select one among the devices (Device handler) that contain the requested capability, and the selected device would be bound to the SmartApp.

2.2 IoTivity

Overview. IoTivity is an open-source IoT framework that enables seamless device-to-device connectivity to address the emerging needs of the IoT. It follows the Open Interconnect Consortium (OIC) standard specifications[4] and is available on various platforms [32]. The IoTivity architecture contains *servers*, *clients*, and *resource hosting devices*.

- Server: represents an IoT device or a hub device that aggregates the data of connected IoT devices. A server provides various functionalities of IoT devices to clients.
- Client: represents a user or a user device that attempts to access IoT devices (servers).
- Resource hosting device: helps clients to discover the address of servers and monitors server status. In general, this role is taken by gateway devices (e.g., routers for Wi-Fi communication).

IoTivity developers implement client and server applications with IoTivity API. A server application called a resource runs on an IoT device and handles requests from client applications on external devices.

Resource. The unit of connection and control is a *resource* in IoTivity. A resource consists of three elements: *identity*, *property*, and *attribute*. *Identity* is a uniform resource identifier that consists of each device's address and path. *Property* includes each device's resource type or name defined by a server, and its interface type, (e.g., the Internet and Bluetooth). *Attribute* is a key-value data of functionalities (e.g., temperature, humidity, and an air circulation mode).

Access control. To determine whether a client has the right to access a server's resource, a server maintains security information such as an access control list (ACL). Each access control entry (ACE) in an ACL consists of *subject ID*, *resource*, and *permission*. *Subject ID* is the identity of a client device. *Resource* is the resource type of the server. *Permission* is a type of the client's privilege (e.g., read only, write only, or both read and write) to access the server's resource.

When the server receives a request from the client, the policy engine in the server conducts the following procedure. First, it looks up ACL with the subject ID in the request. Next, it searches the ACE that matches with the resource in the request, and checks whether the matching subject (client) has the permission. Finally, it either grants or denies access to the client.

2.3 Problems in Frameworks

We discuss the problems in two frameworks.

SmartThings. Fernandes *et al.* [13] analyzed the design of SmartThings and found several security flaws related to overprivilege and sensitive data (event) leakage. Furthermore, we investigate other design flaws of SmartThings. We concentrate on three problems among the identified security defects.

- SmartApp can access all capabilities of the implemented by Device handler of the selected devices, not only the requested capability.
- SmartApp can monitor any event data published by the Device handler.
- With the location permission on the host device, SmartApp gains access to location-bounded devices through the SmartThings hub.

IoTivity. We investigate the design of IoTivity access control and find three security flaws.

- Resource, the unit of connection and control, has a number of attributes, and IoTivity cannot grant different access policies to each attribute.
- All attributes data of the resource are stored in the same process or file system.
- Subject ID in the ACL maps to a client device, not an application. Thus, IoTivity cannot grant different access policies to each application in the client device.

[4] https://openconnectivity.org/resources/specifications

3 DESIGN FLAWS AND GOALS

3.1 Design Flaws

We organize a number of security flaws in IoT frameworks and focus on the design flaws that cause the over-privileged application and the availability problems in SmartThings and IoTivity.

3.1.1 Coarse-grained Access Control. The two IoT platforms implement the following access control models: an account-based access control and a device-based access control, which result in account-to-device and device-to-device authentication respectively. These models make an access control decision by checking whether a subject (an account or a host device) who sends a request has the privilege to access the corresponding device (a thing). Thus, we call this security model as a device-centric approach.

In the device-centric approach, the basic unit of control and usage is a device. Fig. 2 shows a conventional device-centric access control model in which a user has an IoT device providing three functionalities and aims to use it via a client application or device. The user can either allow or disallow an application to access the IoT device depending on their functionalities. Note that the device-centric access control takes an all-or-nothing approach; therefore, the user cannot specify which functionalities the client can use.

The device-centric access control is suitable for devices that have a single functionality. In the past, most IoT devices (e.g., sensors) have a single functionality; therefore, they did not require any fine-grained access control mechanisms in terms of the functionality. However, given the prevalence of devices with multiple functionalities (e.g., smartwatches), the coarse granularity of the device-centric access control introduces security problems. For example, in Fig. 2, a user wants to use the fn_c functionality. If the user binds the *Client* with $Thing_c$, which has a single functionality, fn_c, no over-privileged problem will occur. However, if the *Client* connects to $Thing_a$ with multiple functionalities, the *Client* will be granted access to all functionalities defined in $Thing_a$, which include not only fn_c but also other functionalities fn_a and fn_b that do not correspond to the user's original intention. Furthermore, the device-centric access control disallows an application to access a device to protect privacy-sensitive functionalities, resulting in utility degradation. For example, if a user disallows an application to access one of the privacy sensitive functionalities, then the application will be restricted from accessing all of the functionalities declared in the application including the privacy-insensitive functionalities (e.g., clock, temperature, and humidity). Given that IoT devices deal with both public and private information simultaneously, this becomes a serious problem.

