An Intent-Based Automation Framework for Securing Dynamic Consumer IoT Infrastructures

Vasudevan Nagendra
Stony Brook University
vnagendra@cs.stonybrook.edu

Arani Bhattacharya
KTH Royal Institute of Technology
aranib@kth.se

Vinod Yegneswaran
SRI International
vinod@csl.sri.com

Amir Rahmati
Stony Brook University
amir@cs.stonybrook.edu

Samir R Das
Stony Brook University
samir@cs.stonybrook.edu

ABSTRACT

Consumer IoT networks are characterized by heterogeneous devices with diverse functionality and programming interfaces. This lack of homogeneity makes the integration and secure management of IoT infrastructures a daunting task for users and administrators. In this paper, we introduce VISCR, a Vendor-Independent policy Specification and Conflict Resolution engine that enables *intent-based conflict-free policy specification and enforcement* in IoT environments. VISCR converts the topology of the IoT infrastructure into a tree-based abstraction and translates existing policies from heterogeneous vendor-specific programming languages, such as Groovy-based SmartThings, OpenHAB, IFTTT-based templates, and MUD-based profiles, into a vendor-independent graph-based specification. These are then used to automatically detect rogue policies, policy conflicts, and automation bugs. We evaluated VISCR using a dataset of 907 IoT apps, programmed using heterogeneous automation specifications, in a simulated smart-building IoT infrastructure. In our experiments, among 907 IoT apps, VISCR exposed 342 of IoT apps as exhibiting one or more violations, while also running 14.2x faster than the state-of-the-art tool (Soteria). VISCR detected 100% of violations reported by Soteria while also detecting new types of violations in 266 additional apps.

CCS CONCEPTS

• **Security and privacy → Access control**; *Usability in security and privacy*; *Security requirements.*

KEYWORDS

Intent-based policy and automation framework, Consumer IoT security, Conflict detection and resolution.

ACM Reference Format:
Vasudevan Nagendra, Arani Bhattacharya, Vinod Yegneswaran, Amir Rahmati, and Samir R Das. 2020. An Intent-Based Automation Framework for Securing Dynamic Consumer IoT Infrastructures. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3366423.3380234

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380234

1 INTRODUCTION

The adoption of the Internet of Things (IoT) has led to an explosion in the number of devices integrated into consumer IoT infrastructures (e.g., "smart homes", "smart campus", and "smart cities") [1, 2]. The heterogeneity of these infrastructures poses two major challenges in developing and enforcing policies across them:

(1) *Coherent Policy Expression*: Today, IoT device vendors support web and mobile-based apps, wide range of IoT automation frameworks, and specification languages [3–7] that allow users and administrators to program their IoT infrastructures. However, directly capturing the high-level automation (or policy) intent of IoT administrators, using these vendor-specific IoT apps, is a challenging task, which requires administrators of the IoT infrastructure to manually decompose their high-level intents into device-specific rules prior to installation onto IoT infrastructures.
(2) *Conflict-free Enforcement*: Consumer IoT infrastructures are programmed by multiple administrators having complex *roles* and varying levels of *skill*, which may include novice smart-home users (e.g., parents, kids), smart-campus or smart-city administrators (e.g., HVAC admins, fire-safety admins). Configuring *conflict-free* automation in such multi-administrative environments is a tedious process.

Recent studies independently underscore the need for new access control and policy frameworks to address the security goals of IoT ecosystem [8–10]. These solutions are designed to identify violations in IoT ecosystems with *homogeneous* programming specifications. However, they do not apply to contemporary consumer IoT deployments that commonly involve devices using diverse vendor-specific APIs and heterogeneous programming specifications. Furthermore, existing tools either rely on model checking for static analysis of IoT automation programs to detect the conflicts [9], or require IoT automation code to be instrumented for detecting run-time violations [10]. Existing tools and techniques also leave a wide spectrum of automation bugs and conflicts undetected. For example, *gap in automation* due to lack of expertise among administrators, *rogue policies* on infrastructures that the user does not control, violations that might arise due to loops among automation rules, and other potential run-time violations are some of the key issues left unaddressed by the existing automation frameworks (§5).

In this paper, we introduce "VISCR", a new intent-based IoT policy automation framework[1] that allows administrators of an IoT

[1] Intent-based policy frameworks provides administrators with the ability to directly capture their intention by decoupling "what" policies are to be enforced from "how" to implement and enforce them.

	Eco-system	Goal	Policy Intents of the IoT Administrator
P1	Campus	Safety, Privacy	Allow video feeds from camera to be sent to fire-safety admins in event of fire alarm.
P2	Campus	Security	Cameras for monitoring is turned OFF between 9PM – 7AM, and turned ON only when motion is detected.
P3	Home	Safety	Between 10PM – 6AM open main door to kids & guests only with authorized user's approval (e.g., parents).
P4	Home	Safety	In case of a fire event, warn users, and then open the windows and doors to allow residents to exit safely.

Table 1: Policy conflict examples in consumer IoT infrastructures.

infrastructure to directly and succinctly capture their dynamic automation use-cases and policy intents in a vendor-independent manner. For effectively accommodating the automation rules and policies that are already specified in the existing IoT infrastructure using heterogeneous programming specification languages[2], `VISCR` translates these IoT automation programs into vendor-independent graph-based specifications (§ 4.1). `VISCR` uses these graphs to detect bugs and conflicts across policies that could otherwise go undetected using existing model checking-based tools [9, 10]. These include: (*a*) static compile-time conflicts, (*b*) gap in the automation, (*c*) conflicts due to chaining and loops among automation rules, (*d*) access violations, (*e*) potential run-time violations, and (*f*) rogue policies. `VISCR` supports automatic conflict resolution using the `PRECEDENCE` operator. Any unresolved conflicts are forwarded to appropriate IoT users or administrators for manual resolution. The policy reconciliation engine decomposes the composed conflict-free policy graph into a set of device-specific rules for enforcement onto actual devices (e.g., IoT devices, IoT Gateways) as IoT apps and ACL rules (§5.2.3).

We evaluate `VISCR` on a simulated smart-building IoT infrastructure with 907 apps. `VISCR` detected a wide range of bugs i.e., ≈37.7% of IoT apps are reported for one or more conflicts and bugs compared to ≈8.4% of static compile-time conflicts detected with existing model checking-based tool [9] while incurring less than 3.8% false positives. We discuss the resultant conflicts and bugs detected by `VISCR` and their categories in §6.

In summary, our paper makes the following key contributions:

- We design `VISCR`, a vendor-independent graph-based policy specification mechanism that translates automation rules and policies specified using heterogeneous programming specification languages into vendor-independent graph-based policies. (§4).
- We implement efficidnt graph composition techniques to detects bugs and conflicts that arise among the policies and resolve them using `PRECEDENCE` operation for conflict-free enforcement (§5).
- We evaluate `VISCR` using 907 IoT apps (i.e., both vetted and unvetted) in a simulated smart building infrastructure with real world automation use cases reported by IoT users and administrators and compare with existing techniques (§6).

2 CHALLENGES AND RESEARCH GOALS

A wide variety of automation frameworks and specification languages are supported by IoT device vendors for automating IoT infrastructures [3–7]. Such heterogeneity makes programming the IoT infrastructure a challenging task. In addition, the presence of multiple administrators[3] in IoT infrastructures with different roles and responsibilities further complicates the situation. Hence, the *heterogeneity*, and lack of a *unified policy-specification* in multi-administrative IoT infrastructures forces administrators to independently develop automation policies and manually detect conflicts and violations.

