ComFlex: Composable and Flexible Resource Management for the IoT

Gyeongmin Lee, Bongjun Kim, Seungbin Song, Seonyeong Heo, Hanjun Kim Member, IEEE

Abstract—The Internet of Things (IoT) enables new services such as smart home and smart healthcare integrating various resources of networked devices. However, managing the IoT resources is very challenging because a device may have multiple different resources and a service accesses subsets of the resources in multiple devices. To simplify the resource management, existing IoT frameworks provide resource models and protocols for manufacturers and programmers, but their access granularity is too coarse-grained causing overprivilege problems, or too fine-grained causing management overheads. To avoid the overprivilege problems without the management overheads, this work proposes a new composable and flexible resource management scheme, and implements its prototype compiler-runtime framework called ComFlex. The ComFlex compiler allows manufacturers to register their devices as composition of fine-grained resources by inheriting existing interfaces of the resources, and programmers to define their own access granularity as a virtual resource that consists of the fine-grained resources of multiple devices. The ComFlex runtime supports fine-grained access control without additional overheads by mapping the fine-grained resources of the virtual resource into physical resources of different devices. To evaluate the ComFlex framework, this work implements 52 resources on 14 devices and six IoT services with ComFlex and an existing fine-grained resource management scheme. Compared to the existing scheme, the ComFlex framework supports the IoT services with 40.2% less lines of code and 80.9% less discovery time, without any overprivilege problem and response time delay.

I. INTRODUCTION

Despite the promising applicability of the Internet of Things (IoT), managing the IoT resources is very challenging because a device may have multiple different resources and a service accesses subsets of the resources in multiple devices. Since a device may have multiple resources, its manufacturer should abstract and register all the accessible resources and their properties to IoT platforms. For example, Figure 1 illustrates a smart pet care service that allows people to remotely interact with their pets. The pet care device in the service has different types of resources such as a camera, a microphone, a speaker, and a pet feeder, so its manufacturer needs to abstract and register all the accessible resources and their properties into standard camera, microphone, speaker and actuator APIs of IoT platforms.

Moreover, since an IoT service integrates parts of resources in multiple devices, programmers should manage access control of the distributed resources suffering from a huge amount of the programming burden. In the pet care service example, its programmer should implement codes to search the screen resource, to achieve its access permission from the mobile device, and to send video streams. Here, the programmer must get the access permission only for the required resources to avoid overprivilege problem. If the pet care service achieves access permission for files in the mobile device, the pet care service may leak private data in the mobile. Thus, programmers should write access control lists (ACL) of required resources, which increase the programming burden.

Existing works [1]–[13] have proposed IoT platforms that reduce the resource management burden of manufacturers and programmers. Some IoT platforms [10], [13] support composable or flexible resource registration for device abstraction and provide library functions to manage the resources in the service programs. However, since the platforms use a coarse-grained access control granularity, allowing a service to access a whole device that include a required resource, they suffer from an overprivilege problem [14], [15]. To avoid the overprivilege problem, other existing frameworks [1], [8], [11] propose fine-grained resource management approaches that allow manufacturers to register fine-grained resources and their properties, and allow programmers to manage each resource in a fine-grained access control granularity. However, fine-grained resource management approaches cause a huge programming burden to the manufacturers and programmers because they should manage the registration, access control, and remote control for a large number of fine-grained resources. Moreover, fine-grained resource management approaches cause additional resource discovery delay.

To avoid the overprivilege problem without additional programming and performance overheads, this work proposes a new composable and flexible resource management scheme, and implements its prototype compiler-runtime cooperative
framework called ComFlex. With two annotations that mark the resources and their accessible properties, the ComFlex compiler allows manufacturers to implement fine-grained resources by extending existing resources, to register a device as a composition of fine-grained resources, and to abstract a device with various granularity. Moreover, the compiler allows programmers to define their own management granularity as virtual resources with another annotation. To provide fine-grained access control for each service easily, the compiler automatically analyzes the service source codes and generates access control lists for each service. The ComFlex runtime automatically registers a device with containing resources, accessible properties, and various abstraction. Moreover, the runtime registers, deploys, and executes programmer-defined virtual resource instances to make programmers manage multiple different fine-grained resources with virtual resources.