3.1.2 Lack of Resource Isolation. An IoT device conceptually separates functionalities to provide only the requested access to the subject. Nevertheless, some concerns remain regarding availability, because most IoT frameworks and devices hardly support any resource isolation techniques. For example, a malicious application can exhaustively request a certain functionality to an IoT device (a type of DoS attack). These requests can exhaust the resource (e.g., CPU, memory, and storage) of the devices by running a computation-intensive process or abusing a log system. Moreover, because of developers' inexperience in mobile and IoT fields, recently released applications exhibit disruptive behaviors such as checking the status of the device too frequently and causing the devices' processor keep spinning [18]. Considering that the IoT hubs or devices do not have sophisticated request handling mechanisms, these malicious attempts can eventually damage the availability by freezing or shutting down a device [25].

3.2 Threat Models and Design Goals

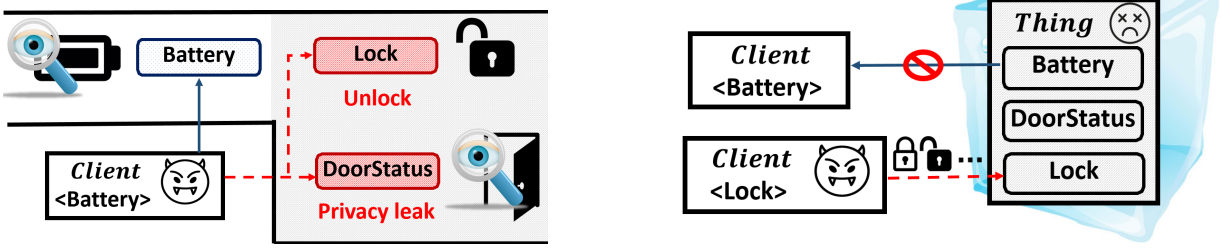
3.2.1 Threat Models. By exploiting the problems derived from the design flaws in the existing IoT frameworks, we consider two types of attacks, called *misusing the functionality* and *reducing the functionality*, can be performed against IoT devices. Note that we target application-level attacks on the IoT frameworks; therefore, other types of attacks including network and hardware attacks are out of the scope of this paper.

First, over-privileged applications can abuse the functionalities of IoT devices. An attacker attempts to access unauthorized functionalities of the IoT devices using a malicious or compromised application. For example, we assume a smart doorlock that has 1) battery status monitoring, 2) door status monitoring (open or closed), and 3) door locking and unlocking functionalities (Fig. 1a). A user wants to check a smart doorlock's battery status through an application; therefore, he/she grants the application the privilege to access the smart doorlock (e.g., the SmartApp declares a battery capability for SmartThings, and the client is registered to the ACL of the smartLock resource for IoTivity). Unfortunately, the application is malicious and has a backdoor to exploit other functionalities of the attached doorlock. While showing the battery status information to a user to fulfill the original objective, the application can check the lock status of the door and even stealthily unlock the door when it is locked.

Second, an attacker can freeze or cease the IoT devices. We revisit the smart doorlock case (Fig. 1b) to illustrate this attack scenario. The user runs two applications on his smartphone; *App*1 requests a functionality to monitor battery status. *App*2 asks functionalities to (un)lock the doorlock. However, if *App*2 repeatedly requests the smart doorlock device to lock and unlock, *App*1 would have no way to monitor the battery status.

3.2.2 Design Goals. To solve the problems derived from the design limitations in the existing IoT frameworks, we aim to provide a novel access control mechanism that encompasses multiple functionalities of IoT devices with the following design goals.

- *The principle of least privilege*: The access control mechanism has to be able to grant privileges to a subject as least as it requests to an object.



(a) Challenges in device security because of application backdoors that abuse unauthorized functionalities

(b) Challenges in device availability because of lack of functionality separation

Figure 1: Attack scenarios induced by design flaws of existing IoT frameworks

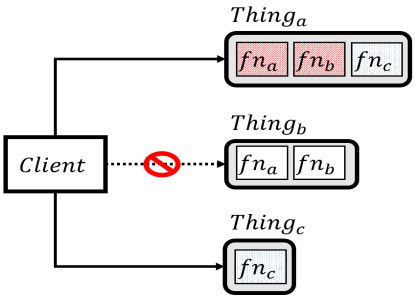


Figure 2: Device-centric access control in IoT frameworks (*Client*: client application or device, fn: functionality, and *Thing*: device)

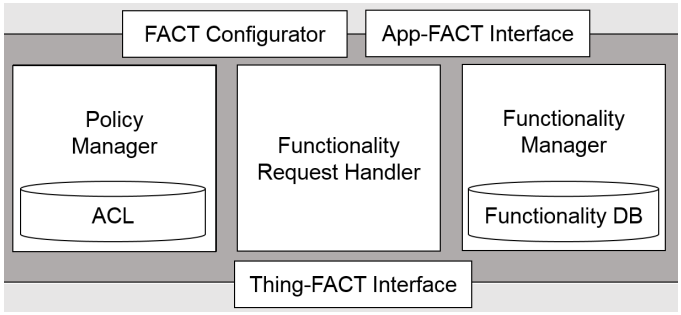


Figure 3: Overview of FACT. (ACL: Access Control Lists, App: Application)

- *Availability*: The access control mechanism has to guarantee the availability of an IoT device even when it suffers from a subject's disruptive requests.

