RemedIoT: Remedial Actions for Internet-of-Things Conflicts

Renju Liu
rl@cs.ucla.edu
University of California - Los Angeles (UCLA)

Ziqi Wang
wangzq312@ucla.edu
University of California - Los Angeles (UCLA)

Luis Garcia
garcialuis@g.ucla.edu
University of California - Los Angeles (UCLA)

Mani Srivastava
mbs@ucla.edu
University of California - Los Angeles (UCLA)

ABSTRACT

The increasing complexity and ubiquity of using IoT devices exacerbate the existing programming challenges in smart environments such as smart homes, smart buildings, and smart cities. Recent works have focused on detecting conflicts for the safety and utility of IoT applications, but they usually do not emphasize any means for conflict resolution other than just reporting the conflict to the application user and blocking the conflicting behavior. We propose RemedIoT, a remedial action [1] framework for resolving Internet-of-Things conflicts. The RemedIoT framework uses state of the art techniques to detect if a conflict exists in a given set of distributed IoT applications with respect to a set of *policies*, i.e., rules that define the allowable and restricted state-space transitions of devices. For each identified conflict, RemedIoT will suggest a set of remedial actions to the user by leveraging RemedIoT's programming abstractions. These programming abstractions enable different realizations of an IoT module while safely providing the same level of utility, e.g., if an air-conditioner application that is used to implement a *cooling module* conflicts with a CO_2 monitor application that requires ventilation at home, a non-conflicting smart fan application will be suggested to the user. We evaluate RemedIoT on Samsung SmartThings applications and IFTTT applets and show that for 102 detected conflicts across 74 sample applications with 11 policies, RemedIoT is able to remediate ~ 80% of the conflicts found in the environment, which would normally be blocked by prior solutions. We further demonstrate the efficacy and scalability of our approach for smart city environments.

CCS CONCEPTS

• **Software and its engineering** → **Software safety**; General programming languages.

KEYWORDS

Smart Environment, Environment safety, IoT conflicts

[1]This paper draws on the analogy of remedial action schemes for safety-critical, complex industrial control systems such as the electric power grid.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
BuildSys '19, November 13–14, 2019, New York, NY, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7005-9/19/11…$15.00
https://doi.org/10.1145/3360322.3360837

ACM Reference Format:
Renju Liu, Ziqi Wang, Luis Garcia, and Mani Srivastava. 2019. RemedIoT: Remedial Actions for Internet-of-Things Conflicts. In *The 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '19), November 13–14, 2019, New York, NY, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3360322.3360837

1 INTRODUCTION

The proliferation of the Internet-of-Things (IoT) has enabled sensors and actuators across several facets of society for the purpose of automating and optimizing our daily tasks. As humans settle into smart environments, the collection of data and automation of associated processes draw concerns of safety, security, and privacy [2, 15, 27]. IoT programming platforms such as IFTTT [28], Zapier [31], and Samsung SmartThings [25] have eased the configuration process of smart environment devices, but they placed the burden of addressing those aforementioned concerns on both the IoT application developers and the application users who are configuring the environment.

Previous works have attempted to alleviate this burden by checking potential IoT configurations against *policies*, which are rule specifications of a particular system with respect to some objective function such as safety or utility. An example policy may aim at preventing racing conditions between two applications that may turn a device on or off. Policy-based mechanisms have been proposed to enforce policies that avoid unsafe or unsecure states for smart environments. Solutions such as IoTGuard [6] will raise an alert to the user if a particular policy is violated and block the associated action. The general focus of these mechanisms has been to determine inter or intra-app safety, security, and privacy conflicts for either explicitly defined relationships amongst devices [8, 11, 14] or for implicit relationships that are learned based on the device behavior [3, 12, 20].

Although these solutions provide promising approaches to conflict detection, they typically do not have a means of taking the right course of action once a conflict is detected–they usually only block the conflicting actions [4]. Several of these conflicts may be in the context of a safety-critical application and require immediate remediation, i.e., an automated solution to resolve the conflict without deadlocking any critical applications. For example, an IoT application may specify that if the concentration of CO_2 is high, an immediate ventilation action needs to be performed by opening the windows. Meanwhile, another thermostat application ensures that the windows remain closed in cold weather to save energy. If the indoor CO_2 accumulates to a dangerous level in winter, a conflict may arise. The current state of the art conflict protection

mechanisms, however, may block the safety-critical ventilation actuation. In this case, a *remedial action* would resolve the conflict by finding an alternative or redundant path that implements the same utility, e.g., an HVAC system would be used instead of the smart window for ventilation. However, finding alternative or redundant realizations for commodity IoT utilities can be difficult. Implemented as traditional device drivers, IoT device interfaces are typically *monomorphic* and, as such, cannot be easily realized by alternative means. We, therefore, need a method to provide a *polymorphic* interaction interface such as polymorphic programming abstractions for those devices that facilitate such remedial actions.

Polymorphic abstractions for sensors and actuators have been employed by IoT macroprogramming frameworks that suggest high-level commands for distributed IoT frameworks [23]. These frameworks rely on the notion that they can specify *what* should be sensed or actuated as opposed to *how* sensing or actuation is carried out. Preliminary works show the possibility of raising abstractions via *inference graphs* for sensors[26], e.g., a "fitness activity" sensor abstraction may be realized by either an inertial movement sensor or a heart rate monitor. However, the same notion has yet to be realized for actuation as it is difficult to reason about the utility of an actuator based on the user's intention.

In this paper, we present RemedIoT, a remedial action engine framework for resolving the conflicts (i.e., policy violations) of IoT-based smart environments. RemedIoT helps the application users remediate conflicts when they configure the smart environment through smart application platforms. We introduce a key component, *actuation graphs*, that allows raised, polymorphic abstractions of both IoT actuators and sensors. RemedIoT leverages the IoT actuation graphs to provide alternative realizations of sensors and actuators for smart environments. Safety, security, and privacy policies can then be specified with respect to these raised high-level sensor and actuator abstractions. RemedIoT then uses a conflict detector to determine if a given IoT actuation or control command triggers any conflicts. If a conflict is detected, the Remedial Action Engine suggests alternative realizations of a user's intended application that can be installed to avoid such a conflict. Once a remedial action has been selected, the same conflict runtime executes the remedial action scheme.

We evaluate RemedIoT on Samsung SmartThings applications and IFTTT applets by extracting 195 possible automation logics and generating 74 sample applications with 11 policies. For these applications, we utilized state of the art conflict detector approaches to detect 102 possible conflicts. We show that RemedIoT is able to remediate $\sim$ 80% of the conflicts that would normally be blocked by prior solutions. We show how remedial actions can be optimized against abstraction cost functions such as power consumption and device profiling. We further discuss the efficacy and scalability of RemedIoT in the context of smart city environments.