This work implements the ComFlex framework on top of the LLVM C++ compiler infrastructure [16], and 14 IoT devices based on 52 resources and six IoT services with an existing fine-grained resource management scheme and ComFlex. This work measures the programmability, resource discovery time, and event response time delay of the frameworks. The evaluation results show that ComFlex supports the IoT services with 40.2% less lines of code and 80.9% less discovery time, without the overprivilege problems and the response time delay.

The contributions of this paper are:
- a new composable and flexible resource management scheme that avoids the overprivilege problem without additional programming and performance overheads,
- a compiler-runtime cooperative framework called ComFlex that implement the composable and flexible resource management,
- and in-depth evaluation with 52 resources on 14 devices and six IoT services.

II. BACKGROUND & MOTIVATION

This section describes composability and flexibility of the IoT resources in devices (Section II-A) and services (Section II-B), and introduces the problems of the existing IoT platforms in the resource management (Section II-C).

A. Composability and Flexibility of IoT Resources in Devices

**Composability**: A device is a composition of multiple different IoT resources. For example, SmartWatch in Figure 1 contains resources such as a GPS, a notification module, and a heart rate checker. To notify an alarm to a user through the SmartWatch, the PetCare service needs to discover and access the notification module in the SmartWatch. Thus, the manufacturer should register each resource of a device to IoT platforms, making the resources discoverable and accessible.

**Flexibility**: Different IoT resources share the same features while having their own specific features. For example, Mobile and SmartTV have the same screen resource as ‘m-scr’ and ‘l-scr’ in Figure 1. Although the screen resources from different vendors have different specific features such as size and portability, the resources share common functionalities as a screen.

Thus, when an IoT service searches a screen, both resources should be searchable. However, when another service searches a portable screen, only ‘m-scr’ should be searched.

B. Composability and Flexibility of IoT Resource in Services

**Composability**: A service discovers and accesses a subset of resources in multiple devices. For example, the PetCare service in Figure 1 discovers and accesses screens and a speaker, a notification module, and secure payment in multiple devices such as Mobile, SmartWatch, and SmartTV. Programmers should implement codes for resource discovery and access control lists (ACLs) to discover and access required resources only.

**Flexibility**: Various IoT services use similar subsets of the resources but with different purposes. This work analyzes 267 SmartThings services [17], [18], and finds that various IoT services use similar subsets of the resources as Figure 2 illustrates. However, although the services use the same resources, the services access different properties or functionalities of the discovered resources. Therefore, to prevent indiscreet access, programmers should write different ACLs in a fine-grained granularity for each service for the same subset of resources.

C. Problems of Existing IoT Platforms

To effectively manage IoT resources, IoT platforms should support composability and flexibility of IoT resources in devices and services. IoT platforms should allow manufacturers to easily register their device as a composition of different resources, and each resource as different types. Moreover, the platforms should provide a simple way to integrate multiple resources in different devices and make the integrated resources shared across different services with different ACLs.

However, existing platforms [1]–[13] do not manage IoT resources in a composable and flexible way at various resources.
management steps such as resource registration, resource discovery and access control grant. Table I summarizes their device management models.

**Resource Registration**: Existing IoT platforms [2]–[7], [12] allow manufacturers to register a device in a device or resource granularity. However, most of them do not allow to register a device as a composition of multiple different resources, nor to flexibly register a resource in different types. Some IoT platforms [1], [8], [9], [11], [13] allows device registration as a composition of multiple resources, but they do not allow a resource to have multiple types. Esperanto [10] can support flexible device registration by inheriting existing device programs, but it does not support composable resource registration. Thus, all of the existing IoT platforms cannot support composable and flexible resource registration, causing additional programming burden due to manual resource registration.

**Resource Discovery**: When an IoT service uses multiple resources together, all the existing platforms [1]–[13] force the service to search each resource individually without supporting a composable search. Moreover, almost all the platforms except Esperanto [10] do not support flexible resource discovery that allows a service to find a resource with one of its multiple types. For example, with the flexible resource discovery, an IoT service can discover ‘m-scr’ with either a screen type or a portable screen type in Figure 1.