4 FACT

In this section, we propose FACT, a fine-grained functionality centric access control system for IoT frameworks. FACT is deployed on IoT devices, hubs, and clouds to achieve the goals defined in Section 3.2.2. It allows IoT frameworks to grant a subject *the least privilege* by providing only the requested functionalities. Furthermore, FACT isolates the functionalities from each other so that any disruption in one functionality would not affect the *availability* of the other functionalities.

4.1 Overview

Components. FACT mainly consists of six components (Fig. 3).

- **Functionality Request Handler (FRH):** A component to securely connect the application with authorized functionalities by interacting with Policy Manager and Functionality Manager shown below.
- **Policy Manager:** A component to manage the functionality permission information of each application.
- **Functionality Manager:** A component to manage the overall functionalities of currently registered IoT devices.
- **Application-FACT Interface (AFI):** An interface between applications and FACT to register applications and demanded functionalities.
- **Thing-FACT Interface (TFI):** An interface between IoT devices and FACT to manage IoT devices and process-requested functionalities.
- **FACT Configurator:** An administration application to manage the overall settings of FACT.

Procedure. The overall procedures of FACT are as follows. First, an IoT device is registered to FACT through the Functionality Manager. The Functionality Manager identifies the functionalities that the attached IoT devices provide and initiates a server for each functionality. Next, the user registers his application to FACT through the Policy Manager, which checks the functionalities requested by the application. Then, it asks the user to select a device with the requested functionalities he/she prefers to use with the application. Once the user selects a device, the Policy Manager asks the user to permit the applications to use the requested functionalities in the selected device. Finally, the application sends a request to the FRH. To check if the request is valid, the FRH searches for the applications' permission stored in the ACL of the Policy Manager. If the request is valid, the FRH communicates with the Functionality Manager to approve the actions corresponding to the request.

4.2 Functionality

FACT uses a *functionality* as a minimal object unit that represents an independent service entity. Functionalities are comprised of two types: *sensing* and *actuating*. A *sensing* functionality has only one method that receives sensor's data or status information. An *actuating* functionality can have multiple methods (e.g., lock and unlock in a smart doorlock device). To provide a fine-grained access control, FACT grants a client application different policies on different functionalities. For example, recall the smart doorlock, which has three functionalities, i.e., *battery*, *door status*, and *lock*. FACT can authorize client applications only to read lock's status, while prohibiting access to other functionalities.

4.2.1 Requesting Functionalities. To simplify the application registration process and reduce the developer's burden, we propose the FACT policy language (FPL). An application developer has to write all the required methods corresponding to the desired functionalities in FPL form and include it in the application's description. When a user or a system wants to register a client application to FACT, it finds things that support the requested functionalities.

Grammar. FACT adopts the Backus-Naur Form (BNF) notation [23] for context-free grammar that effectively expresses the clients' requested methods.

We define the syntax of FPL as shown below:

$$\langle \textit{policy rule} \rangle ::= \varnothing \mid \langle \textit{functionality list} \rangle$$

$$\langle \textit{functionality list} \rangle ::= \langle \textit{functionality} \rangle \mid \langle \textit{functionality} \rangle \text{`,'} \langle \textit{functionality list} \rangle$$

$$\langle \textit{functionality} \rangle ::= \langle \textit{sensing} \rangle \mid \langle \textit{actuating} \rangle$$

As mentioned previously, FACT considers two types of functionalities, namely, *sensing* and *actuating*. A *sensing*-type functionality represents an action to detect any occurrence of events or change of status. An *actuating*-type functionality represents an action that causes the device to move its component or change its status. To reflect these characteristics, each functionality can have different methods according to its type as follows:

$$\langle \textit{sensing} \rangle ::= \langle \textit{sensing name} \rangle \text{ `<'} \langle \textit{sensing method} \rangle \text{`>'}$$

$$\langle \textit{sensing name} \rangle ::= \textit{string}$$

$$\langle \textit{sensing method} \rangle ::= \textit{'getStatus'}$$

A *sensing* functionality has only one method which is receiving its sensor's status data. Note that functionality is a minimal object unit such that a sensor functionality cannot cover more than one sensor's data.

$$\langle \textit{actuating} \rangle ::= \langle \textit{actuating name} \rangle \text{ `<'} \langle \textit{actuating methods} \rangle \text{`>'}$$

$$\langle \textit{actuating name} \rangle ::= \textit{string}$$

$$\langle \textit{actuating methods} \rangle ::= \textit{'all'} \mid \langle \textit{method list} \rangle$$

$$\langle \textit{method list} \rangle ::= \varnothing \mid \textit{'getStatus'} \mid \textit{'setStatus'} \mid \langle \textit{method name} \rangle \text{`,'} \langle \textit{method list} \rangle$$

$$\langle \textit{method name} \rangle ::= \textit{string}$$

An *actuating* functionality has multiple methods (a set of operations). It may contain vendor-defined methods, for an action which cannot be controlled through simply setting the status of the actuator. The FPL supports the vendor-defined methods by allowing a method type as a string (line 6). Furthermore, an application developer can conveniently declare device's own methods for the *actuating* functionality by adding 'all'. With this command, the developer does not need to register all methods of the target functionality (line 3).

4.2.2 Examples. In our previous example, the smartLock device has three functionalities, namely *battery*, *doorStatus*, and *(un)lock*. In FACT, the functions to obtain battery and door status information (*battery*, *doorStatus*) are *sensing* functionalities, while the actions to lock and unlock the door (*(un)lock*) are *actuating* functionalities.