2.1 Intuitive Policy Specification

IoT infrastructures are typically managed by multiple administrators, each of them responsible for the management of a specific group of devices. For effectively capturing policy intents from multiple administrators, a policy framework should support following capabilities: (*i*) Ability to logically group devices in accordance with the policy requirements of each of the administrator. For example, the IoT administrators handling cameras of BLDG1 and BLDG2 of the campus network should have abstractions that logically group the devices belonging to those locations; (*ii*) Provide isolation among administrators while exposing only the necessary abstractions required for policy specification avoiding *rogue policies* from being specified. For example, a fire-safety administrator should not be exposed to other IoT infrastructure details (e.g., cameras, HVACs etc.) unless cross-device policy specification is required.

Currently, it is challenging to provide logical isolation across different policy administrators and the devices they administer, which motivates the need for fine-grained abstractions and logical grouping of IoT devices (Figure 3). Also, the lack of expertise in programming using heterogeneous automation specification languages makes the IoT infrastructures prone to errors [11–15].

2.2 Conflict Detection & Resolution

Policy intents of IoT administrators are implemented using discrete automation rules configured onto each device using the mobile or web-based user interfaces, and automation specification languages [3, 5, 6, 16]. The existing consumer IoT ecosystem lacks means to effectively detect the conflicts, bugs and violations that arise among the policies that are captured independently by each of the administrators using heterogeneous specification languages and reconcile them onto each of the IoT devices.

Recent studies have demonstrated that major vulnerabilities in the IoT infrastructure are commonly due to simple human errors and lack of expertise in policy configuration [11, 12, 17, 18]. As highlighted in Table 4, there are range of automation bugs, policy conflicts and violations (e.g., gap in automation, potential run-time conflicts, and code sanity bugs) that could go undetected with existing conflict detection techniques [9, 10, 19]. Let us consider following policy conflicts:

Policy Conflict 1: Consider for example two policies P3 and P4 (Table 1) with conflicting actions in opening main door. In case of a fire alarm after 10 P.M., (both policies P3 and P4 are activated) the current IoT automation keeps the door locked due to policy P3, preventing users from leaving and fire-safety officers from entering.

Policy Conflict 2: Similarly, policies P1 and P2 results in run-time violation. When video feed is shared with fire-safety staff, the feed is interrupted by the camera's idle-time event i.e., turns OFF the

[2] For example, Groovy-based programs for smartthings [5], OpenHAB-based rules [3], MUD-profiles [6] and, IFTTT-based applets [7].

[3] For example, the smart home automation rules are specified by the members of the family (e.g., parents, kids, guests), while smart campus and enterprise IoT infrastructures are managed by different types of IoT administrators such as HVAC, fire-safety, utilities & energy, and infrastructure (or building) administrators.

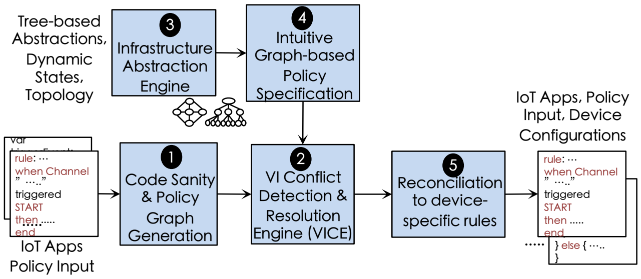

Tree-based Abstractions, Dynamic States, Topology
3
Infrastructure Abstraction Engine
4
Intuitive Graph-based Policy Specification
IoT Apps, Policy Input, Device Configurations
var
rule: ···
when Channel
" ···.."
triggered
START
then .....
end
IoT Apps Policy Input
1
Code Sanity & Policy Graph Generation
2
VI Conflict Detection & Resolution Engine (VICE)
5
Reconciliation to device-specific rules
rule: ···
when Channel
" ···.."
triggered
START
then .....
end
} else { ···..
}

Figure 1: VISCR: Highlevel system architecture.

camera interrupting the feed from being shared, leading to safety violations. Proactively detecting and resolving the conflicts 1 & 2 discussed above in compile-time is a challenging task as these policies depends on asynchronous events (e.g., inactivity timer) and environmental conditions (i.e., time and motion sensing).

3 VISCR: SYSTEM OVERVIEW

We describe the components of VISCR's policy specification and enforcement framework. As shown in Figure 1, VISCR performs sanity checks on IoT automation programs that are captured using heterogeneous specification languages [3, 5–7] and then translates them into vendor-independent graph-based specifications (❶). VISCR maintains mapping between the IoT programs, associated device configurations, and graph-based policies as *policy mappings*, which are then used in policy enforcement phase i.e., for reconciling composed policy graph to device-specific rules (§5.2.3). For translation of vendor-specific automation rules to vendor-independent graph-based policies, we have built necessary lexing and parsing grammar (.g4), and mapping (.map) files specific to each of the automation specification language with ANTLR (§4.1).

To enhance the usability of our policy framework, VISCR also supports a drag-and-drop interface for graph-based policy specification (❹) and input policy entities that follow tree-based infrastructure abstractions (§4.2.2). The tree-based abstractions that are required for each IoT administrator to specify policies (❸) are automatically constructed using data sources present within the IoT infrastructure and cloud interfaces (§4.2.1).

The vendor-independent graph-based policies (i.e., outcome of modules ❶ & ❹) are inputs to the VICE module for detecting the conflicts, violations and bugs. VICE initially detects *rogue policies* (❷) that are configured by policy administrators on any specific portion of the IoT infrastructure they are not authorized to specify automation rules (Section 4.2.2). In the next step, the policy composition engine uses graph-based composition algorithms and *precedence* mechanisms for automatically detecting the conflicts and resolving them respectively. Unresolved conflicts and bugs are reported to the administrators of IoT infrastructure for manual resolution (❷).

The composed policy graphs are further analyzed for security bugs and violations (§5.2): (*i*) *gap in the automation* that could make infrastructure vulnerable, (*ii*) *loops* that exist among automation rules, (*iii*) access violations, and (*iv*) *potential conflicts* that could arise in run-time. Finally, the composed conflict-free policy graphs and policy mappings are used to reconcile the policies into device-specific rules for enforcing (❺).

4 VENDOR-INDEPENDENT SPECIFICATION

Realizing coherent automation in existing IoT infrastructures that uses heterogeneous specification languages is a challenging task. It requires administrators to analyze each of the IoT Apps manually for detecting the conflicts and bugs for manually resolving them, which is a tedious process and could be prone to error. Hence, we propose a vendor-independent specification engine (VISCR) that serves following key purpose: (*i*) translates IoT automation programs and policies specified using Groovy, IFTTT, MUD, and OpenHAB-based programs into a vendor-independent graph-based specification, and (*ii*) performs sanity checks on IoT apps.

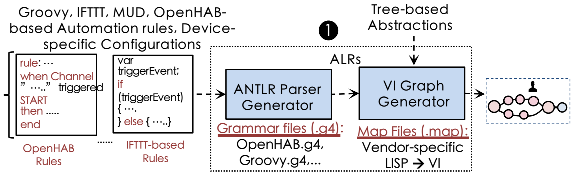


Figure 2: Functional block diagram of graph generation module. architecture for generating vendor-independent graph-based specifications from vendor-specific automation rules.