**Access Control**: Most IoT platforms [2]–[7], [9], [12], [13] provide an access control protocol and generate ACLs. However, their permission grant granularity is too coarse-grained, giving unnecessary grants to a service and causing the overprivilege problem [14], [15]. To avoid the overprivilege problem, some IoT platforms [1], [8], [11] propose fine-grained access control protocols that support a function and variable granularity. However, the fine-grained access control increases the programming burden because it forces programmers to manually write ACLs for each IoT service.

![Image](https://via.placeholder.com/150)

**TABLE I**

**Comparison of Device Management Models in Existing IoT Frameworks. C, F and Auto stand for composableility, flexibility and automatic ACL generation.**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Device Registration</th>
<th>Device Discovery</th>
<th>Permission Grant</th>
<th>Access Control List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Granularity</td>
<td>C</td>
<td>F</td>
<td>Granularity</td>
</tr>
<tr>
<td>Esperanto [10]</td>
<td>Device</td>
<td>X</td>
<td>O</td>
<td>Device</td>
</tr>
<tr>
<td>ComFlex [This Paper]</td>
<td>Resource</td>
<td>O</td>
<td>O</td>
<td>Resource</td>
</tr>
</tbody>
</table>

**TABLE II**

**ComFlex Syntax**

<table>
<thead>
<tr>
<th>Syntax of the ComFlex primitives</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Declaration</td>
<td>#pragma resource className;</td>
</tr>
<tr>
<td>Virtual Resource Declaration</td>
<td>#pragma vresource className;</td>
</tr>
<tr>
<td>Open API Declaration</td>
<td>#pragma open { void functionName(); }</td>
</tr>
</tbody>
</table>

**III. ComFlex Language**

To efficiently support composable and flexible resource management, this work proposes the ComFlex language. Exploiting the correspondence between composable and flexible IoT resources and composition and inheritance features of the object oriented programming (OOP) model, the ComFlex language extends an OOP model such as C++ programming language. The proposed language allows manufacturers to register a device as a composition of multiple resources using composition in the OOP model, and to abstract a resource or device with a flexible granularity by inheriting interfaces of existing resources. Moreover, the language also allows programmers to define their own resource type by composing multiple resources in different devices using composition of the OOP model, so a service can discover and access resources through the newly defined resource type. Finally, the language allows inheritance of the newly defined type and proposes an annotation to support flexible and fine-grained access control.

The ComFlex language extends the existing C++ language with three annotations. Despite the dependence between the ComFlex language and C++ language, the basic idea of the extension can be applied to other OOP languages. Table II describes the proposed three annotations.

**resource** annotates a class that declares an IoT resource. Since the class represents a resource type, its object represents an instance of an IoT resource. For example, if a manufacturer annotates resource on PetCare class in Figure 3, the ComFlex framework declares a resource named PetCare,
and considers an instance of PetCare as a PetCare resource. To support flexible resource registration, the framework finds parent classes of PetCare annotated as resource, and registers the resource as the parent types also. To support composable resource registration, the framework allows a resource class to have other resources as member variables, and registers the resources together if they are annotated as open.

`vresource` annotates a class that represents a programmer-defined custom resource, called virtual resource. With the `vresource` annotation, programmers can newly define their own resource type by composing multiple resources and virtual resources, and a service can discover and access resources through the newly defined virtual resource, allowing composable resource discovery. Like resource, a vresource can inherit other virtual resources, allowing flexible resource discovery. For example, figure 4 shows that a programmer defines a virtual resource VirtualVideoPlayer that has fine-grained resources such as Speaker and Screen. If a service get an access permission on VirtualVideoPlayer, the service automatically has authority to access functions and variables of fine-grained member resources.

`open` annotates discoverable and accessible member functions and variables of a resource or vresource class. In figure 3, the manufacturer annotates `open` on multiple functions and variables, and an IoT service can invoke the open functions such as `getVideo` and `feed` that control the PetCare resource. Moreover, if the manufacturer annotates a member variable like "PetFeeder* pfeeder" that is also annotated as resource, the ComFlex framework considers the variable as a discoverable and accessible resource. Here, the framework analyzes the source codes of the service and its virtual resources, and automatically generates ACLs to avoid indiscreet access to resources from the service.