Sensing. 1) battery - method: getStatus (battery remaining percentage), 2) doorStatus - method: getStatus (whether door is open or closed)

Actuating. 1) lock - method: setStatus (lock and unlock)

For a battery monitoring application, it needs to know the battery status only.

```
1 description{
2   battery<getStatus>
3 }
```

With this description, FACT can show a user the list of devices that have battery functionality that retrieves battery status. Then, the user selects one of the devices from the list to register the application to FACT. After registration, the application can access only the battery status of the selected device.

For an auto-lock application, the application locks a door when the door is closed and unlocked. To perform this job, the application requires a set of functionalities including the retrieval of the door and lock statuses as well as to lock the door.

```
1 description{
2   doorStatus<getStatus>,
3   lock<getStatus, setStatus>
4 }
```

In the case of an administration application, it requires all methods of functionalities.

```
1 description{
2   battery<getStatus>,
3   doorStatus<all>,
4   lock<all>
5 }
```

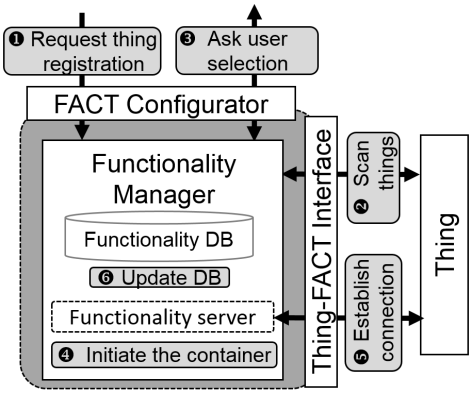


Figure 4: The procedures of how an IoT device and its functionalities are registered to FACT

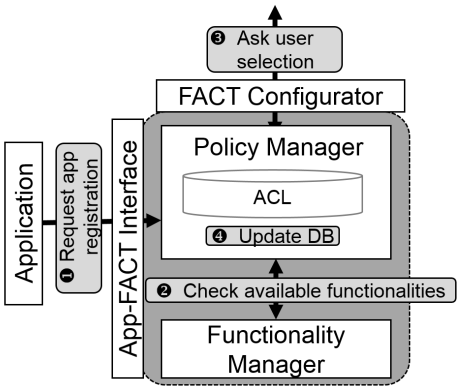


Figure 5: The procedure of how an application and its functionality demands are registered to FACT

Table 2: Functionality DB (Func ID: Functionality ID)

Thing ID	Func ID	State
`smartLock`	`lock`	`Active`
`smartLock`	`battery`	`Active`
`smartBulb`	`switch`	`dormant`
`smartBulb`	`lightColor`	`dormant`
`temperatureSensor`	`temperature`	`Active`
`humiditySensor`	`humidity`	`Active`

Table 3: Access control lists

App ID	Func ID	Methods
`lockapp`	`lock`	`SetStatus`
`lockapp`	`battery`	`GetStatus`
`bulbapp`	`switch`	`SetStatus`
`bulbapp`	`lightColor`	`SetStatus`
`airConapp`	`temperature`	`getStatus`
`airConapp`	`humidity`	`getStatus`

4.3 Management of Functionalities

4.3.1 Functionality Isolation. To isolate each functionality from others, FACT applies *Linux Containers (LXC)* [16], one of virtualization techniques. LXC makes programs portable and isolated by packaging them in containers. It solves some problems such as dependency conflicts and platform differences, but we focus on its security benefits that isolate a container from a host and other containers. LXC virtualizes at the operating system level, whereas hypervisor-based techniques virtualize at the hardware level. Thus, LXC requires less computing and memory overheads than other hardware-level virtualization [11]. Therefore, LXC is a suitable isolating technique for the IoT frameworks, which fits for low-computing power devices. FACT creates some containers according to the number of functionalities. Given that FACT makes one functionality server run on each container, the container sandboxes its contained functionality from the other functionalities.

4.3.2 Managing Thing Functionalities. We explain how the Functionality Manager enables a user to register and update functionality information of his/her IoT devices (Fig. 4).

❶ The thing registration process is initiated periodically. If the process is initiated by a user, it skips ❷-❸ which are device discovery processes, and goes to ❹.

❷ In the case of periodical registration, the Functionality Manager scans nearby IoT devices via TFI to discover new devices.

❸ The Functionality Manager asks a user's consent to register the discovered devices.

❹ If the user approves, the Functionality Manager creates some containers according to the number of functionalities and sets the default amount of resources (e.g., CPU, memory, and storage) to the containers automatically if the user does not assign specific values to them.

❺ Functionality servers establish a connection with the selected device. These servers can interact with external things via TFI.

❻ After the establishment, the Functionality Manager updates the *functionality DB* that stores the provided thing ID, functionality ID, and connection states as shown in Table 2.

In the case when the firmware is updated, an IoT device may obtain new functionalities. To handle the added functionalities, the Functionality Manager needs to update its DB. The update procedure is the same as the registration process except that it only deals with the new functionalities. Thus, we do not explain its details.