Contributions. Our contributions are summarized as follows.

- We provide *actuation graph* abstractions for actuators in IoT smart environments.
- We present RemedIoT, a remedial action engine framework that utilizes alternative realizations of IoT applications to provide remedial actions for a given IoT policy conflict.
- We evaluate RemedIoT on a set of Samsung SmartThings applications and IFTTT applets and show how RemedIoT can remediate $\sim$ 80% of conflicts that would normally be blocked by previous solutions.
- We show how remediation can be optimized for providing automated remedial actions based on abstraction metadata.

Our source code and datasets are available online:
`https://github.com/nesl/buildsys-19-code`

2 BACKGROUND

In this section, we provide the preliminary information necessary to understand the rest of this paper. We first discuss the state-of-the-art for IoT event service platforms. We then discuss the state-of-the-art for conflict detection across different IoT services along with their limitations.

2.1 IoT Event Service Platforms

The current programming paradigms for commodity IoT smart environments typically facilitate IoT device interaction through a control hub, e.g., Samsung SmartThings [25], Apple HomeKit [1], and the Microsoft Azure IoT Edge [19].

Recently, programming paradigms have emerged that allow users to interact with IoT devices through *event services*. Instead of having the user manually control IoT devices through a control hub, these services abstract away the complexities of automation. These mechanisms automate user control processes through *events*,i.e., user-defined trigger-action schemes for IoT environments. For instance, event service platforms such as IFTTT (If-this-then-that) [28] and Zapier [31] provide a more convenient way to execute an action when a user-defined condition or set of conditions is satisfied. The user only needs to set an "if" condition (i.e., event trigger condition), and the corresponding IFTTT rule will interact with the control hub to execute the associated event action. Although these paradigms generally provide a convenient means of programming IoT domains, we will show that their limitations reside in their sensor and actuator programming abstractions.

Generalized system model. The system model we consider in this paper has a control hub that may be running one of the hub services mentioned above. A user may have direct access to a device via an application interface or can write an *event* for the smart environment using a commodity IoT event service platform. Each event consists of a condition or a set of conditions and an associated action(s). The condition is an expression that specifies the state of the smart environment, and the action is the actuation command that changes the environment state(s). The device state-space representation for both conditions and actions is domain-specific and depends on the API provided by the IoT event service platform, e.g., a smart bulb being on or off. A *policy* will also be domain-specific as it defines the sets of allowed and disallowed state-space transitions for an IoT system [6]. Our model allows a policy to be specified by whoever is configuring the IoT system. In this paper, we seek to provide conflict remediation, i.e., policy violation resolution, for the system model and a number of defined events. The remediation, in essence, is a set of new events suggested for replacing some old ones in order to resolve conflicts in the event set. We first enumerate the

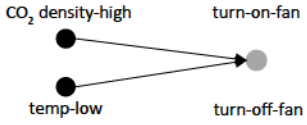
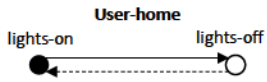
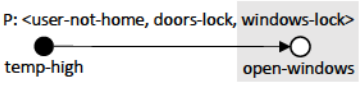
Types of Policies		
P1: Mutually exclusive states must not exist in the environment.		**P2: User-defined rules.**
Racing Events	Cyclic Events	**E.g.: Doors and windows must be locked is user is not home**
E1: CO_2 density-high -> turn-on-fan E2: temp-low -> turn-off-fan	E1: user-home, lights-on -> lights-off E2: user-home, lights-off -> lights-on	E1: user-away -> user-away-mode-on E2: user-away-mode-on, temp-high -> windows-on
		

Table 1: The two general categories of policies: the policies defined by Celik *et al.* [6] and user-defined policies.

limitations of current conflict detection methods and formalize the conflicts considered in this paper.

Conflict types. We define a *conflict* as a violation of policy. We consider two general categories of policies: user-defined policies and the class of policies introduced by Celik *et al.* that do not allow mutually exclusive states to exist in an environment [6]. The latter category can be decomposed into *racing events* and *cyclic events*. *Racing events* stand for two or more events that are triggered at the same time while having conflicting actions, and *cyclic events* represent two or more events consist of a set of conditions and a set of actions that mutually trigger each other continuously. Table 1 summarizes the types of policy violations we consider with illustrative examples.

As was previously pointed out, any forms of conflict resolution from prior works have been limited to simply blocking a prospective action or reporting the conflict to the user[4]. The bottleneck for providing remedial actions for such conflicts resides in the current state-of-the-art for obtaining the user's intention since the current IoT actuation programming abstractions hide such information. These premises allow us to provide an overview of the RemedIoT framework.

2.2 Related Work

We now discuss the relevant works directly related to RemedIoT.

Conflict Detection. Several works have already focused on the problem of detecting conflicts between IoT events. BuildingRules [22] proposes an occupant customized building configuration system. Surbatovich *et al.* [29] builds an information-flow model to analyze how IFTTT recipes violate the integrity, and it then categorizes what damages the IFTTT recipes could cause for the user. IoTSAT [21] models the cyber-physical behaviors of IoT devices based on the factors including the network, device configurations, and user policies to analyze the possible or potential vulnerability of the IoT network. Danger-system [24] detects the smart building environment conflicts through mobile crowd sensing. Ma *et al.* [17] and CityGuard [16] provide runtime detection for the specific conflicts in the smart city through an intermediate layer of watchdog. IoTMon[8] uses data mining and NLP-based technology to analyze how the applications affect physical environments, which further suggests the risks of the applications. IoTGuard [6] enforces a set of policy rules on IoT applications by injecting monitor code into the target app, and it blocks any unsafe states. Soteria [5] builds a static analysis system from the application descriptions to infer whether there are potential safety or security issues of the app. SIFT (Liang *et al.*) [14] proposes a safety-centric programming platform for connected devices in IoT environments to safely verify that there are no conflicts when the developers compile the applications. Miettinen *et al.* [20] introduced a technique for IoT *device-type* identification used for security enforcement based on device-fingerprinting.

All of the above works have focused on analyzing or detecting conflicts or security violations. However, none of them has yet provided a way to resolve such conflicts or violations except for simply reporting them to the user. Nextly, we introduce some other works showing how conflicts can be *debugged*.

Conflict Debugging. CityResolver [18] uses Integer Linear Programming (ILP) method to find the most optimal conflict resolution in a given set of resolutions. Meanwhile, RemedIoT provides a means of automatically generating the resolutions. Liang *et al.* [13] provides an automated debugging tool for the IFTTT platform by adjusting the clauses or the parameters of IFTTT rules. However, such modifications might not lead to the original desired state that the user wants. RemedIoT focuses on providing an alternative path to get to the exact original desired states without violating any policies or conflicting with other events.