### IV. COMFLEX COMPILER

This work proposes a compiler-runtime cooperative framework to support the ComFlex language. The ComFlex compiler compiles two kinds of programs such as device program and service program. The two compilation processes share a common logic, called resource compilation. Section IV-A describes how the resource compiler analyzes the resources and virtual resources and generates event handlers, interfaces and metadata files. Section IV-B shows how the ComFlex compiler compiles a device program, and Section IV-C describes how the compiler compiles a service program and automatically generates its ACLs.

#### A. Resource Compiler

Figure 5(a) illustrates the overall structure of the resource compiler that consists of five modules such as Parser, Resource Analyzer, EventHandler Generator, Interface Generator and Customizer.

**Parser** parses the annotations in the ComFlex language. The parser marks classes that are annotated with `resource` and `vresource` as resource and virtual resource types. Moreover, the parser generates a list of accessible functions and variables annotated with `open` to identify properties that can be accessed by a service.

**Resource analyzer** analyzes composable and flexible relations among resources. First, to analyze the composable relation, the resource analyzer inspects all the accessible member variables that represent the resources. For the example code in Figure 3, the resource analyzer finds PetSpeaker and PetFeeder resources in the PetCare class, and reports...
the composition relation between them. Second, to analyze flexible relation, the resource analyzer checks parent classes of a resource class. For example, since PetSpeaker inherits its parent class Speaker in Figure 3, the resource analyzer reports Speaker as a flexible abstraction of PetSpeaker. Finally, the resource analyzer generates a resource metadata file about the analysis results.

**EventHandler generator** generates an event handler function for each resource. For an IoT service to execute the accessible functions of a resource, a device program needs to have its event handler function that receives requests from the service and executes appropriate functions. Given a list of accessible functions from the parser, the EventHandler generator creates event handler functions like evt_handler_PetCare in Figure 6. The event handler function parses a request from a service, selects its target function, executes the function with received arguments, and sends its return value to the service. Moreover, the event handler function executes read and write requests for accessible member variables if the requests are proper.

**Interface generator** generates a resource interface between an IoT service and accessible functions of a resource. Since there can exist multiple resource instances for a single resource type, when a service invokes a resource function, the resource interface needs to identify its corresponding instance. Since an object in the service represents an instance of the IoT resource, the ComFlex runtime maps the object memory address to its resource instance. Using the map and the memory address, the resource interface identifies its target resource and sends the request (line 7 in Figure 7).

**Customizer** receives the results of the resource analyzer, the EventHandler generator and the interface generator, customizes the results on target platforms such as a Cloud server and a fog node and generates resource metadata, event
handlers and resource interfaces.

B. Device Program Compilation

Figure 5(b) shows the overall structure of the ComFlex compiler for manufacturers. After the resource compiler generates resource metadata, event handler functions and interfaces, the ComFlex compiler generates the device binary with two steps.

First, the compiler generates a device-level event handler function that redirects requests to corresponding resource-level event handlers. At line 13 in Figure 6, evt_handler indicates the device-level event handler. evt_handler checks the resource type of a request, and redirects the request to the corresponding resource-level event handler.

Second, the compiler generates a device registration function like __registerDevice in Figure 6. The device registration function registers all the member resources in the device, invoking their creators like PetCare in Figure 6. The creator registers the instance with its memory address, and the ComFlex runtime maps the memory address into the physical resource. Finally, the device registration function registers all the composition and abstraction between resources.

C. Service Compilation

Figure 5(c) shows the compilation flow of a service program with virtual resources. The service program compilation proceeds in two steps such as ACL generation and service registration.

To provide different fine-grained access control for each service while the multiple services use the same virtual resource, the ComFlex compiler analyzes all the accessed functions and variables from each service, and automatically generates an ACL for the service. Moreover, the compiler analyzes all the read and write operations in the service program including remotely executed function, and reflects the analysis result to the ACL generation. For example, the PetCare service in Figure 4 invokes the playVideo function of VirtualVideoPlayer that is a virtual resource. In the playVideo function, the VirtualVideoPlayer invokes playAudio and playVideo functions of speaker and screen resources. The ComFlex compiler generates a white list of accessible variables and functions as ACLs for the PetCare service. If there is any read or write operation for a member variable annotated with open, the compiler adds the variable type into the ACLs with a read or write property. In the example, the compiler adds Speaker and Screen into the ACLs with write property due to playAudio and playVideo function invocations. Moreover, the compiler adds playAudio and playVideo functions in the ACLs to support fine-grained resource access control. Here, since the generated ACLs contain only type information without physical resource information, the ComFlex runtime needs to map each type to an appropriate instance during resource discovery procedure.