4.1 Vendor-Independent Model

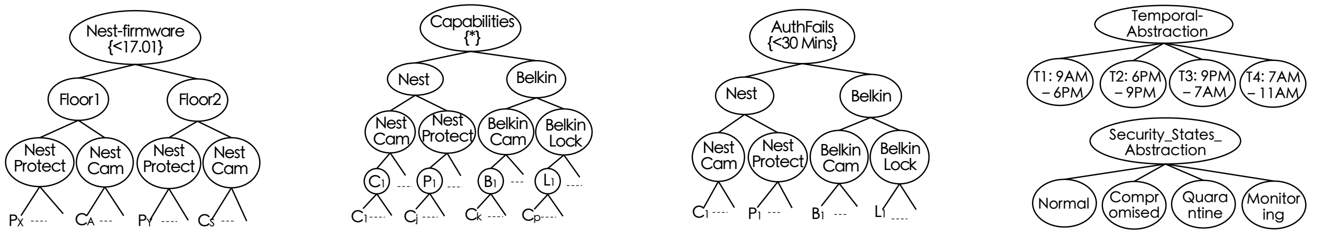
As illustrated in Figure 1, Code sanity and Graph generator module of VISCR engine consumes vendor-specific IoT Apps (i.e., IFTTT-based Applets, Groovy-based SmartThings, OpenHAB programs) and translate them into vendor-independent trigger and action-based policies (as shown in Figure 5). Similarly, the *MUD profiles*[4] and translated into vendor-independent graph-based ACL policies (as shown in Figure 4). This approach allows each of the functional component of VISCR i.e., composition engine to operate seamlessly on vendor-independent (VI) model. As shown in Figure 2, VISCR generates the vendor-independent (VI) graph-based policies using following key functional modules:

- *ANTLR Parser Generator*: As a first step, we develop the lexer and parser grammar (.g4) files required to translate the vendor-specific IoT apps to an abstracted intermediate representation (i.e., Abstracted LISP Representation (ALR) and Abstracted Tree Representation (ATR) formats) with ANTLR module. ANTLR parser generator uses the Abstracted LISP representations (ALR) of the IoT Apps for performing the code sanity analysis. Note, both the ALR and the ATR representations of IoT apps are not exposed to end users, only the outcome of VI graph generator (❶) will be exposed to users and composition engine.
- *VI Graph Generator*: In the next step, automation rules represented in the Abstracted LISP syntax representation are consumed by the VI Graph-generator module to translate it to vendor-independent graph-based policies (as shown in Figures 5 & 4). We built (.map) file, which is used for maintain necessary mappings between different vendor-specific automation/policy attributes and vendor-independent graph attributes. These vendor-independent attributes (or labels) are used in the construction of final vendor-independent graph-based representation. The vendor-independent graph-based specifications outcome of VISCR (❶) is captured using networkx python library for maintaining the policy graphs, required for composing the policy graphs to detect the conflicts (discussed in Section 5).

4.2 Abstractions & Graph-based Specification

As discussed in Section 4.1, VISCR supports mechanism to translate the vendor-specific IoT Apps into vendor-independent graph-based specification. In addition, VISCR also supports simple drag and

[4] Currently MUD-profiles are limited to support ACL-based traffic filtering rules. It does not support dynamic trigger and action-based policies yet.



(a) Nest device's firmware version <17.01 specific to location & type.

(b) Capability-specific abstraction for nest & belkin devices.

(c) Vendor-specific authentication fails in last 30mins.

(d) Temporal and security state abstractions Trees.

Figure 3: Automatically constructed tree-based abstractions for the smart campus IoT infrastructure: (a) captures devices as leaf nodes and abstraction, (b) captures device capabilities and property values as leaf nodes, (c) captures the dynamic security states (e.g., authentication) specific to IoT devices, (d) temporal and security state abstractions of IoT infrastructure.

drop-based user interface that allows administrators to directly and intuitively capture their policies in a vendor-agnostic manner using nodes of infrastructure abstraction trees, as discussed below.

4.2.1 Tree-based Abstractions. The *Infrastructure Abstraction Engine* automatically builds necessary abstraction trees required for graph-based policy specification. The abstraction engine continuously churns data from local IoT infrastructure and cloud data sources using device and vendor-specific APIs for building the abstractions [20, 21]. The data extracted from IoT device's data sources as text is translated into data tables by the data-source-driver engines that we designed for each vendor's data format.

Generation of such abstractions allow administrators to delegate the responsibility of policy specification to sub-administrators. For example, it is now possible for a global building/campus administrator to assign Floor1 and Floor2 responsibilities to different sub-administrators. Also, VISCR allows abstraction trees to be built on physical or logical grouping of device (e.g., Figure: 3a & 3b). The global administrator simply uses *abstraction mappings* to delegate infrastructure to each of the administrator, illustrated below. Depending on the assigned *abstraction mappings* the abstraction engine generates the abstraction trees. For example, to derive the abstraction tree shown in Figure 3a (i.e., *list of Nest devices of BLDG1 with firmware <17.01 organized as per floor and type of devices*), the abstraction-mapping required is:

```
abstraction-tree{"Nest-Firmware{<17.01}"} =
    location{BLDG1}.floors{*}:
    vendor-type{Nest}.device-category{*}:devices{*}
```

As another example, the abstraction-mapping for capturing the list of devices and their capabilities with respect to their vendor and device-types (Figure 3b) is provided below:

```
abstraction-tree{"Capabilities{*}"} =
    vendor-type{*}:vendor-type{*}.device-category{*}:
    devices{*}:capabilities{*}
```

Representing Abstractions: We represent the complete IoT infrastructure as set of *infrastructure-abstraction trees*, specific to each of the administrators. The root node of the abstraction tree (i.e., abstraction tree name) uniquely represents each of the abstraction tree present in any IoT ecosystem. The leaf nodes represents set of individual devices. Each intermediate node represents different infrastructure abstractions such as device-types, device-vendors, location, temporal, application details. The infrastructure abstractions trees implicitly capture the boolean Union operator represented as sibling nodes in the abstraction tree. Similarly, the boolean AND operator corresponds to the relationship between the child and parent nodes of the abstraction tree. For representation, each level of abstraction is separated with ":" operator, while the constraints to be imposed on to that level is represented with a "." operator.

VISCR supports a diverse set of abstractions, which will allow IoT administrators to capture different types of policies based on their security status, locations, vendor-type and so on. These include abstractions specific to: (*i*) *security-state*, (*ii*) *location*, (*iii*) *applications*, and (*iv*) *device- or vendor-specific abstractions*.

4.2.2 Graph-based Specification. As discussed in Section 4.1, VISCR engine translates the vendor-specific IoT Apps into two different types of policies: (*i*) *trigger- and action-based policies*, and (*ii*) *dynamic ACL-based policies*. In addition to this capability, VISCR also supports intuitive graph-based specification framework that allows IoT administrators to directly express their policy intents through simple drag and drop-based user interface (UI), which uses policy attributes from tree-based abstractions.

Trigger- & Action-based Policies: Trigger & action-based policies allows the administrator to capture their policy intents for set of IoT devices that perform some predefined action in response to an triggering event. These types of policies can be specified as complex graph-based policies, where each of the node captures abstractions, device names, conditions or actions. Our trigger- and action-based policy graphs have the following format: source IoT device with associated states, set of conditions, dynamic states, associated set of events and respective action/s on target IoT devices (examples in Figure 5).

The source node represents the IoT device on which the event is received. Rest of the nodes represent conditions and state of the IoT infrastructure. The sequential and parallel operators >> and || are used to specify the sequence of IoT-commands that need to be executed. The +/− operators are used to add or remove the list of ACLs that are specific to the current state or condition.