IoT programming abstractions Beam [26] abstracts sensors into modules based on the inference that can be made from the sensory data. Such abstraction is realized using inference graphs, and then the developers only need to focus on sensing capabilities rather than choosing particular sensors. However, Beam does not deal with the abstraction of actuators that might cause conflicts among the devices. ExPat [30] proposes a formal specification programming language to ensure that the rules in the smart environment can be varified by the user's intention without conflicting with other rules. However, it does not provide any solutions to the conflicts. HomeOS [9, 10] and BOSS [7] provide an operating system for smart home environments so that programming or accessing the IoT devices at home can be done through an OS-like interface, i.e., device drivers. However, like all other traditional operating system, HomeOS simply puts a lock on the device when other processes are trying to access it. It does not make attempts to find conflicts or resolve them.

3 REMEDIOT OVERVIEW

RemedIoT is a dynamic and context-based mediator for IoT event services. RemedIoT aims to provide a set of meaningful remedial actions for conflicting actions rather than merely block the actions. For example, an application user may configure an event that opens the window when the temperature is too high, but this configuration may violate a safety policy which states that all windows

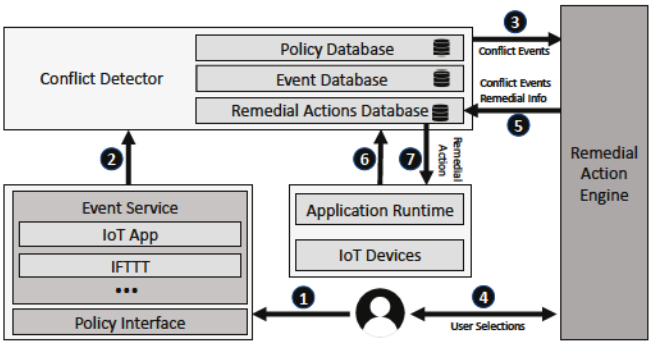


Figure 1: An overview of the REMEDIOT framework.

and doors should be closed when the user is not home. REMEDIOT would provide a remedial action to suggest the user turn on the HVAC instead of opening the window as the former provides the same utility. Figure 1 shows the design overview and the workflow of how REMEDIOT interacts with the aforementioned system model.

At configuration time, a user may attempt to install a third-party IoT safety policy through a policy interface or configure an event service (1) that can be analyzed by the conflict detector (2). The conflict detector will check if the prospective event conflicts with any installed events or policies. If conflicts do not exist, the new event will be installed. If a conflict exists, it will send the set of conflicting events to the REMEDIOT Remedial Action Engine (3) that will analyze the conflict and see if any remedial actions can be installed to prevent conflicting actions from being triggered at the same time. The remedial actions will then be suggested to the user (4). After the remedial actions are selected by the user, they will then be written back to a database for future use (5). When such a conflict happens during runtime, the remedial action from the database will be executed. At runtime, all events from the installed event services are examined in case an event is conflicting with a policy(6). If a conflict exists, the appropriate remedial action is carried out if available. Otherwise, the action is blocked (7).

The core contribution of this paper resides in the Remedial Action Engine. However, in order to realize such a tool, we need to design appropriate programming abstractions first.

4 REMEDIOT PROGRAMMING ABSTRACTIONS

The main goals of REMEDIOT's programming abstraction design are to not only increase the usability of distributed applications for dynamic smart environments, but to also ease the development effort for application developers with respect to safety and security. The abstractions enable the application developers to specify high-level programming intentions of an application instead of focusing on which devices will realize the application utility. For example, if a developer wants to develop an application that lowers the temperature of a space, he or she only needs to specify "cooling down" rather than selecting an HVAC or a smart fan realization. REMEDIOT maintains the dynamic realization of these high-level intentions through lower-level device abstractions–where the lowest level abstractions will be the monomorphic device drivers. Such a design provides an inherent redundancy for abstractions that provide the core of REMEDIOT's remedial action engine. Previously, Beam [26] presented a framework to provide such a hierarchical structure for sensors called an *inference graph*. REMEDIOT couples this notion of an inference graph with an *actuation graph* to abstract the actuators of the system.

4.1 Actuation Graph

REMEDIOT's actuation graph–provided by the programming environment developers –maintains how an actuation should be realized. The graph is built as a directed graph where each node stands for an *actuation module* that indicates *how* to perform actuation or sensing tasks by combining other submodules or devices either collaboratively or independently. Each actuation module can have different *implementation units*–which are hidden from the application developers–to realize the module. The direction of the actuation graph corresponds to the dependency relationship among the actuation modules. With the assistance of the actuation graph, REMEDIOT can easily infer the intentions of an application user by looking at which module is used. The actuation graph subsumes the notion of Beam's inference graph [26] because it encapsulates the sensing dependencies of actuators. Moreover, the actuation graph is highly customized and developed differently for each smart environment.

Although abstraction modules enable REMEDIOT to reason about a user's high-level intentions as well as to provide alternative actuation realizations, REMEDIOT conversely needs a mechanism to expose the capabilities of physical devices. It is necessary to build device-level abstractions that are then used to realize the higher-level modules. As such, we introduce a *device abstractor module* that allows developers to interface devices with abstraction modules. For clarity, we will present the aforementioned actuation graph components in an illustrative example.

Motivating example: home control application. Figure 2 shows an actuation graph used by a home control application. The home control application has two installed events:

(1) *If dangerous incidents are detected, then send notification to the user.*
(2) *If the temperature of a room is higher than 80 degrees, then cool it down.*

The first rule will require the actuation graph to provide a detection module for "dangerous incidents" and a "notification" module such that the user can be appropriately notified. The second rule needs to have a "temperature" module to sense the ambient temperature as well as a "cool down" module that lowers the ambient temperature. In the associated actuation graph, the sensing module *temperature* requires the data from a smart thermal sensor installed in the room to infer the temperature. Those data are hidden from the application developer's perspective. The actuation module *cooling down* needs to access either a smart window or the air conditioner–where the application developers need not worry about which device to use. However, in order to be compatible with existing IoT systems such as SamSung SmartThings and Apple HomeKit, our programming model also allows event services to directly access the devices without going through high-level sensing or actuation modules. In any case, the abstraction modules are designed to account for all possible conflicts that may exist in the network.