The ComFlex compiler automatically inserts codes to register service itself with the generated ACLs. At line 13 in Figure 7, __registerServiceInstance registers the service name and gets the unique service ID. Moreover, __registerACL registers the generated ACLs to the ComFlex runtime to regulate the indiscreet access from malicious services to the resources.

V. COMFLEX RUNTIME

The ComFlex runtime supports automatic registration of IoT devices and services, provides efficient resource discovery, and allows IoT services to control physical resources of IoT devices with negligible delay.

A. Overall Structure

Figure 8 shows the overall structure of the ComFlex runtime. The ComFlex runtime consists of three components, interface and binary manager, discovery manager, and access control manager. The interface and binary manager contains interfaces of (virtual) resources and executable binaries of IoT services and virtual resources. The discovery manager receives resource discovery requests from installed IoT services and discovers appropriate resources with type information in resource metadata. The access control manager checks remote function call and variable accesses from IoT services with static ACLs from the ComFlex compiler and reports access violation.

B. Device and Service Registration

The ComFlex runtime installs executable binaries and interfaces that the ComFlex compiler generates into proper locations. For an IoT device, the ComFlex runtime allows manufacturers to download and install device binaries to the IoT device. A device binary contains multiple event handlers. Each event handler responds to requests from the service to each resource in the device. Since this paper assumes a cloud-centric model for the IoT platform, the runtime installs interfaces and executable binaries into the interface and binary manager in the cloud. Moreover, the runtime updates ComFlex resource metadata that contains composable and flexible relations between the resources and static ACLs from the services.

After the installation, a device binary registers itself. First, the device binary sends a list of its containing resources. Moreover, the device registers composable and flexible relationship among containing resources to the interface and binary manager, and the discovery manager. For example, a laptop device that contains two resources such as speaker and screen registers itself in Figure 8 (a). The laptop registers its two resources into the interfaces and binary manager, and updates the resource table in the discovery manager with the resource type and instance name (identifier). With this information, the discovery manager can distinguish each resource registered to the interface and binary manager. Moreover, the laptop updates ComFlex resource metadata in the discovery manager and static access control lists (ACLs) in the access control manager.

When an user executes an IoT service, the IoT service binary registers itself to the ComFlex runtime. First, the service adds a newly defined virtual resource type in the service to the resource table of the discovery manager. Moreover, the
service adds a new row in the access control table in the access control manager with its service identifier. For example, a new PetCare service registers a newly defined virtual resource type named VirtualVideoPlayer to the resource table in the discovery manager, and adds its service name, PetCare, as a service identifier to the access control table in Figure 8 (a).

C. Resource Discovery

To find available resources of IoT devices, an IoT service requests resource discovery to the ComFlex runtime. Resource discovery process consists of four steps such as sending discovery request, checking flexible types, checking access control lists, and generating accessible resource lists.

**Sending discovery request:** To allow the service to initiate resource discovery, the ComFlex runtime provides a function named `getResources`. `getResources` takes a resource type as an argument and returns a list of available resources. With `getResources`, an IoT service can send a resource discovery request to the discovery manager.

**Checking flexible type:** To support flexible resource discovery, the discovery manager of the ComFlex runtime checks child resource types of the target resources, and adds them into target resource lists. For example, if the PetCare service requests screen type resources, the discovery manager adds child resources of screen such as laptop screen and TV screen into the target resource lists.

**Checking ACLs:** The access control manager checks whether the service is allowed to access the target resources and their child resources with static ACLs that contain access control lists with type information. If some resource types are not allowed to access for the service, the ComFlex runtime excludes that resource types in the resource discovery process.