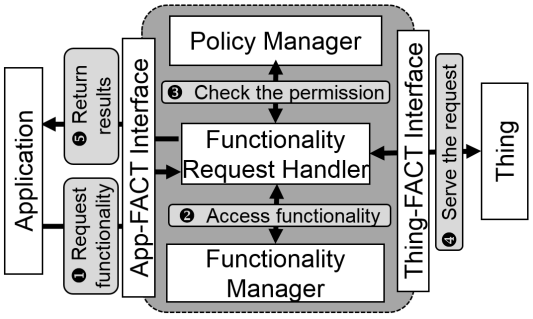


Figure 6: The procedure of how an application's functionality request is checked and delivered to an selected device

Some low-power IoT devices may not provide detailed information and malfunctioning devices may provide misinformation. Therefore, to obtain the complete and exact information of IoT devices, the Functionality Manager needs to interact with the external trusted server.

4.3.3 Managing Application Functionalities. The Policy Manager registers a newly installed application that is aware of FACT as shown in Fig. 5.

- ❶ The application contacts the Policy Manager via AFI to register itself with a list of functionalities it demands.
- ❷ The list can contain functionalities a user's IoT devices do not support, i.e., *not-available functionalities*. To exclude them from the list, the Policy Manager asks the Functionality Manager about the currently available functionalities.
- ❸ The Policy Manager displays the filtered list to the user and lets the user selectively grant permissions to the application.
- ❹ The Policy Manager updates the ACL according to the user's decision. The DB records the application ID, functionality ID, and the requested methods of the functionality as shown in Table 3.

The Policy Manager updates its DB when an application is updated, or a user wants to change the permission settings. The update procedure is the same as the registration procedure except that it only deals with new or changed functionality information. Therefore, we omit the detailed explanation on updating application information.

4.3.4 Processing Functionality Requests. We explicate the process of how FRH securely delivers an application's functionality request to the selected IoT device as shown in Fig. 6.

- ❶ A client application sends a functionality request to the FRH through AFI.
- ❷ The FRH connects to a container that includes the server deals with the requested functionality.
- ❸ The FRH asks the Policy Manager whether a user has permitted the application to access the requested functionality.

```
1  Response requesthandler(request){
2    switch (request.func.method){
3      case "getStatus":
4        if (!aclCheck(request.AppID,
             getStatus))
5          break;
6
7        return server.getStatus();
8
9      case "SetStatus":
10       if (!aclCheck(request.AppID,
             setStatus))
11         break;
12
13       return server.setStatus(request.
             setValue);
14
15     case default:
16       if (!aclCheck(request.AppID,
             request.func.method))
17         break;
18
19       return server.action(request.func.
             method);
20   }
21     return error("Access Denied");
22 }
23
24 bool aclCheck(AppID, method){
25   return pairMatch(server.id, AppID,
       method);
26 }
```

Listing 1: Request handling process

- ❹ If the application has a privilege to access the requested functionality, the FRH conveys the functionality request to the device through TFI. Otherwise, the FRH discards the request.
- ❺ The device returns the result to the application.

Listing 1 shows a request handling process after the FRH has accessed the requested functionality server. The FRH determines whether the requested method is *getStatus*, *setStatus*, or vendor-defined. Afterward, the FRH checks the permission through the ACL of the Policy Manager. These procedures are described in Lines 4, 10, and 16 of Listing 1. If the ACL has no matching ACE with the application's request, then the FRH returns an error message to the application (Line 21). If the request is legitimate, the FRH interacts with the things via the functionality server.

5 IMPLEMENTATION AND EVALUATION

In this section, we elaborate on how we implemented a prototype of FACT described in Section 4 and evaluate its security effectiveness and performance overhead.

5.1 Implementation Details

We implemented a prototype of FACT on Raspbian Jessie[5] with IoTivity version 1.2.1. We developed two IoT devices, one for a server and the other for a client device. For the server, we used RaspberryPi 3 connected to an ultrasonic sensor, a temperature sensor, an infrared light motion sensor, and a Phillips Hue smartbulb, which has switch and color change functionalities. The sample server applications that we have built were running on the device. The total components of the server consisted of 472 Source Lines of Code (SLoC). The SLoC associated with the functionality registration and method checks were 75 and 95 respectively. The server stored the ACLs in Samsung 32GB EVO Class 10 Micro SDHC Card (MB-MP32DA/AM). For the client, we built applications on another RaspberryPi 3 device. The total components of the client applications consisted of 196 SLoC. The SLoC related to the functionality discovery and functionality requests were 110 and 86 respectively.

To separate each functionality from the other functionalities, we adopt Docker[6] [24] to apply container techniques for functionality isolation. Using Docker, FACT can insulate each functionality server in each container and regulate the amount of resources for the container.

5.2 Security Evaluation

In our security evaluation scenario, a device owner wants to permit a client application to access the ultrasonic sensor, the temperature sensor, and the smartbulb's switch functionality, whereas disallow access to the infrared light motion sensor and the smartbulb's color change functionality. The owner has recorded corresponding permission rules in the ACLs according to above scenario (Table 4). We tested whether FACT can protect the IoT device from unpermitted functionality access. Furthermore, we conduct application-level DoS attacks on IoT devices, which apply FACT.