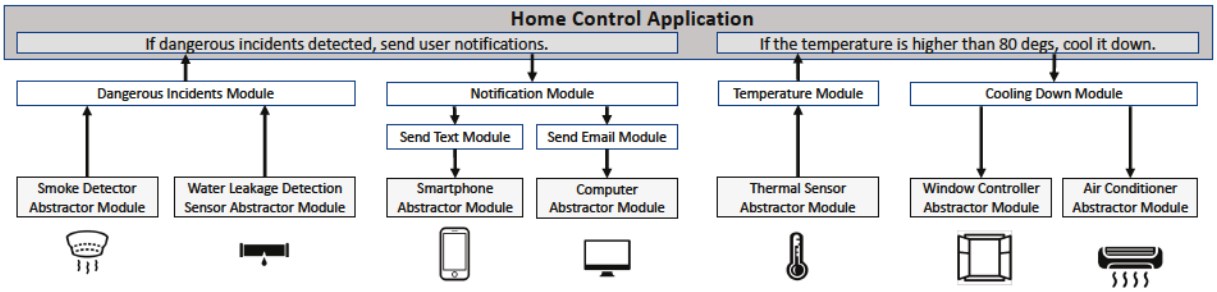


Figure 2: A sample smart home control application and its associated actuation graph.

We now describe the attributes of an abstraction module required to maintain the actuation graph.

4.1.1 Abstraction Module. The abstraction module is a programming interface of a sensing or actuation functionality performed by specific implementations. An abstraction module may also be realized by other abstraction modules. Each module is implemented through *implementation units*, where the module provides a template of the attributes to be defined. The implementation units encapsulate the actuation or sensing algorithms of the associated modules. For instance, the aforementioned *cool down* abstraction module is generally associated with lowering the ambient temperature of a space, while the actuation graph developers may be able to develop different cooling down *implementation units* for a specific smart home environment, such as cooling down through an air conditioner, or cooling down by opening the windows or doors. The following attributes are necessary to realize an abstraction module for our actuation graph:

- **Module name.** The module name is the label for the abstraction modules, which summarize the characteristics of that module, e.g., "cooling down".
- **Implementation unit name.** The implementation unit name summarizes the function of an actual implementation unit that may realize the associated abstraction module. For instance, "turn on AC" or "open the window". Each abstraction module may correspond to multiple implementation units.
- **Module type.** The module type indicates whether the abstraction module is a sensing module, an actuation module, or both.
- **Implementation prerequisites.** The prerequisites refer to the implementation unit's dependent modules. Each implementation unit needs to specify what it may depend on in order to be realized. For instance, one of the implementation units of the *notification* module might depend on a *send text* module, and another implementation unit of the *notification* module might depend on a *send email* module. Different implementation units can have different prerequisites under the same module.
- **Cost and utility metrics.** Each abstraction module may be associated with a cost or utility metric that will establish how implementation units should be prioritized when RemedIoT is suggesting remedial actions. For instance, the utility metric of a "cooling down" abstraction can be defined as to how efficiently an implementation unit may lower the ambient temperature, while a cost metric may be associated with the power consumption of a unit.

Defining each of the attributes mentioned above will be domain specific, and it is up to the programming environment developer to ensure that all of the components of the actuation graph are compatible with each other. The first step in doing so is to implement a *device abstractor module*.

4.1.2 Device Abstractor Module. The device abstractor is a type of special abstraction module that handles the details of a particular device's metadata so that the high-level information is exposed to the developer. In particular, the abstractor module characterizes and implements the connection to the device through the device driver and the network. Every physical device corresponds to an implementation unit, but each device abstractor module might have one or more device implementation units. For example, if there are three installed cameras, each of them will have one corresponding implementation unit, and only one device abstractor module "RGB camera" is needed to encapsulate all the three implementation units. Similar to HomeOS [9], RemedIoT dynamically handles the connections and disconnections of devices in the network. When a device is disconnected from RemedIoT, RemedIoT updates the actuation graph accordingly. Moreover, if a new type of device is connected, RemedIoT creates a new device abstractor instance and updates the graph if other abstraction modules depend on it.

To provide context of these abstractions, we describe how a an example abstraction module may be implemented.

4.2 Programming Example

Listing 1: Sample Inference Module

```
# Import the existing modules
from ActuationGraph import *

# Implementation of cooling down module via AC
class ACTurnOn(ImplementationUnit):
    def __init__(self):
        ImplementationUnit.__init__(...)
        acDA = ACDeviceAbstractor() #device abstractor
        super(ACTurnOn, self).appendChildDeviceAbstractor(acDA)

    def performFunc(self, *args):
        # Perform the specific action to turn on AC
        ...

# Cool Down Module
class CoolDownModule(Module):
    def __init__(self):
        Module.__init__(...)
        acTurnOn = ACTurnOn() # Implementation Unit
        fanTurOn = FanTurnOn() # Implementation Unit
        super(CoolDownModule, self).addImplUnit(acTurnOn)
        super(CoolDownModule, self).addImplUnit(fanTurOn)
```

To develop an implementation unit of a module in the actuation graph, the actuation graph developers need to follow the template of the module and implementation units. Moreover, each actuation module can only allow prerequisites on either device abstractor modules or other general actuation modules for its implementation units. The device abstractor modules and the general actuation modules cannot simultaneously be the prerequisites of one actuation module. For instance, in Figure 2, the send text module and *computer abstractor module* cannot be the prerequisites of *notification module* at the same time. This is to ensure that all prerequisites of one module have the same level of hierarchy.

Listing 1 shows an example implementation of how a `CoolDownModule` class may be implemented as a base abstraction module associated with device modules that can cool down the ambient temperature. We show an example implementation unit of this module called `ACTurnOn` that may define the behavior associated with turning on an air conditioning unit.

Given the aforementioned programming abstractions, we now show how prior notions of a policy file can be augmented to incorporate abstraction modules.

4.3 Policy Grammar Definition

We show how policy grammars for conflict detectors in previous works can be augmented to incorporate our abstraction modules. For instance, IoTGuard [6] defined a Backus-Naur Form (BNF) policy grammar to allow developers to define rules regarding the sets of allowed and restricted state transitions for IoT devices. The same policy grammar can be augmented to define the allowed and restricted state transitions for abstraction modules, as shown in Figure 3. The semantics would remain the same as long as abstraction module attributes are syntactically correct.

These programming abstractions, along with a conflict detector that utilizes an associated augmented policy grammar, allow us to realize the RemedIoT remedial action engine.

5 REMEDIAL ACTION ENGINE

As discussed in Section 3, RemedIoT's runtime remedial action engine takes as its input a conflict generated by a conflict detector and attempts to resolve the the conflict by utilizing a configured remedial action for the particular conflict. If no remedial action exists, the conflicting events will be blocked. To provide the design details of this system, we first discuss how these remedial actions are configured and installed prior to runtime.