**Generating accessible resource lists:** The discovery manager searches resources with only allowed resource types using the resource table in the discovery manager and resource instances in the interface and binary manager. If the target resource type is a virtual resource type, the ComFlex runtime gives two options to users. The runtime allows users to choose existing virtual resources with the same type, or to create a new virtual resource instance. When the user wants to create a new virtual resource instance, the runtime shows users available (virtual) resources to bind with the newly created virtual resource in a composable manner.

For example, Figure 8 (b) shows overall process of resource discovery. When the PetCare service tries to find SecurePay resource, the ComFlex runtime checks resource metadata in discovery manager to enable flexible resource discovery. Then, access control manager examines access
authority of PetCare service with static ACL. Lastly, discovery manager finds a SecurePay instance installed in interface and binary manager. In case of virtual resource, VirtualVideoPlayer, the ComFlex runtime creates a new VirtualVideoPlayer2 instance instead of choosing existing instance used by GameManager service. The runtime allows users to bind Laptop Speaker and Screen with newly created VirtualVideoPlayer1. Despite the two services use their own VirtualVideoPlayer instance, they share the unified interface of VirtualVideoPlayer and control each instance in flexible manner.

D. Remote Function Call

The ComFlex runtime allows an IoT service to control physical resources of IoT devices with (virtual) resource interfaces. In resource discovery, the ComFlex runtime binds the IoT service with appropriate (virtual) resources through the interfaces. The IoT service just calls a function of an interface to execute function logic remotely in an IoT device. The ComFlex runtime supports network communication between the service and the device. For example, the PetCare service wants to relay video through laptop, then it calls playVideo function of VirtualVideoPlayer interface. The ComFlex runtime also delivers the arguments of function call through the interfaces to resource instances such as Laptop Speaker and Laptop Screen. Then, the runtime sends function call signal and arguments to target IoT devices using registered IP address.

VI. Evaluation

This work implements the ComFlex compiler-runtime cooperative framework on top of the LLVM compiler infrastructure [16]. To evaluate the ComFlex framework, this work designs and implements 6 IoT services in the ComFlex language, and deploys the services on Amazon Elastic Compute cloud (Amazon EC2) instance (t2.micro). Moreover, this work implements 14 IoT devices with 52 resources, and runs each device binary on embedded devices, ODROID-XU4. The cloud runs Ubuntu 16.04.4 LTS, and the embedded devices run Ubuntu Make 1.10.2.

A. Evaluated IoT Services

To evaluate the ComFlex language and compiler framework, this work designs 6 IoT services as benchmarks such as pet care, hvac, smart alarm, sleep care, door lock, and game manager which support 14 events in total. To guarantee the practicality of the benchmarks, this work analyzes 267 SmartThings services from official Github and community website of SmartThings [17], [18] and implements the benchmarks with the ComFlex language.

Pet care is a pet caring service that automatically takes care of companion animal when a user is not at home. If the pet barks, then the notifier in pet care device sends a message to the user (Event Bark). Moreover, the pet care device automatically feeds the animal with stored feed. If there is not enough feed or potty pads, then the pet care device automatically requests purchase of new feeds or new potty pads to the user (Event Pay). Lastly, if the users want to see the video of pet using their smartphone or laptop, the pet care device sends video to screen of smartphone or laptop (Event Video).

Hvac is a simple service that automatically controls temperature and humidity of the house. Hvac service senses a change of temperature or humidity of the house using multiple temperature and humidity sensors. If the average temperature or humidity passes predefined bounds, then hvac service turns on air conditioner or humidifier to regulate the temperature or humidity (Event temp and humid).

Smart alarm checks the user’s sleep status using sound, motion, and pressure sensors (Event sleep), and wakes up the user at predefined time with speaker or smart bulbs (Event awake).

Sleep care similarly checks the user’s sleep status to smart alarm service. However, sleep care service uses sleep tracker functionality of smart watch instead of pressure and sound sensors (Event sleep). Moreover, sleep care service regulates intensity of illumination, temperature, and humidity of bedroom to provide better conditions for sleep.

Door lock simply notifies when the door bell rings with various notifiers such as smart bulb, smartphone, and speaker (Event noti). If the users want to watch the video of outside the front door, door lock service relays the video from camera outside the front door to laptop or smart TV (Event video).