ACL-based Policies: This type of policies either allows or restricts the communication between devices and internet according to the dynamic infrastructure states and conditions. Examples of graph-based access control policies are shown in Figure 4. For the trigger and action-based policy shown in Figure 5, we implicitly add ACLs to ALLOW traffic between devices (i.e., Belkin CO device and other devices on which the action is enforced. In reality, as the communication happens indirectly between the hub or cloud interface and the Belkin device, equivalent ACLs are added to the network.

The starting and end nodes represents the source and destination entities of policies. The edge between the source and target node captures following properties: (*i*) set of network functions through which the traffic should traverse, i.e., network function chain (NFC), (*ii*) conditions to be enforced on the traffic depending on the state, and (*iii*) actions to be taken on the traffic between the source and destination entities. With both the ACLs and trigger-action-based

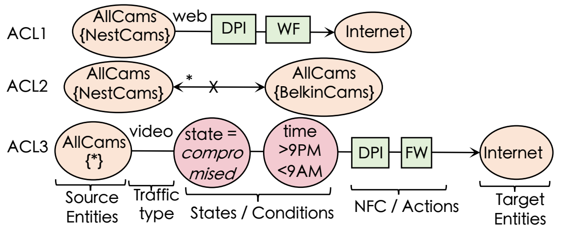


Figure 4: ACL-based IoT policy specification.

policies, the states and conditions are represented as nodes along the path for simplified design in the drag-and-drop-based graph specification framework. Alternatively, these could be represented as simple edge properties. The → and −x→ arrowed-lines represents the action (i.e., ALLOW/DENY) on the traffic.

4.2.3 Graph-based Policies Specification Syntax. An equivalent specification syntax (Figure 6) is required for capturing each of these graph-based policies in the backend. For example, in the ACL-based policy specification syntax, the permissions to communicate between source and target nodes is specified using action attributes => (i.e., ALLOW) and ! => (i.e., DENY) symbols. The sequence of network functions through which the traffic from a specific source node should traverse to reach the target node or the sequence of actions to be taken is specified using the sequential or parallel operators (e.g., FW (Firewall) >> DPI (Deep packet inspection), FW || LB (Load balancer), Exhaust=ON >> ExhaustSpeed=High). Note that the sequence and parallel operators are used for traversal of a set of middleboxes in the case of ACL-based policies. In case of trigger-action-based policies, it is used to represent the sequence of actions. Also, in the trigger-action-based policy specification syntax, actions are captured using the `iot-commands-action` keyword. The wide range of attributes, keywords and symbols used in the policy specification syntax is captured in Table 2.

4.2.4 Rogue Policies. VISCR's *abstraction trees* and *graph-based specification* allows the administrators to specify the policies explicitly using the abstraction trees exposed to each of the IoT users or administrators. This approach of isolating and assigning explicit infrastructure abstractions to each admin, allows VISCR to prevent admins from specifying policies on the infrastructure they do not own (i.e., called rogue policies). However, policies that are directly specified using policy abstraction syntax (§4.2.3) (i.e., bypassing the web-based user interface) may introduce violations. For detecting *rogue policies* in syntax-based policy specification, the policy composition engine extracts the source and target nodes from the specified policy specification syntax and verifies if both the nodes belongs to the *policy abstraction trees* owned by that administrator. If the policy attributes used in the policy specification are part of the abstraction trees assigned to that admin, then the policies are allowed to perform actual policy composition to further detect other conflicts and violations (discussed in Section 5.2). Policy violations detected in this step are reported as *rogue policies*.

5 DECONFLICTION & SECURITY ANALYSIS

In this section, we discuss about graph-composition techniques and security analysis used by VISCR for detecting variety of conflicts, bugs and violations that arise among the automation rules.

5.1 Graph-based Composition

The policy composition in VISCR seeks to provide the following capabilities: (*i*) composition of policies for proactively detecting conflicts in compile time rather than detecting at run time, (*ii*) automatic resolution of conflicts using *precedence* mechanism, and (*iii*) *incremental composition* that allows trigger-action policies to adapt to the environment at runtime. However, allowing run-time policy composition imposes stringent latency constraints. Hence, policy composition time has to be minimized.

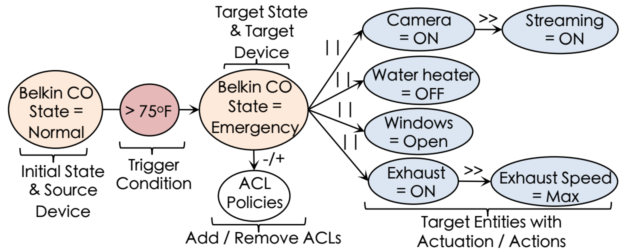


Figure 5: Dynamic trigger & action-based IoT policy specification.

Type	Symbol	Definition
Policy specification keywords	location, devices, device-type, device-vendor, parent, traffic-type, source-node, target-node, source-state, target-state, etc.,	Keywords for capturing policy attributes and the properties of the IoT infrastructure.
Sequence & Precedence Operators	> > (serial or precedence), || (parallel), → (flow / action sequence)	Operators to specify the sequence of operations to be carried out in policy.
Conditional Operators	!, =, <, >	Operations used to specify dynamic conditions in the policy.
Composition Operators	policy-add (+), policy-remove (−)	Operators to add and remove policies from the existing list of policies.
Action attributes / Keywords	iot-commands-action, => (ALLOW), ! => (DENY)	Attributes or keywords used to specify the actions to be taken in the policies.

Table 2: List of keywords, attributes, operators and symbols used in VISCR policy specification syntax.

```
iot-devices{smoke_co_alarms}.vendortype{*}.co_alarm_state{emergency}
    => iot-commands-action {
          Cameras:switch=on >> Cameras:streaming=on | |
          Water_Heater:switch = off | |
          Window-sensors:state = open | |
          Exhaust_switch:switch = on >> Exhaust_Regulate:speed = max
    } + PSet1
```

(a) Equivalent syntax for dynamic trigger–action IoT policies.

```
Spec{ACL1}: AllCams{NestCams}.parent{BLDG1->Floor1}.device-vendors{Nest}
.device-types{Cameras}.traffic-type{web} >> DPI >> WF => networks{Internet}

Spec{ACL2}: AllCams{NestCams}.parent{BLDG1->Floor1}.device-vendors{Nest}.
device-types{Cameras}.traffic-type{*} !=> AllCams{NestCams}.parent{BLDG1-
>Floor1}.device-vendors{Belkin}.device-types{Cameras}

Spec{ACL3}: AllCams{*}.parent{none}.device-vendors{*}.device-types{Cameras}.
temporal-state{>9PM <7AM}.security-state{compromised} >> DPI >> WF >>
networks{Internet}
```

(b) Equivalent policy specification syntax of ACL-based policies (ACL1 – ACL3) illustrating different properties.

Figure 6: Examples of ACL-based & trigger and action-based IoT policy specification syntax.

VISCR's policy composition procedure involves following key steps. First, the *normalization* step brings all the policies specified by various administrators, using different abstraction trees, to a common abstraction level for identification of contradictory and duplicate policies. The second step, finds contradictions among normalized policies by running composition engine and resolves the conflicts using precedence rules. Unresolved conflicts are flagged to the administrator.