5.1 Remedial Action Configuration

Remedial actions are configured upon encountering a new conflict from the conflict detector. New conflicts may arise during the installation of either event services or policies. Algorithm 1 describes

⟨*policy-set*⟩ ::= ⟨*statement*⟩; (⟨*statement*⟩;)* **EOF**
⟨*statement*⟩ ::= ⟨*restrict-clause*⟩ | ⟨*allow-clause*⟩
⟨*restrict-clause*⟩ ::= **RESTRICT** : ⟨*state-transition*⟩
⟨*allow-clause*⟩ ::= **ALLOW**: ⟨*state-transition*⟩
⟨*state-transition*⟩ ::= ⟨*transition*⟩ (, ⟨*transition*⟩)* : ⟨*state*⟩ (,⟨*state*⟩)*
⟨*state*⟩ ::= ⟨*abstraction-module*⟩ . ⟨*abstraction-attr-id*⟩ (.⟨*abstraction-attr-id*⟩)* <*logical-operator*⟩ ⟨*attr-value*⟩

Figure 3: Policy grammar for IoTGuard [6] augmented with RemedIoT's programming abstractions.

the remedial action configuration process for a candidate event e. The first step is to detect whether the candidate event conflicts with any existing event services or policies.

Conflict detection. RemedIoT ensures that any candidate events or policies are first passed through the conflict detector (Line 4) to see if any conflict exists with other installed events or policies. Because the state space for event services considered by RemedIoT is limited to two states (*on* or *off*), the set of conflicts that arise, E_{conf}, can then be divided into two sets: the singleton set with the candidate event and the set of all events that have a conflicting action with e. Further, we only care about remediating the conflicts for the set of events with lower priority, where "priority" refers to the aforementioned cost and utility metrics of an abstraction module. The higher priority set should always take precedence for executing a particular action at runtime. Therefore, RemedIoT will first generate the full set of conflicting events E_{conf} (`Line 4`), sort the events based on priority (`Line 5`), and then extract the lower priority set of conflicting events E'_{conf} (`Line7`). Given this extracted set, RemedIoT will attempt to find a remedial action for each conflict.

Remedial action generation. RemedIoT first iterates through each conflicting event service, e_{conf}, and prompts the user (`Line 10`) for the intended abstraction module of the conflicting implementation unit by utilizing the actuation graph, G_{act}. For instance, in Figure 2, if the conflicting implementation unit was the "Window Controller", the user may confirm that the intended abstraction module was the parent "Cooling Down" module. If the user declines to use the parent abstraction module, then RemedIoT will block this action upon detecting a conflict at runtime. RemedIoT ensures that any parent abstraction module that is presented to the

Algorithm 1 Remedial Action Configuration

1: **Input:** candidate event e, actuation graph G_{act}, policy database DB_{policy}, event database DB_{event}, remedial action database DB_{rem};
2: **Output:** updated event database DB'_{event}, updated remedial action database $DB'_{remedial}$;
3: //Get conflicting events for given event.
4: $E_{conf} \leftarrow$ conflictDetector(e, DB_{event}, DB_{policy}, DB_{rem});
5: Sort conflict events E_{conf} based on priority;
6: // Extract lower priority set of conflicts to remediate:
7: $E'_{conf} \leftarrow$getLowerPrioritySet(E_{conf})
8: **for all** $e_{conf} \in E'_{conf}$ **do**
9: | // Get user to select the intended non-conflicting module:
10: | $intendedModule \leftarrow$ getIntentionFromUser(e_{conf},G_{act});
11: | // Given intended module with pruned impl. units, prompt
12: | // user to select a non-conflicting rem. action impl. unit:
13: | $a_{rem} \leftarrow$getRemActionFromUser($intendedModule$, G_{act});
14: | // Map each conflict pair to its remedial action in database:
15: | **for all** $e^{\complement} \in E_{conf} \backslash E'_{conf}$ **do**
16: | | $DB_{rem} \rightarrow$add($\langle (e_{conf}, e^{\complement}), a_{rem} \rangle$);
17: //Add candidate event to event database:
18: $DB_{event} \rightarrow$add(e);

user will have a set of non-conflicting implementation units, i.e., any implementation units from the actuation graph G_{act} that will cause a new conflict are pruned. Given the intended abstraction module, RemedIoT then prompts the user to select an alternative implementation unit (Line 13), e.g., the aforementioned example will prompt the user to select the "Air Conditioner" implementation unit. If the user selects a suggested remedial action, the action is stored in a remedial action database $DB_{remedial}$ by mapping each conflicting event pair, i.e., e_{conf} and the complementary e^C for all events in the higher priority set $E_{conf} \backslash E'_{conf}$, to its associated remedial action a_{rem} (Line 15). If no remedial action is chosen, then the default action will be to block the conflicting event. Once all remedial actions have been generated, RemedIoT can now add the candidate event e to the event database DB_{event}.

5.2 Runtime Remediation

At runtime, RemedIoT runs alongside the event service platform. IoT event service platforms such as IFTTT typically check the trigger conditions of all installed event services periodically. When a new event is activated, RemedIoT will utilize the conflict detector mentioned in Algorithm 1 to check if the new event has conflicts with other events that are currently active. If a conflict exists, it will query the remedial action database DB_{rem} to select the appropriate action.

Each of the aforementioned components will have engineering challenges which we address in the subsequent section.

6 IMPLEMENTATION

In order to implement RemedIoT's core utilities, we first demonstrate how an existing conflict detector can be instrumented to interface with the RemedIoT remedial action engine. We first describe the conflict detector instrumentation and then discuss the implementation for both the abstraction graph module support as well as the RemedIoT action engine.

6.1 Conflict Detector Instrumentation

We implement the framework for the conflict detector as a directed graph–as was done in the conflict detector for IoTGuard [6][2]. Within the graph, an event can be described as two nodes and one directed edge. Each node stands for a set of device (or module) states and each directed edge indicates a trigger-action relationship. Note that one node may contain a cascade of conditions (actions) instead of only one module state. A conflict arises when a collection of compatible conditions, i.e., conditions that can be satisfied at the same time, can simultaneously cause a set of incompatible actions, i.e., lead to mutually exclusive states of the same device.

When a new event is added, the conflict detector[3] traverses the graph nodes to determine all the *compatible conditions* nodes as well as all the *incompatible actions* nodes. We then perform a reachability analysis to determine if a path exists between these two sets. Any detected path will be considered as a potential conflict and removed from the graph. If any conflicts arise, these conflicts, along with the new event, are delivered to the remedial action engine to generate the associated remedial actions. If no direct conflicts are observed in this stage, the new event will be added to the graph. Finally, we employ a depth-first search-based algorithm to find all the loops inside the dependency graph. All discovered conflicts and loops are forwarded to the Remedial Action Engine.

[2]We implemented our own conflict detector inspired by IoTGuard since the source code was not available at the time this paper was written.

[3]The conflict detector module was implemented in Python with 400 LoC and is available in the repository.