Game manager provides mirroring screens of mobile game (Event mirror), remote and secure payment in the game with smartphone or laptop (Event pay), and login functionality using web server (Event login).

B. Programmability of ComFlex Language

To evaluate programmability of the ComFlex language, this work implements all 6 IoT services in Section VI-A with the ComFlex language. Moreover, this work implements 14 IoT devices with 52 resources in the ComFlex language. Some resources such as screen types from different vendors have same screen type, so the ComFlex language flexibly abstract with abstract type screen. The resources such as laptop or smart TV contains multiple different resources in composable manner, so the ComFlex language defines the laptop or smart TV as a composition of other resources.

To compare the programmability of the ComFlex language with other approaches, this work also implements the same IoT devices and services with find-grained resource models [1], [8]. Fine-grained resource models do not abstract similar resources flexibly, and define a new resource as a composition of existing resources. The fine-grained resource models also force programmers manually to write ACLs in a variable and function level to avoid overprivilege problems. This work excludes the other approaches in Table I in this evaluation because they adopt coarse-grained resource models, causing overprivilege problems.

Figure 9 shows the lines of codes of each IoT device program using two different approaches such as fine-grained and ComFlex resource model. Device program written with
fine-grained resource model consists of two parts, device logic and registration codes. Device registration codes support registration of IP address, device identifier, resource list, and etc. Device program written in the ComFlex language also contains device logic, however it does not need registration codes because the ComFlex compiler automatically inserts the registration codes. Instead of registration codes, the ComFlex language needs additional annotations in the ComFlex language. On average, the ComFlex language saves 9% of lines of codes.

Figure 10 shows the lines of codes of each IoT service program with the fine-grained and ComFlex resource model. The IoT service program with fine-grained resource model needs three parts of codes, i.e., service logic, resource discovery codes, and access control list. In case of the ComFlex language, programmers need only service logic and additional codes of newly defined virtual resources. However, definition of virtual resources can be shared as shown in Figure 2, so some services such as Hvac and game manager do not need unnecessary codes for redefinition of existing virtual resources. On average, the ComFlex language saves 40.2% of lines of codes compare to fine-grained resource model.

Based on two results, the ComFlex language successfully simplifies programming of IoT device and service programs.

C. Response Time Analysis

To evaluate the performance of the ComFlex runtime, this work measures the response time of each event of the 6 IoT services described in Section VI-A. Compared to fine-grained resource model, this work estimates additional delays from virtual resources and the ComFlex runtime. Figure 11 shows the results of response time of 14 events in 6 IoT services. Maximum delay from virtual resources and the ComFlex runtime is 20 ms, and average delay is 1.8 ms which is negligible in the IoT services.

This work also measures resource discovery time of each IoT service. Figure 12 shows the results of experiment. The ComFlex shows extremely short resource discovery time in all 6 IoT services. On average, the ComFlex runtime only needs 19.1% of delays for resource discovery, because fine-grained resource model has to request multiple times for each fine-grained resource in different IoT devices. However, the ComFlex runtime only needs to find small number of virtual resource instances which are commonly shared by multiple services.

VII. RELATED WORK

Resource management of IoT frameworks: Many existing IoT frameworks [1]–[13] try to manage various IoT resources across multiple devices efficiently, but they cannot solve all the challenges. Homekit [2] gives its own standard abstraction of IoT devices, however they cannot abstract various third-party devices from different vendors. Moreover, device management granularity is too coarse-grained causing overprivilege problems. Some IoT frameworks [3]–[7], [12] propose a resource-level device management granularity to avoid overprivilege problems. However, they causes huge additional programming burden because the management granularity is too fine-grained. In addition, they still cause variable or function-level overprivilege problems due to resource-level access control. Esperanto [10] proposes flexible resource management scheme for device registration and discovery, reducing the programming burden. However, Esperanto still has coarse-grained management granularity which causes overprivilege problem. Then, some industrial frameworks and publications [1], [8], [9], [11], [13], [19] define an IoT device as a composition of multiple different fine-grained resources to reduce the programming burden caused by complex registration codes. However, they do not support flexible resource abstraction.
VIII. Conclusion