5.1.1 Proactive & Incremental Composition. The goal of policy composition mechanism is to produce *conflict-free composed policy*

Algorithm 1 Graph-based Policy Composition

```
 1: $L \leftarrow$ list of normalized policies to be added
 2: $s(p) \leftarrow$ source node of policy $p$
 3: $t(p) \leftarrow$ destination node of policy $p$
 4: $a(p) \leftarrow$ action of policy $p$
 5: $b((s, t)) \leftarrow$ action of edge $(s, t)$
 6: $G \leftarrow$ Composed graph
 7: for all Policy $p \in L$ do:
 8:     if $s(p) \in G$ & $t(p) \in G$ then
 9:         if ($b(s, t) \in G$ & $a(p) == b(s, t)$ then
10:             Discard $p$                                        ▷ Duplicate Policy
11:         else if $b(s, t) \in G$ & $a(p) \neq b(s, t)$ then
12:             Apply $b(s, t)$ or $a(p)$ based on precedence
13:             Raise conflict alert if policies have equal precedence
14:         else if $b(s, t) \notin G$ then
15:             Create $b(s, t)$ from $p$
16:             Add $b(s, t)$ to $G$                                ▷ Add policy
17:     else
18:         if $s(p) \notin G$ then
19:             Create $s(p)$
20:         if $t(p) \notin G$ then
21:             Create $t(p)$
22:         Create $b(s, t)$ from $p$
23:         Add $b(s, t)$ to $G$                                    ▷ Add policy
    return $G$
```

graph (example shown in Figure 7) that is derived by composing together overall vendor-independent graph-based policies (i.e., either generated from IoT apps or specified directly by policy administrators). In the final composed graph, the source and target entities represent the devices or group of devices onto which policies are executed. The edges capture the set of conditions for policy activation and the corresponding actions.

As a first step of policy composition, `VISCR`'s normalization mechanism identifies the common abstraction level to which all policies specified by administrators need to reduced for conflict detection. If policies are specified using abstraction nodes at different levels of an abstraction tree, composing policies without normalization is an infeasible task. For example, consider two policies specified using the abstractions BLDG1 and Floor2 (homogeneous abstractions) or using BLDG1 and NestCams (heterogeneous abstractions). Here, we do not know if the Floor1 node and Nest node have any relation (i.e., subset, superset, or overlapping hosts or devices) for composing them together into a single policy graph. We resolve this issue by (*a*) automatically deriving relations across non-leaf nodes of different abstraction trees and maintain these relations as mappings i.e., capturing the details of set of devices or hosts that belongs to each of these nodes of abstraction trees, and (*b*) choosing an optimum level to which the nodes used in the policy specification need to be normalized.

Normalization: A naive normalization approach is to bring all the policies to the bottom most level, i.e. leaf node level which captures the device-specific details for performing composition. This approach increases complexity along two dimensions: (*i*) It brings down all policies to the bottom most level even though very small number of policies might need it, exponentially increasing the composition times, and (*ii*) normalizing all the source and destination policy nodes to bottom most level (i.e., to the level of leaf nodes) also increases the enforcement complexity, due to increase in the number of rules required to enforce the policies. Therefore, `VISCR` finds an intermediate optimum level i.e., Enforcement Level (`ELevel`) for each policy abstraction tree for effectively normalizing the policies. The enforcement level (`ELevel`) is chosen in such a way it would allow the policy enforcement engine to directly compile/translate these composed policies to enforceable rules.

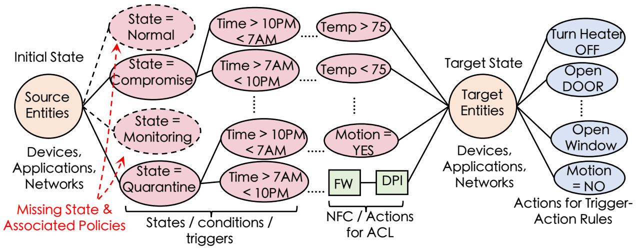


Figure 7: Sample outcome of graph-based policies composed together representing final composed policy graph. Possible missing states are highlighted in the figure with dotted lines.

Mathematically, we represent the choice of enforcement level as follows. Let K_i be the abstraction level using which a policy $P_i (i = 1, \ldots, N)$ is specified. Then, policy P_i can be normalized to abstraction level $A_i \geq K_i$. Thus, we aim to normalize to a level A such that $A \geq A_i \; \forall i = 1, \ldots, N$. However, the number of nodes increases with an increase in A_i. Thus, our objective is to choose the minimum value of A that satisfies above set of constraints, i.e.,

$$\text{Minimize } A \text{ subject to: } A \geq K_i \; \forall i = 1, \ldots, N. \qquad (1)$$

Algorithm 1 describes the graph-composition algorithm, which accepts: (*i*) a list of graph-based policies that are normalized, and (*ii*) an empty graph as input, which is used for storing the composed policy graph. Each of the input policy is iteratively checked with the composed graph for identifying an identical or conflicting policy. Each policy's source node, is compared with the source nodes of the composed policy graph for checking the overlaps. Next, the target node of the input policy is compared with the target nodes of the composed policy graph. Finally, the conditions and action attributes present along the edge of the input policy are compared for overlaps with the edge attributes of the composed policy graph.

The algorithm attempts to resolve conflicts by checking whether a precedence rules exists for any of the policies. Duplicate policies are tracked separately and discarded from composition graph. If it finds a conflicting policy that cannot be resolved with precedence, the policy is dropped from the graph (Lines 11-13) and notified to administrator. If it finds that the source node and the destination node both exist and no conflicts are possible, the policy is then added to the graph (Lines 14-16). Otherwise, it creates new nodes and then adds the policies as edges to the graph (Lines 17-23).

To analyze the algorithmic time complexity, we need to evaluate the cost of iterating over the complete list of policies (L), and adding them to the composed Graph G. With the addition of each policy to the composed policy graph G, the composition engine checks for the existence of conflicts. The major time complexity of the algorithm lies with the iteration of the policies O(L) over the list of all source nodes L_s in the composed graph G, and then comparing the policy's source node $s(p)$ to composed graph's source nodes $S(G)$. L_e is the list of edges for which the edge-properties overlap with the policy's edge among the overlapping source nodes. Also, L_t is the list of target nodes of the edges of $s(p)$ that actually overlaps with the target node of the policy p_i. Therefore, the overall worst-case complexity is: $O(L * L_s * L_e * L_t)$.

For reducing the composition complexity we employ hashing mechanism and caching technique: (*i*) the m host entries of $s(p)$ are hashed as key-value pairs and the host entities of $S(G)$ are looked up in the hash for the existence of the n hosts, reducing baseline complexity will be reduced to: $O(L * L_s * (m + n))$. (ii) caching the comparison calculation outcome as key-value pairs ($s(p)$:$S(G)$) in hash table reduces the overall baseline complexity to $O(L * L_s)$.

5.1.2 Precedence. Precedence rules are used to resolve conflicts among competing policies specified at different levels. *Administrator-level precedence* evaluation is based on the scope of authority of the policy author. For example, a campus-level administrator in a smart campus may be granted precedence over a building administrator; *Action-level precedence* allows for explicit prioritization in action invocation. For example, for IoT traffic's ACL-based policies, Drop > Allow > Quarantine > Redirect can be used as the precedence hierarchy. Similarly, in the case of trigger-action-based policies, when the smoke detector is in `fire-alarm` state, the action turn OFF heating is given higher precedence than turn ON heating. *Custom precedence* enables policy attributes (e.g., `user`, `device-type`, `device-state`) to be associated with precedence. Based on precedence, the overlapping nodes that result in conflict are removed and the edge specific to the policy with highest precedence is retained.