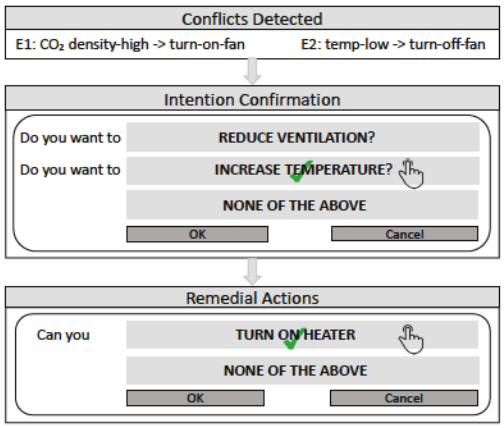


Figure 4: An example interface for RemedIoT that suggests a remedial action to the user when a conflict is detected.

Given a conflict detector, we now describe how we implement the core components of RemedIoT. Before we can describe the implementation of the remedial action engine, we first describe the implementation of the actuation graph module support that enables the remediation.

6.2 Actuation Graph Module Support

We implement a generic API for IoT programming environments to support the aforementioned actuation graphs. The graph support is implemented as a base abstract class in Python with a domain-specific implementation. We illustrate how an actuation graph can be constructed in Section 7. The actuation graph support enables the remediation of conflicts.

6.3 Remedial Action Engine

The Remedial Action Engine is also implemented in Python based on Algorithm 1. We implement a proof-of-concept interface to allow the user to select the proper remedial action, as illustrated in figure 4. The remedial action, event, and policy databases are implemented and maintained as dictionaries in Python. The Remedial Action Engine also utilizes the aforementioned conflict detector both at configuration time and at runtime. We now detail our evaluation with illustrative examples.

7 EVALUATION

To illustrate a domain-specific implementation, we evaluate RemedIoT on Samsung SmartThings and IFTTT applets. We first describe how we manually extract events and policies for the platforms in order to perform an evaluation.

7.1 Event and Policy Realization

We manually extract a large set of events from both programming platforms. Although we do not obtain all possible events

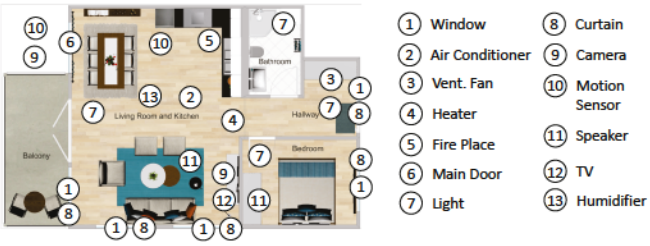


Figure 5: The simulated smart home environment with artificially placed IoT devices utilized in our evaluation.

from both platforms, we try to choose the most representative actions from each application[4]. When we extract the events from the platforms, we format the input events as "IF conditions THEN actions[,priority]", where the priority can be one of three categories (highest to the lowest): safety, energy, and utility. By default, each event is designated as a utility. The priority is used to determine which event is more important when a conflict is detected. For example, in a Samsung SmartThings safety application, we manually extract an event as "`if co2.density >= 50 then vent_fan.state = on, safety`". The extracted event is then parsed and passed to the conflict detector as an input.

Further, all policies are implemented as special events to our system. For our evaluation, each specific policy has been assigned to a category – *house safety*, *energy economy*, or *user comfortability*. For example, P.1 in table 3 is implemented as "if user.home = 0 then doors.state = 1 and windows.state = 1,safety". We now discuss the environmental setup for our experiments.

7.2 Environment Setup

We first describe how we set up our experimental environment. We simulate a smart home environment, as shown in figure 5. It has 13 different smart devices. We run our experiment on an Intel Nuc desktop equipped with an Intel Core i7-6770HQ. For this smart environment, we have to construct an actuation graph for the associated IoT modules.

Actuation graph modules. We construct an actuation graph that implements 13 different devices through 8 actuation modules that cover all situations in our selected events. Moreover, each module provides 2 to 3 different paths (implementation units) to achieve the desired state specified by the module. Table 2 summarizes the modules and the implementation units we provide.

The cost of each module or implementation is simply the energy it might consume using a hard-coded value. For example, turning on the AC has a higher cost than opening the window[5]. Given this environmental setup, we now describe the set of microbenchmarks used to evaluate RemedIoT.

7.3 Microbenchmarks

We propose a set of benchmarks extracted for real Samsung SmartThings applications and IFTTT applets. The goals of our benchmarks are to determine how many conflicts–which would normally be blocked by prior works–can be remediated. We achieve this by essentially trying to configure as many events as possible in our simulated smart environment. We further want to show the configuration time overhead to illustrate the efficacy and usability of RemedIoT. In total, we collect 195 representative events from 74 applications. Moreover, we propose *ten* specific policies ranging from housing safety to user comfort and one general policy, as illustrated in table 3. We implement *eight* different modules with 20 implementation units, as shown in table 2, to support the Remedial Action Engine.

[4]All the events we select are available in the github repo.

[5]Although power consumption profiles can be made for devices, we simply set relative values to ensure the devices are sorted correctly according to their expected power consumption for a proof-of-concept.

Module	Description	Implementation Unit	Description
Cooling down	Cool down the house	Window Opening Low	Open the window when outside temp. is lower
		AC Turning On	Turn on Air Conditioner to cool down
		Fan Turning On	Turn on cooling fans
Heating up	Heat up the house	Window Opening High	Open the window when outside temp. is higher
		Heater Turning On	Turn on the heater to heat up
		Fireplace On	Light up the fireplace to heat up the house
Ventilation	Ventilate the air of the house	Fan Turning On	Turn on the fan to ventilate
		Window Opening	Open the windows to ventilate
		Door Opening	Open the doors to ventilate
Illumination	Brighten up the house	Light Bulbs On	Turning on the light bulbs
		Curtain Opening	Open the curtain when outside is brighter
Green Energy	Save the energy of the house	High-Power Devices Off	Turn off the high power consumption devices
		Low Power Mode On	Set low power mode for the devices
Movement	Detect movement at home	Camera Detection	Detect the motion through cameras
		Motion Sensors	Detect the motion through motion sensors
Warning Notification	Send the warning info to the user	Text Warning	Text the warning message to the user
		Speaker Warning	Play warning sounds through speaker
		Flashlight Warning	Display flashlight using the lightbulbs
Increasing Humidity	Increase the humidity	Fan Turning Off	Turn off the ventilation fan
		Humidifier On	Turn on the humidifier

Table 2: The actuation modules and their associated implementation units we considered in our evaluation.

Category	Index	Policy Description
General	P.0	Mutually exclusive states must not exist in the environment.
House Safety	P.1	The doors and the windows must be locked when the user is not home.
	P.2	The emergency alarming system must be on.
	P.3	The security cameras must be on when the users are not home.
	P.4	The electronic devices must be off when fire sprinklers are on.
Economic Energy	P.5	The heater and the AC must not be on at the same time.
	P.6	The sprinklers must be off when it rains.
	P.7	The dryer and the humidifier must not be on at the same time.
User Comfortability	P.8	The non-emergency sound system must be off when users are sleeping.
	P.9	The curtains must be closed when private mode is on.
	P.10	The lights must be off while the users are sleeping.