This work proposes a compiler-runtime cooperative framework called ComFlex that supports a new composable and flexible resource management scheme. The compiler allows manufacturers to implement fine-grained resources by extending existing resources, to define an IoT device as a composition of resources, and to abstract an IoT device with flexible granularity. Moreover, the compiler allows programmers to define their own resource management granularity while avoiding the overprivilege problem with automatically generated fine-grained access control list (ACL). The ComFlex runtime registers an IoT device with its accessible properties, and an IoT service with its access control list. In addition, the runtime registers, deploys, and executes programmer-defined virtual resource instances to make programmers manage various fine-grained resources easily. With the composable and flexible resource management scheme, ComFlex supports the IoT services with 40.2% less lines of code and 80.9% less discovery time without the overprivilege problem and the response time delay.

Overprivilege problems in IoT frameworks: Through security analysis of the widely-used Samsung SmartThings IoT platform, Fernandes et al. [15] address overprivilege problem caused by its coarse-grained permission model based on functional similarity. This overprivilege problem can appear other IoT frameworks [14]. To overcome overprivilege problems in IoT frameworks, recent publications [11], [22]–[25] propose IoT security solution to assure least-privilege of IoT resources.

FACT [11] proposes functionality granularity of access controls on IoT devices. Users can register their devices with functionality information, and grants the permissions of required functionalities when an application is installed. Unlike FACT, manufacturers do not need to make information about functionalities separately in ComFlex; manufacturers develop their resources with few annotations, and ComFlex compiler automatically generates interfaces of the resources which contain information about the resource and its opened functionalities.

Other proposals [22]–[25] consider end-users who actually use the IoT services by considering user contexts and information-flow controls. ComFlex will extend the language and its runtime to support user-defined applications or access controls as a future work.

In addition to IoT systems, mobile, desktop and web environment also has security problems on permitting usage of resources. Previous works [26]–[31] have proposed fine-grained access controls and permission models on Android and Linux platforms. These policy languages and security modules are flexible to manage fine-grained access control of packages, but a programmer should write access control lists or make security modules, which can be a burden of the programmer. Unlike existing works, ComFlex compiler automatically builds access control lists based on the IoT application written by a programmer. Also, ComFlex runtime provides access control lists to resources, which decentralizes access control mechanism and provides per-resource protection.
ACKNOWLEDGMENT

We thank the anonymous reviewers and editors for their insightful comments and suggestions. We also thank the CoreLab for their support and feedback during this work. This work is supported by NRF2015M3C4A7065646, IITP-2017-0-00195 and IITP-2018-0-01392 through the National Research Foundation of Korea (NRF) and the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. Hanjun Kim is the corresponding author of this paper.

REFERENCES

Bongjun Kim is a Ph.D. student in the Department of Computer Science and Engineering at POSTECH. He received his B.S. in computer science and engineering from POSTECH. His research focuses on compiler optimization and security enhancement for the Internet of things framework. He is currently working on a compiler framework for many-core neural processing units. Contact him at bong90@postech.ac.kr.

Seungbin Song is a Ph.D. student in the Department of Electrical and Electronic Engineering at Yonsei University. He received his B.S. and M.S. in creative IT engineering from POSTECH. His research focuses on compiler optimization for the Internet of things framework and programmable network switches. Contact him at seungbin@yonsei.ac.kr.

Seonyeong Heo is a Ph.D. student in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). Her research interests include real-time system design for deep neural networks and the Internet of Things. Heo received a B.S. in computer science and engineering from POSTECH. Contact her at heosy@postech.ac.kr.

Hanjun Kim is an associate professor in the Department of Electrical and Electronic Engineering at Yonsei University. He received his B.S. in electrical engineering from Seoul National University, and his M.A. and Ph.D. in computer science from Princeton University. From 2013 to 2018, he was an assistant and associate professor at the Departments of Creative IT Engineering and Computer Science and Engineering at POSTECH. He was awarded the Intel Corporation Ph.D. Fellowship and the Siebel Scholarship at 2012. His research interests include computer architecture and compiler optimization for distributed and emerging systems. Contact him at hanjun@yonsei.ac.kr.