5.1.3 Incremental Updates. The dynamic characteristics of the consumer IoT infrastructure demand that the policy framework be agile in enforcing new set of rules to IoT devices. Since complete policy composition consumes time, up to a few minutes, efficient re-composition techniques are required for rapid policy response. Hence, we use *incremental policy composition* to ensure an expedient response to dynamic conflicts that arise in the network. The composition engine recomposes only the updated set of policies with the whole set of composed policies. Updating a policy from the composition graph involves first deleting the policy from the graph, and then inserting a modified version. Deleting a policy requires one to remove the edges that belong to the policy from graph. However, the composition procedure might have removed portions of other policies that had a higher precedence during conflict resolution. Hence, these lost portions must be returned.

Incremental policy composition reduces the time to react to dynamic state changes and enforce new rules under the following scenarios: (i) *Scenario 1*: New polices are added or removed from the IoT infrastructure; (ii) *Scenario 2*: IoT infrastructure changes (e.g., device location updates, new devices added or existing devices removed); and (iii) *Scenario 3*: IoT device-state changes (e.g., from *normal* to *compromised*). Also, for run-time composition `VISCR` enumerates all the possibilities to pre-calculate the possible outcomes. Hence, for run-time enforcement `VISCR` simply identifies the outcome specific to the event and simply executes it, which can be carried out in sub-second latency, much faster than incremental composition.

5.2 Security Analysis & Policy Enforcement

In this section, we describe about: (i) how we use the composed policy graphs to perform security analysis for detecting the bugs and violations, and (ii) conflict-free policy enforcement by reconciling policies to device-specific rules.

5.2.1 Gap Analysis. Gap in the automation could leave the IoT infrastructure in a state that is either unstable or unpredictable in its behavior. Such dangling state could make IoT infrastructures vulnerable to attacks. For example the policies S_3, S_4 and S_8 shows the gap in the automation with respect to its temporal and temperature conditions (Table 4). It is evident from these policies that during 8PM −9PM (conflict) 9PM − 8AM (Gap) the thermostat's temperature settings could not be effectively predicted. As listed in Table 4, similar automation gap could be visible in other types of policies specific to its spatial, security states and other environmental conditions. Therefore, identifying the gap in automation is key step towards detecting the *potential* bugs that might arise in the IoT infrastructure during run-time.

The `VICE` module traverses through the final composed policy graph to identify the missing states or conditions for which the policies are not captured (as shown in Figure 7). This is achieved by enumerating and verifying if the policies exist for all the possible temporal, spatial and security conditions (e.g., states captured as part of abstraction tree's leaf nodes such as shown in Figure 3d), which helps in identifying potentially missing policies i.e., gap in automation. For example, the possible security states of IoT infrastructure could be *Normal*, *compromised*, *monitoring* and *quarantine* (Figure 3d). By identifying the missing states and their associated policies using the abstractions tree as reference, we can effectively detect the gap in automation.

The completeness of the gap analysis depends on the abstraction engine's ability to extract all the possible states, conditions and infrastructure details captured as part of the abstraction tree. For example, the temporal, spatial and security abstractions that are auto-generated by the abstraction engine could be verified by user for its correctness, allowing the administrator to add the missing states as leaf nodes to the abstraction trees.

5.2.2 Loops in Automation & Potential Conflicts. In general detecting chains and loops within the automation rules is essential to detect the potential conflicts and violations that might arise during run-time, which are rather challenging to be detected at policy compile time. From the composed policy graph, we detect the loops as follows: (i) We check for existence of paths with in a composed policy graph that has more than one trigger and action pair (i.e., chaining among policies). (ii) Check if any of the actions along the path triggers back any of the events with in the chain. (iii) Check if any of the actions along the path triggers any of the events with in the chain resulting in taking different action, which results in ambiguity and continuous toggling of states and actions. Loops and chaining among the policies (e.g., policies S_3, S_4, S_5 and S_6 in Table 4) with continuously toggling actions are marked as conflicts, while the identified policy chains are marked for potential violations. These policies when realized together will result in either unintended behavior, continuous toggling of actions or might result in unsafe state i.e., leaves door locked or opened at unanticipated time resulting safety violations or sets unintended temperature conditions in home.

Consider for example policies S_4, S_6 and S_9, which results in run-time violations. These policies does not have any thing in common except the actions taken by them i.e., S_6 closes windows in case of specific outdoor temperature, while S_9 opens the windows in case of raining and humidity. Therefore, identifying the run-time conflicts is a challenging task with such policies. As a straw-man solution one could mark all the policies that has conflicting actions and lack of specific temporal attributes among these policies as potential violation. Such approach will result in generating vast

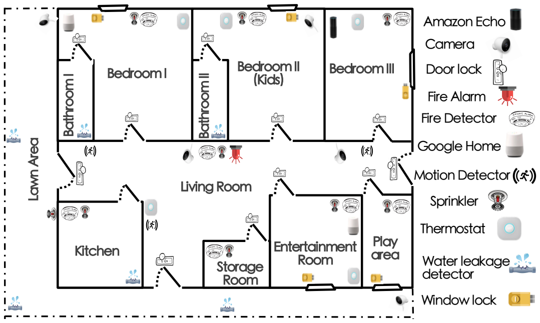


Figure 8: Simulated smart building infrastructure with 907 IoT Apps that uses groovy-based smartthings, OpenHAB rules, IFTTT-based Applets and MUD profiles with 30 different devices from 8 vendors. For simplicity only 11 different types of devices are shown.

false positive. Exposing large number of false positives to novice smart home users could prevent them from using the automation framework, rendering these tools useless.

Therefore, it is essential to proactively identify potential violations and the policies suitably fine-tuned to avoid run-time violations. To identify *potential* run-time violations VISCR, checks for *mutually exclusiveness*[5] among the policies. For detecting *potential run-time violation*, we build relation among all the events, states and environmental conditions that are part of the IoT infrastructure and cluster the events. By clustering the events, states and conditions, we will be able to effectively detect if parameters among any two policies are mutually exclusive or not. To achieve this, we encode our policies [22] (i.e., captured as policy tuple: <*source-node*, *source-state*, *edge states/conditions*, *target-node*, *target-state*, *actions*>) and use k-means clustering with elbow method [23] for effectively clustering different types of triggering conditions and associated states. If the policy states and conditions are not mutually exclusive (i.e., on the basis of cluster distance) and pose conflicting actions are considered to generate potential violations during run-time.

5.2.3 Policy Reconciliation. As discussed in Section 4.1, the policy mappings (i.e., mapping between the IoT app and the respective graph-based policy) maintained by the VISCR engine helps in effectively enforcing the policies after the conflicts are detected and resolved. The conflict-free composed policy graph along with these mappings are provided as input to the policy reconciliation engine (Figure 1). We develop APIs that effectively leverage this association and reconcile composed policy graph to device-specific IoT apps for enforcement. The policies that are directly provided as graph-based policy input to the VISCR are reconciled to device-specific rules and distributed on to network of IoT devices on the basis of: (*i*) vendor-type, (*ii*) device location, (*iii*) source devices that are generating the triggering events, and (*iv*) destination nodes on which the actions need to be enforced. Upon choosing the right set of devices on which the rules are required to be placed, the reconciliation engine translates it device-specific rules for configuring or programming the specific device.