Over-privileged access prevention. The executed application on the device attempted to discover the resource and access the ultrasonic, temperature sensors, and the smartbulb's switch functionality, as well as the restricted infrared light motion sensor and the smartbulb's color change functionality. Without FACT, the IoT device could not distinguish the unpermitted functionality access from the permitted access. Even though the application only had access to the ultrasonic or temperature functionalities, the application was able to access all functionalities when the device owner granted the applications the privilege to access the IoT device. With FACT, the IoT device successfully differentiated access to each functionality. When the application attempted to access the device functionalities, the device checked the ACL to prevent access to unpermitted functionalities. As a result, the IoT device only granted access to the ultrasonic and temperature functionalities, and access to the infrared light motion functionality was successfully thwarted.

[5] https://www.raspberrypi.org/
[6] https://www.docker.com/

Table 4: Functionality permission rules in the scenario

App ID	Func ID	Methods
lockapp	ultrasonic	GetStatus
airConapp	temperature	GetStatus
airConapp	infrared light motion	✗
bulbapp	switch	SetStatus
bulbapp	changeColor	✗

Table 5: Effectiveness of FACT preventing attacks

Attack type	Attack description	Effectiveness
Misusing	Over-privileged application	✓
Reducing	Resource exhaustion	✓
	Packet flooding	✗

Availability. An application with a permission still can cause problems by generating excessive access to the functionality either by programming mistakes or with malicious intention. We conduct storage exhaustion attacks on the temperature functionality server by exploiting the sensor data log system. Without FACT, it disrupts the device's other functionalities because the device's resources are exhausted by the attacks. With FACT, the functionality isolation allows the device to confine how much resources of the device each functionality can consume (e.g., CPU, memory, and storage), and thus prevent the misbehaving functionality from disturbing other functionalities. Therefore, our fine-grained access control guarantees the availability of functionalities.

Furthermore, we conduct the UDP flood attack on a functionality, which is one of the popular DoS attacks. FACT is not able to protect an IoT device against the attack. However, this type of attack is out of scope because it is related to a network layer, not an application. Several researchers investigate the defending mechanisms against flooding-based DoS attack [1, 3, 12, 34]. Thus, when these mechanisms are combined with our system and applied to the IoT environments, the security and the availability of IoT devices can be more robust and reliable.

5.3 Performance Evaluation

FACT handles multiple functionality servers and uses Docker to apply container techniques for functionality isolation. We measured the performance overheads caused by the management of multiple functionalities in FACT.

We varied the supported functionalities of a device from 1 to 30 and checked the latency and memory overhead 10 times each in two cases: 'without Docker' and 'with Docker' cases. We considered the 'without Docker' scenario because currently some operating systems (e.g., Windows 7 and 8) do not support container techniques yet. Nevertheless, FACT without Docker can be applied on such OSes and still prevent over-privileged applications. In the 'Without Docker' case, we executed a server process per functionality to keep the

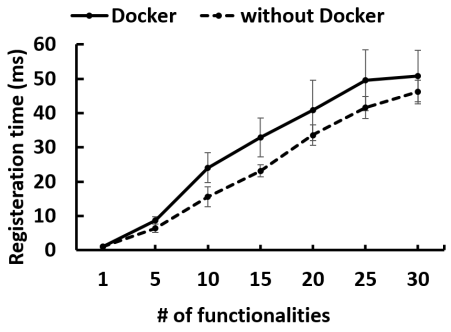


Figure 7: Registration time according to the number of functionalities

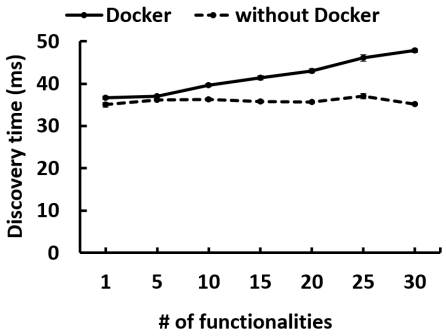


Figure 8: Discovery time according to the number of functionalities

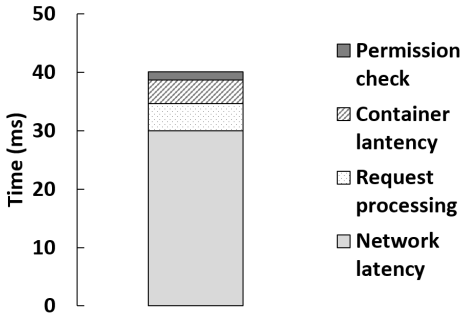


Figure 9: Breakdown of the overall communication latency with 1,000 permission rules in ACL

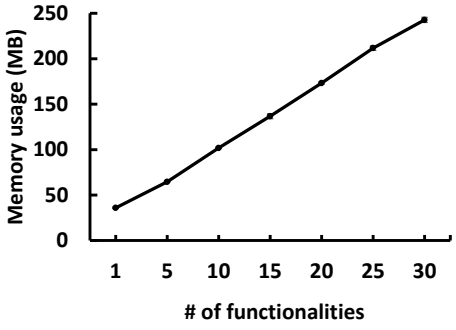


Figure 10: Memory usage according to the number of functionalities with Docker

functionality-centric access control without containers. In the 'With Docker' case, we made a container per functionality and executed a server in each container to separate the resources from the other functionality containers.