Table 3: The policies used to evaluate RemedIoT on Samsung SmartThings and IFTTT applets.

# Apps	# Events	# Conflicts	% Conflicts (Out of Events)	% Blocked Events (Out of Conflicts)	% Remedial Actions (Out of Conflicts)
74	195	102	52.31%	19.61%	80.39%

Table 4: The aggregated results for conflict detection and remediation using RemedIoT in the context of the Samsung SmartThings and IFTTT applets.

7.4 Results

The aggregated results for installing the set of events are shown in Table 4. Based on 195 extracted events from 74 sample applications and 11 policies, we detect 102 conflicts, i.e., 52.31% of all events conflicted with each other upon configuration. By using the RemedIoT framework, 80.39% of the conflicts can be sought for remedial actions. These remedial actions highlight the contributions of our paper: in previous conflict detecting systems, theses conflict events are simply blocked. RemedIoT provides a set of substitution events that can realize the same function for 80.39% of the conflicts, i.e., 82 of the 102 events could be configured without worrying about

Index	Event	Conflict	Remedial Action
C.1	home temp >= 60 & outside temp < 60 ⇒ *open window*	window is on and off at the same time	home temp >= 60 & outside temp < 60 ⇒ vent. fan on
	user away ⇒ user away mode on		Not Applicable
	user away mode on ⇒ *close window*		Not Applicable
C.2	vacation mode on ⇒ *all lights off*	Lights on and off at the same time	Not Applicable
	motion detected ⇒ *lights on*		motion detected ⇒ text the user
C.3	home mode on ⇒ *turn on TV*	TV on and off at the same time	Not Applicable
	energy saving mode on ⇒ *turn off TV*		energy saving on ⇒ TV low power mode
C.4	home temp >= 70 ⇒ *ac on*	AC on and off at the same time	home temp >= 70 ⇒ fan on
	home temp >= 75 ⇒ *ac on*		home temp >= 75 ⇒ fan on
	user away mode on ⇒ *ac off*		Not Applicable
C.5	light lux <= 1500 ⇒ *open curtain*	P.9	Light lux <= 1500 ⇒ light on
C.6	user away mode on ⇒ camera on	P.3 ; camera on and off at the same time	Not Applicable
	user away mode on ⇒ *camera off*		user away mode on ⇒ motion sensor on
C.7	CO_2 density > 50 ⇒ fan on	Fan on and off at the same time	Not Applicable
	light on ⇒ fan on		Not Applicable
	time scheduled ⇒ *fan on*		Not Applicable
	humidity <= 50 ⇒ *fan on*		Not Applicable
	home temp >= 70 & outside temp <= 65 & ac off ⇒ *fan on*		Not Applicable
	home mode on & home temp > 75 ⇒ *fan on*		Not Applicable
	smoke detected ⇒ *fan on*		Not Applicable
	user shower ⇒ *fan on*		Not Applicable
	humidity <= 50 ⇒ *fan off*		humidity <= 50 ⇒ humidifier on
C.8	home temp >= 71 ⇒ *heater off*	Heater on and off at the same time	Not Applicable
	home mode on ⇒ *heater on*		home mode on ⇒ window open
C.9	motion detected ⇒ *lights on*	Lights on and off at the same time	Motion detected ⇒ play sound on speaker
	sleep mode on ⇒ light off		Not Applicable
C.10	arrive home ⇒ door open	Door open and close at the same time	Not Applicable
	CO_2 high ⇒ door open		Not Applicable
	home mode ⇒ door close		home mode ⇒ fan on

Table 5: Ten selected remedial actions provided by REMEDIOT after running the microbenchmarks.

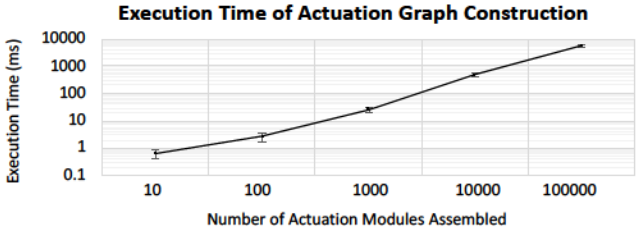


Figure 6: Average performance overhead of 50 iterations when constructing an actuation graph while varying the number of actuation modules.

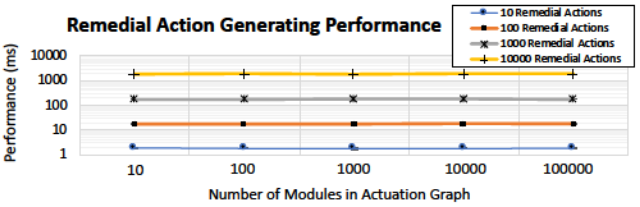


Figure 7: Average performance overhead of generating remedial actuations. We performed over 50 iterations while varying the size of the associated actuation graph.

possible conflicts. Table 5 describes *ten* selected remedial actions suggested by REMEDIOT framework. For instance, for conflict C.1, one event is trying to close the window when the user is away, and the other is trying to open the window to cool down the house. Because the former event has a higher priority for safety, the smart fan is chosen to realize the cooling abstraction. We now discuss the feasibility of deploying REMEDIOT in a real environment.

7.5 Configuration Time Overhead

Because REMEDIOT might be deployed in large scale smart environments such as smart cities, we evaluate the scalability of REMEDIOT. In particular, we calculate the overhead induced by REMEDIOT when constructing the actuation graphs. We investigate the execution time of actuation graph construction against the number of its associated actuation modules. For each different number of actuation modules, we average the execution time over 50 runs. Figure 6 shows a nearly logarithmically linear relationship between the number of assembled modules and the time consumed. Given the fact that the number of smart devices is limited, the time to construct the actuation graph can be ignored. Further, the actuation graph only needs to be constructed once before runtime, so configuration time may be a moot point.

We also obtain the overhead time of running our Remedial Action Engine by generating different quantities of remedial actions on different sizes of the actuation graph. We again run each experiment for 50 iterations to get the average overhead as well as the error range. As illustrated in figure 7, the performance of the Remedial Action Engine is irrelevant to the number of actuation modules in the actuation graph, which means that simply increasing the number of devices will not incur more overhead. The computation time logarithmically increases with the number of remedial actions that need to be proposed. Our results show that a personal desktop can easily enable constructing 100000 modules while suggesting up to 10000 remedial actions at runtime, highlighting REMEDIOT's scalability. Our results also validate the scalability of REMEDIOT even if more complex policies are added, i.e., more conflicts are triggered.

8 CONCLUSION REMARK

There are several future directions enabled by the RemedIoT.