6 PROTOTYPE & EVALUATION

We developed the complete prototype implementation of VISCR in Python and integrated it with our VISCR policy specification dashboard. We implemented following components in VISCR:

Policies	Smart Building Policy Description
S_1	Any time fire is detected, turn on sprinklers and cameras, and open all locks (doors and windows)
S_2	From 10PM to 7AM keep the outer doors and windows locked
S_3	From 8AM to 9PM set the thermostat to 65°F in bedrooms (kid)
S_4	Between 6PM to 10PM (i.e., till 11PM) keep the main doors unlocked
S_5	If main doors and windows are open for more than 5 minutes, turn OFF the heating/cooling in that room to prevent energy wastage
S_6	When outside temperature is between 60-75°F open the windows and turn OFF cooling
S_7	Turn bedroom II Camera OFF (or no access) after 10PM (kid)
S_8	If building temperature rises above 95°F, lock all windows and reset the thermostat to 65°F
S_9	In case of rain and humidity <40% and >50% close the windows
S_{10}	Keep Camera ON in all rooms and access to it at any time (parent)

Table 3: Example list of smart building policies (S_1-S_{10}).

The abstraction engine of VISCR is integrated with vendor-specific cloud data sources for extracting dynamic states and configurations of IoT devices. We developed data-source drivers for IoT devices which performs to key functions: (*i*) translates the states, configurations, and logs of IoT devices into data tables, and (*ii*) generate datalog rules required for automatically generating tree-based infrastructure abstractions based on the *abstraction mappings* supplied by administrators [24]. The data-push option provided by each of the vendor's cloud data-sources allows VISCR to capture the logs and events specific to dynamic updates in the IoT network.

The graph-based policies specified using the abstraction tree nodes are translated first into equivalent vendor-independent specification syntax and ultimately into a graph dictionary. We used the networkx Python library[25] for capturing and building policy graphs. For visualizing graph-based policies and the composed policy graph, we built the VISCR policy specification dashboard integrated with networkx [25] and GraphViz library [26].

The composition engine captures the policies in networkx-based graphs, runs normalization and composition algorithms to detect the conflicts and resolves them using precedence rules that are maintained as key value pairs. These conflict-free policies are further analyzed for other bugs and violations. Finally, the composed conflict-free policies are translated to enforceable rules i.e., IoT apps and device-specific configurations. The VISCR's policy composition outcome is interfaced with the VISCR specification dashboard for fine-tuning the policies before enforcement.

6.1 Evaluation

Testbed. We used simulated smart building IoT infrastructure with 907 IoT apps or automation policies framed with 30 different types of consumer IoT devices from 8 different vendors framed for multiple floors of the building. For brevity few policies (Table 3) and simulated smart building view of single floor shown in the Figure 8. The VISCR module is run on a Dell R710 server with 48GB RAM, 12 cores (2.6GHz) with Ubuntu 4.4.0-97-generic kernel.

Datasets. We use IoT market apps (i.e., both vetted and unvetted), extracted from vendor marketplaces, and publicly available data sources for the smart-home and smart-campus use cases [27–31], also consolidated in our VISCR repository [32]. We have built following three datasets for our experiments:
DS-1: We simulate a smart building IoT infrastructure (Figure 8) with multiple floors and automate it with 907 IoT apps. We use 907 Groovy-based IoT apps (i.e., homogeneous specifications) for evaluating static analysis-based technique (such as Soteria [9]) and compare with VISCR. The same set of automation rules programmed with Groovy, OpenHAB, IFTTT, and MUD profiles (i.e., heterogeneous specifications) are used for evaluating VISCR.

[5] Two policies are mutually exclusive, when no two events, conditions or states among these policies are not related to each other or can co-exist i.e., occur at the same instance of time.

Policy Conflict	Conflict Description	Conflict Type
S_1, S_2	When fire is detected between 10PM and 7AM by smoke alarm (S_1), all doors and windows are unlocked (open). With S_2 both exterior door and all windows must be locked during this time. This can result in exterior doors and windows toggling from locked or unlocked resulting in unintended behavior.	Static, Loops
S_1, S_9	If temperature raises above 90°F, it will enforce that all windows must be locked and thermostat be set to 65°F (S_9). Conflicts with temperature raised due to fire event (S_1).	Static
S_3, S_4, S_5	Between 7PM and 9PM the outer door and windows are locked, thus trigger for both S_3 and S_5 are valid as a result system will toggle between turning off thermostat and setting it to 65°F. Similarly, S_4 can further intervene due to time overlap and can result in chain and again forming a loop.	Chain, Loop, Gap
S_5, S_6	When temperature outside is 60 – 75°F, S_6 opens the windows, which can possibly trigger S_5 given exterior doors are unlocked too. This policy chaining might result in unintended temperature conditions inside home.	Chain
S_7, S_{10}	Rogue behaviour as S_7 is set by parent and S_{10} set by kid in accessing kid's room Camera after 10PM.	Rogue
S_3, S_8	Gap as condition is not specified for temperature less than 95°F (i.e., between 74°F to 95°F).	Gap, Static
S_4, S_6, S_9	When it is both raining and temperature between 60 – 75°F, conflicting actions arise i.e., undecidable if windows should be opened or closed	Potential Violation

Table 4: Illustrating few types of the conflicts, bugs and violations detected by VISCR for the policies described in Table 3.

DS-2: We use smart-home, smart-campus and smart-city abstractions dataset [28, 30, 33–35] for evaluating the performance of our tree-based abstraction engine in constructing the abstraction trees. **DS-3**: We have built tool to generate ≈20K synthetic policies emulating the dataset DS-1, using the policy abstraction trees generated from DS-2. We use random sampling technique to select the source nodes, destination nodes and edge properties (e.g., dynamic environmental states, conditions, and traffic type) from the policy-abstraction trees.

6.1.1 Policy Abstraction. We evaluate the performance of policy abstraction engine with smart city data of up to 100K devices using DS-2. We built upto 400 policy abstraction trees with four levels of abstractions using the dataset. For generating abstraction trees, the abstraction engine need to join hundreds of data tables performing thousands of table join operations. It took <1.2sec to generate upto 400 abstraction trees with 100K leaf nodes in parallel (Figure 9a). The latency performance of abstraction engine stays mostly linear considering the data join operations it performs to generate the abstraction trees. Since extracting data from the network data sources takes random times considering the network latency, we discard network latency parameter in the calculation of abstract tree generation.

6.1.2 Security Analysis. We evaluate the performance of our conflict detection and resolution engine with DS-1 (i.e., 907 IoT market apps) simulated to program the smart building IoT infrastructure (Figure 8) and compare it with Soteria's violation detection mechanism. Applying model checking, we were able to find 6.6% of the static violations across policies on DS-1, which includes both property and state violations. As shown in Table 5, VISCR on the other hand was able to find ∼37.7% of violations within same IoT apps in DS-1. Importantly, VISCR detected 100% of the violations captured by the static analysis technique such as Soteria. Among the 37.7% of IoT apps VISCR reported 10.2% of IoT apps that has more than one violations. Importantly, different class of violations reported by VISCR is illustrated with examples in Table 4.