Latency Overheads. We measured the amount of time it took to register a thing, discover functionalities of the thing, and access a functionality. The thing registration time increased as the number of functionalities increased in both cases (Fig. 7) due to multiple updates in the functionality DB. Furthermore, Docker brings about 8 ms latency overhead because a registration request goes through via Docker bridges.

The functionality discovery time is constant (about 35 ms) in the 'without Docker' case. In contrast, the discovery time depends on the number of managed functionalities in the 'With Docker' case (Fig. 8). It should be note that as the number of containers that a hub device contains increases, the communication overhead between in and out of containers also increases.

We also measured the request processing latency when a client requires a functionality. The request processing latency is 4.59 ms in the 'without Docker' case and 8.64 ms in the 'with Docker' case. However, the latency difference between the two cases has a low impact because the network latency is the dominant latency (over 85%) among the overall latency (Fig. 9).

Note that our experiments were performed in a small area (about 1 m^2). The network latency will increase in case of distant scenarios, and the percentage of the overhead becomes trivial compared to the entire latency in real situations.

Memory Overhead. Fig. 10 shows the memory usage with the increased number of functionalities in the 'with Docker' case. The Docker system spent 28 MB of memory and it spent an additional 8 MB of memory per container when the number of functionalities increases. We argue that the memory overhead of FACT is within acceptable limits for the hub devices, because personal PCs, smartphones, and even SmartThings hubs (contains 512 MB RAM) have enough memory to afford the additional memory usages.

When we executed a simple server application in the host and the container, the average memory usages of the server application in the host and in the container case were 310 KB and 344 KB, respectively. The container affects the additional memory usage of the server application about 30 KB, which is negligible for IoT hubs or clouds.

6 DISCUSSION

In this section, we discuss heterogeneous communication channels, operating system dependency, and the hub necessity.

Heterogeneous communication channels. Despite conducting our evaluation on Wi-Fi for communication, FACT can adapt to other communication channels such as Bluetooth, because most IoT frameworks serve APIs that cover heterogeneous communication channels. We only need to modify the communication APIs or parameters (e.g., interface types) associated with the communication channels.

Operating system dependency. The isolation technique of FACT relies on Docker, which is LXC-based, such that the operating system of the server devices should be a Linux-based system (e.g., CentOS, Raspbian, or Ubuntu) to work without any problems. However, Docker starts to support MacOS and Windows. Recent Windows (e.g., 64-bit Windows 10 Pro) can use Docker, and we expect that its coverage would expand in the near future.

Hub necessity. We show that FACT applies to hub-based IoT frameworks only. However, we can adopt FACT to device-to-device (distributed) environments if a thing that serves functionalities has abilities to make containers and run server processes. We expect that multiple functionality devices (e.g., smart watch) have sufficient computing power and memory to satisfy those abilities. Thus, the devices could adopt FACT.

7 RELATED WORKS

In this section, we explain some studies related to FACT: IoT security and over-privileged applications.

IoT security. Some researchers have proposed new IoT systems that consider security. HomeOS [9] is an operating system for a smart home, using a new type of abstraction to provide extensibility and management. It considers all devices in a home as peripherals. BOSS [6] is an operating system for a smart building with many IoT devices. It focuses on how to manage the relationship between components including sensors, location, and time. SIFT [22] is a safety-centric programming platform to build safe IoT environments. It especially focuses on how to manage a large number of safety rules while avoiding conflicts between the rules.

Given that smartphones are usually considered as the hub of IoT systems, some researchers have enhanced smartphones' security systems to secure the IoT systems. Dabinder [26] was the first study that considered the security problem of Android when managing external IoT devices. It has found that malicious applications can access any IoT devices by just obtaining a communication permission in the Android system. SEACAT [7] provided an effective solution to the problem by enhancing the Security-Enhanced Android (SEAndroid) [31] to distinguish external resources when defining and enforcing access control lists. Busold et al. [2] proposed context-aware service mobility frameworks that enable users to securely distribute the functionality of the application to mutually untrusted smart devices on Android. Levy et al. [21] proposed Beetle, a new hardware interface that allows many-to-many secure connections between peripheral Bluetooth devices and applications .

Furthermore, as the IoT becomes a reality, some studies have analyzed the security defects of IoT devices and frameworks, and how to protect them securely. Grant et al. [17] examined the security of commercially-available smartlocks. Ronen et al. [30] proposed extending the functionality attack in the case of smart lights. Fernandes et al. [13] analyzed the security design flaws of SmartThings. They have shown that SmartThings applications, called SmartApps, support capability-based device scanning, but allow to access the whole capabilities of each scanned device because of coarse-grained access control.FlowFence [14] is a system that requires consumers of sensitive data to declare their intended data flow patterns using Quarantined Module. ContexIoT [20] is a context-based permission system for IoT platforms that provides contextual integrity by supporting context identification for sensitive actions.

The current approaches of IoT frameworks continue to be problematic because they do not consider each functionality of IoT devices. Therefore, many IoT frameworks cannot satisfy the principle of least privilege for IoT devices with multiple functionalities. To the best of our knowledge, FACT is the first functionality-centric approach that prevents applications from abusing any disallowed functionalities of IoT devices.