Automating remediation. A future direction of this work is to automate such a process so that RemedIoT system is able to dynamically learn and make adjustments based on the application user's behaviors. Techniques such as reinforcement learning, crowdsourcing, and collaborative filtering can facilitate the usability of RemedIoT. However, one limitation is that RemedIoT relies on at least one redundancy path to achieve the same functionality.

Modeling sensors and actuators. Currently, the state space for sensors and actuators are binary, i.e. on or off. Cyber-physical modeling techniques can be utilized to better model different types of sensors and actuators. Further, we can dynamically optimize how an implementation unit implements an abstraction module at runtime. For instance, if an abstraction module performs human detection, a camera would be more effective than a microphone for certain environments.

In this paper, we proposed RemedIoT, a remedial action framework for resolving IoT conflicts. We evaluated the efficacy of RemedIoT on Samsung Smarthings and IFTTT applets in the context of a simulated smart home environment and showed that for a large set of applications, RemedIoT is able to resolve ~ 80% of conflicts. We further show the scalability of RemedIoT for generalizing to smart city environments.

ACKNOWLEDGMENTS

The research reported in this paper was sponsored in part by the National Science Foundation (NSF) under award # OAC-1640813 and CNS-1705135, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] Apple. https://www.apple.com/ios/home/.

[2] Arjunan, P., Batra, N., Choi, H., Singh, A., Singh, P., and Srivastava, M. B. Sensoract: A privacy and security aware federated middleware for building management. BuildSys '12, pp. 80–87.

[3] Balaji, B., Verma, C., Narayanaswamy, B., and Agarwal, Y. Zodiac: Organizing large deployment of sensors to create reusable applications for buildings. BuildSys '15, ACM, pp. 13–22.

[4] Celik, Z. B., Fernandes, E., Pauley, E., Tan, G., and McDaniel, P. Program analysis of commodity iot applications for security and privacy: Challenges and opportunities. *arXiv preprint arXiv:1809.06962* (2018).

[5] Celik, Z. B., McDaniel, P., and Tan, G. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 147–158.

[6] Celik, Z. B., Tan, G., and McDaniel, P. D. Iotguard: Dynamic enforcement of security and safety policy in commodity iot.

[7] Dawson-Haggerty, S., Krioukov, A., Taneja, J., Karandikar, S., Fierro, G., Kitaev, N., and Culler, D. BOSS: Building operating system services. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 443–457.

[8] Ding, W., and Hu, H. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 832–846.

[9] Dixon, C., Mahajan, R., Agarwal, S., Brush, A. J., Lee, B., Saroiu, S., and Bahl, P. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 25–25.

[10] Dixon, C., Mahajan, R., Agarwal, S., Brush, A. J., Lee, B., Saroiu, S., and Bahl, V. The home needs an operating system (and an app store). In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 18:1–18:6.

[11] Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., and Prakash, A. FlowFence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium* (2016), pp. 531–548.

[12] Hong, D., Ortiz, J., Whitehouse, K., and Culler, D. Towards automatic spatial verification of sensor placement in buildings. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings* (New York, NY, USA, 2013), BuildSys'13, ACM, pp. 13:1–13:8.

[13] Liang, C.-J. M., Bu, L., Li, Z., Zhang, J., Han, S., Karlsson, B. F., Zhang, D., and Zhao, F. Systematically debugging iot control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments* (New York, NY, USA, 2016), BuildSys '16, ACM, pp. 133–142.

[14] Liang, C.-J. M., Karlsson, B. F., Lane, N. D., Zhao, F., Zhang, J., Pan, Z., Li, Z., and Yu, Y. Sift: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks* (2015), ACM, pp. 298–309.

[15] Lin, J., Yu, W., Zhang, N., Yang, X., Zhang, H., and Zhao, W. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal 4*, 5 (Oct 2017), 1125–1142.

[16] Ma, M., Preum, S. M., and Stankovic, J. A. Cityguard: A watchdog for safety-aware conflict detection in smart cities. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation* (New York, NY, USA, 2017), IoTDI '17, ACM, pp. 259–270.

[17] Ma, M., Preum, S. M., Tarneberg, W., Ahmed, M., Ruiters, M., and Stankovic, J. Detection of runtime conflicts among services in smart cities. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 1–10.

[18] Ma, M., Stankovic, J. A., and Feng, L. Cityresolver: A decision support system for conflict resolution in smart cities. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems* (Piscataway, NJ, USA, 2018), ICCPS '18, IEEE Press, pp. 55–64.

[19] Microsoft. https://azure.microsoft.com/en-us/services/iot-edge/.

[20] Miettinen, M., Marchal, S., Hafeez, I., Asokan, N., Sadeghi, A.-R., and Tarkoma, S. Iot sentinel: Automated device-type identification for security enforcement in iot. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), IEEE, pp. 2177–2184.

[21] Mohsin, M., Anwar, Z., Husari, G., Al-Shaer, E., and Rahman, M. A. Iotsat: A formal framework for security analysis of the internet of things (iot). In *2016 IEEE Conference on Communications and Network Security (CNS)* (Oct 2016), pp. 180–188.

[22] Nacci, A. A., Rana, V., Balaji, B., Spoletini, P., Gupta, R., Sciuto, D., and Agarwal, Y. Buildingrules: A trigger-action–based system to manage complex commercial buildings. *ACM Trans. Cyber-Phys. Syst. 2*, 2 (May 2018), 13:1–13:22.

[23] Noor, J., Tseng, H.-Y., Garcia, L., and Srivastava, M. Ddflow: visualized declarative programming for heterogeneous iot networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation* (2019), ACM, pp. 172–177.

[24] Piscitello, A., Paduano, F., Nacci, A. A., Noferi, D., Santambrogio, M. D., and Sciuto, D. Danger-system: Exploring new ways to manage occupants safety in smart building. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)* (Dec 2015), pp. 675–680.

[25] SamSung. https://www.smartthings.com/.

[26] Shen, C., Singh, R. P., Phanishayee, A., Kansal, A., and Mahajan, R. Beam: Ending monolithic applications for connected devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 143–157.

[27] Stankovic, J. A. Research directions for the internet of things. *IEEE Internet of Things Journal 1*, 1 (Feb 2014), 3–9.

[28] Surbatovich, M., Aljuraidan, J., Bauer, L., Das, A., and Jia, L. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia, 2017), WWW '17, pp. 1501–1510.

[29] Surbatovich, M., Aljuraidan, J., Bauer, L., Das, A., and Jia, L. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes. In *Proceedings of the 26th International Conference on World Wide Web* (2017), WWW '17, pp. 1501–1510.

[30] Yahyazadeh, M., Podder, P., Hoque, E., and Chowdhury, O. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2019), SACMAT '19, ACM, pp. 61–72.

[31] Zapier. https://zapier.com/.