Over-privileged applications. Many researchers have considered security problems due to over-privileged applications. For example, researchers have discovered that many over-privileged applications exist on application markets [4, 10]. Moreover, they can abuse other components, such as advertisement libraries [15, 29, 35] and other benign applications for privilege escalation [5]. Furthermore, Tuncay et al. [33] raised the problems that originated in coarse-grained access control of in-app embedded browsers. Note that the explained studies have considered over-privilege problems within the Android system. To the best of our knowledge, FACT is the first approach to solving a new over-privileged problem regarding device functionalities.

8 CONCLUSION

Given that recent IoT devices are likely to provide multiple functionalities and interact with third-party applications, a new security mechanism to protect sensitive functionalities effectively from malicious applications is crucial. In this paper, we proposed a functionality-centric access control mechanism for IoT frameworks, called FACT. In FACT, a user can grant an application access to each functionality of IoT devices to fulfill the principle of least privilege in terms of device functionalities. Furthermore, FACT guarantees the availability of IoT devices by isolating each functionality of the device using LXCs from the other functionalities. The novel functionality-centric access control system proves that it can effectively guarantee the security and the availability for IoT frameworks. We implemented a prototype of FACT on IoTivity and showed that it satisfied our design goal with minimal overhead.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their invaluable comments and suggestions. This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1403-04.

REFERENCES

[1] Alexander Afanasyev, Priya Mahadevan, Ilya Moiseenko, Ersin Uzun, and Lixia Zhang. 2013. Interest flooding attack and countermeasures in Named Data Networking. In *IFIP Networking Conference, 2013*. IEEE, 1–9.

[2] Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N Asokan. 2015. Smart and Secure Cross-Device Apps for the Internet of Advanced Things. In *International Conference on Financial Cryptography and Data Security*. Springer, 272–290.

[3] Rocky KC Chang. 2002. Defending against flooding-based distributed denial-of-service attacks: a tutorial. *IEEE communications magazine* 40, 10 (2002), 42–51.

[4] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. 2012. Is This App Safe?: A Large Scale Study on Application Permissions and Risk Signals. In *International Conference on World Wide Web (WWW)*.

[5] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. Privilege Escalation Attacks on Android. In *International Conference on Information Security (ISC)*.

[6] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. 2013. BOSS: Building Operating System Services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[7] Soteris Demetriou, Xiaoyong Zhou, Muhammad Naveed, Yeonjoon Lee, Kan Yuan, XiaoFeng Wang, and Carl A. Gunter. 2015. What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources. In *Network and Distributed System Security Symposium (NDSS)*.

[8] Nitesh Dhanjani. 2015. *Abusing the Internet of Things: Blackouts, Freakouts, and Stakeouts*. O'Reilly Media, Inc.

[9] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. 2012. An operating system for the home. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[10] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *ACM Conference on Computer and Communications Security (CCS)*.

[11] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 171–172.

[12] Paul Ferguson. 2000. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. (2000).

[13] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *Security and Privacy (S & P), 2016 IEEE Symposium on*. IEEE, 636–654.

[14] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security Symposium*.

[15] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

[16] Matt Helsley. 2009. LXC: Linux container tools. *IBM devloperWorks Technical Library* (2009), 11.

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

[18] Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. 2016. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 221–234. DOI:http://dx.doi.org/10.1145/2906388.2906419

[19] International Data Corporation (IDC). 2014. IDC Market in a Minute: Internet of Things. http://www.idc.com/downloads/idc_market_in_a_minute_iot_infographic.pdf. (2014).

[20] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*.

[21] Amit A Levy, James Hong, Laurynas Riliskis, Philip Levis, and Keith Winstein. 2016. Beetle: Flexible communication for bluetooth low energy. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 111–122.

[22] Chieh-Jan Mike Liang, Bor̈je F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: Building an Internet of Safe Things. In *International Conference on Information Processing in Sensor Networks (IPSN)*.

[23] Daniel D McCracken and Edwin D Reilly. 2003. Backus-naur form (bnf). (2003).

[24] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[25] S. Misra, M. Maheswaran, and S. Hashmi. 2016. *Security Challenges and Approaches in Internet of Things*. Springer International Publishing. https://books.google.co.kr/books?id=-Rz4DAAAQBAJ

[26] Muhammad Naveed, Xiaoyong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A. Gunter. 2014. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Bonding on Android. In *Network and Distributed System Security Symposium (NDSS)*.

[27] Colin Neagle. 2014. A guide to the confusing Internet of Things standards world. http://www.networkworld.com/article/2456421/internet-of-things/a-guide-to-the-confusing-internet-of-things-standards-world.html. (2014).

[28] Internet of Things Architecture (IoT-A). 2015. Requirements. http://www.meet-iot.eu/iot-a-requirements.html. (2015).

[29] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

[30] Eyal Ronen and Adi Shamir. 2016. Extended functionality attacks on IoT devices: The case of smart lights. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 3–12.

[31] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS)*.

[32] Ashok Subash. 2015. IoTivity – Connecting Things in IoT. In *TIZEN Development Summit*.

[33] Guliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter. 2016. Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 104–115.

[34] Abraham Yaar, Adrian Perrig, and Dawn Song. 2004. SIFF: A stateless internet flow filter to mitigate DDoS flooding attacks. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE, 130–143.

[35] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: Isolating Advertisements from Mobile Applications in Android. In *Annual Computer Security Applications Conference (ACSAC)*.