Security Analysis	% IoT App Violations	% False Positives
Compile time conflicts	6.6	0
Potential run-time violations	7.9	1.8
Gap analysis	10.4	1.3
Rogue Policies	3.8	0
Access violations	1.6	0
App sanity checker (SC)	4.2	0.7
Loops & chains	3.2	0
Overall	**37.7**	3.8

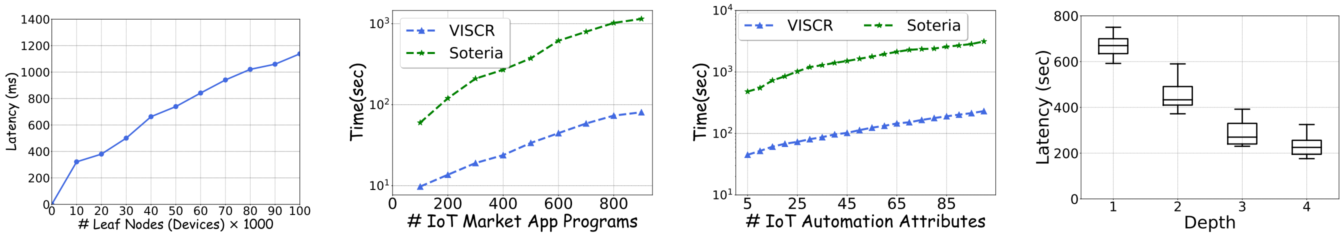
Table 5: Bugs and violations detected on smart building IoT infrastructure dataset DS-1.

In addition, VISCR identifies following major types of conflicts and violations: (*i*) gap in automation due to the inability of users to completely realize the use-case scenario resulted in 10.4% of total IoT apps being violated with 1.3% false positives; (*ii*) code-sanity and programming errors in temporal and spatial policies as well as policies involving specific values (e.g., temperature, humidity, time, space) resulted in 4.2% of violations with 0.7% false positives, where the undefined references are upto 1.1% and unused structures are upto 2.1% of policies; (*iii*) policies that results in access violations due prolonged access to resources beyond the specified period of time counted upto 1.6% of IoT apps; (*iv*) rogue policies that are implemented by administrators who are not authorized to specify policies on portion of IoT infrastructures that they should not be enforcing rules (with 3.8% of total IoT apps violated); (*v*) potential run-time violations that are detected from the policies, which includes 7.9% of total violations; and (*vi*) finally, detected 3.2% of loops among the automation rules that resulted in the unintended and unsafe environmental conditions. Overall, VISCR was able to detect the above discussed violations and bugs with less than 3.8% of false positives or low intensity bugs such as Unused variables or structures and potential violations.

6.1.3 Policy Composition. The composition cost depends on following two factors: (*i*) the number of attributes or states used in the IoT apps, and (*ii*) the number of IoT apps. In this experiment, we perform composition with increasing number of IoT apps (i.e., in subset of 100 apps in each iteration) and capture its composition latency. We keep number of attributes constant at ∼30 in this experiment (i.e., the IoT apps or automation use-cases are chosen only specific to these attributes). For example, temperature is considered as an attribute, while the subcategories (e.g., high, low, different levels) of the attribute are still considered as part of the same temperature attribute. We observed that VISCR took ∼80.7 seconds to compose 907 apps, while using model checking (i.e., such as technique used in Soteria) took approximately 14.2× more time to run, which took ∼1147 seconds to detect the conflicts (Figure 9b).

In the next experiment, we evaluate performance of conflict detection engine with increasing number of attributes and constant number of IoT apps (i.e., 907). With increase in number of attributes, the composition (i.e., conflict detection) cost of both the approaches increased. VISCR took 231 seconds to compose 907 IoT apps and 100 different attributes, while Soteria took ∼3140 seconds, which is ∼13× more time required to detect conflict (Figure 9c).

Following are the vital factors that contribute to improved performance (i.e., reduced conflict detection time with our graph-based composition) compared to Soteria: (*i*) The complex SAT formulation resulting in enumeration of all possible states required to detect the static violations. (*ii*) Our approach optimizes the composition cost by parallelizing the translation procedure (i.e., translation of IoT apps to vendor-independent graph-based specification); (*iii*)



(a) Average latency in building abstraction trees in parallel with increasing # leaf nodes.

(b) Average composition latency with increasing # IoT apps (with ∼30 policy attributes).

(c) Average composition latency with increasing # policy attributes (with 907 IoT apps).

(d) Policy composition latency for ∼20K synthetic policies with abstraction tree depth.

Figure 9: Scalability of VISCR' policy abstraction & composition engine compared to static analysis-based technique (Soteria).

With graph-based composition, we incrementally verify the source nodes and if the overlap exists then further into edge and target properties. This results in avoiding unnecessary comparison operation resulting in improved composition cost with our approach. and (*iv*) Finally, the graph-based composition mechanism generates the *composed graph islands* (i.e., policy graphs that are completely independent), when once a policy is detected as conflicting with one of the policy graph island, the policy is marked as conflicting avoiding comparison with other nodes with in the same composed graph and other graph islands. In addition, to provide fair comparison, we ran VISCR with 907 groovy-based apps, our composition engine took <92.1 seconds to compose these apps.

As VISCR also support incremental composition (i.e., to support dynamic changing IoT infrastructure), we evaluate the performance of the incremental composition as follows. We compose 907 IoT apps and generate the composed policy graph. We then randomly choose 100 IoT apps each time and change its attributes and allow it to incrementally recompose. From our experiments, it is evident that for recomposing 10 IoT app programs took <2.1 seconds, while recomposing 100 apps took ∼16.3 seconds. In normal working conditions of any IoT infrastructure, it is expected to have fewer than 10 IoT app change at any instance of time.

Finally, we evaluate the performance of VISCR with large scale synthetic dataset (DS-3) to emulate the smart city IoT infrastructures with ∼20K graph-based policies. We composed 20K graph-based policies by choosing the source and target nodes of the policies at different levels of abstraction trees i.e., choosing nodes at depth level 1 – level 4 of abstraction tree. We run this experiment for multiple iterations to capture the average composition engine latency. For depth level = 1, the composition cost is much higher than when policy abstracts are chosen at level 4 as the nodes are chosen more towards leaf node level results in much lesser normalization cost. Therefore, 90% of the time our tool took <760 seconds to compose 20K policies specified at level 1. Similarly for composing the policies specified at level 4 VISCR took <300 seconds.

7 RELATED WORK

Intent-based policies that are studied in the context of enterprise networks are limited in flexibility for handling complex and heterogeneous IoT devices [36, 37]. To overcome these limitations in enterprise scenarios, recent works propose the creation of high-level intent-based languages, compilers and conflict detection mechanisms [38–53], and new SDN programming paradigms [54–56]. Prior efforts to develop graph-based policy specification mechanisms (PGA [19], Janus [57], LMS [24]) have focused on enterprise networks. However, these graph-based policy frameworks do not effectively handle dynamic trigger and action-based policies required by IoT devices. Hence, we propose to develop an intuitive graph-based policy framework that handles dynamic trigger and action-based policies and supports vendor-agnostic specification models to seamlessly accommodate different types of IoT programs including Groovy, OpenHAB, IFTTT, and MUD profiles.

Verification and testing of dynamic policies for middleboxes are well-studied problems [58–65]. Unlike our work, prior studies do not address the dynamic group-based policy requirement of IoT infrastructures to handle safety, security, and privacy policies. Recent efforts have attempted to use formal verification techniques and static taint tracking to verify the correctness of deployed automation policies in homogeneous IoT environments [9, 66–69]. Similarly, recently proposed works highlighted the need for novel access control models and policies to secure IoT infrastructures [8, 70]. However, existing IoT infrastructures are dynamic with diverse IoT devices, programmed using heterogeneous programming frameworks, that make static-verification techniques ineffective [9]. Also, a few recent studies that focus on identifying the policy conflicts arising in complex smart-city infrastructures [71, 72] do not deal with security or privacy issues. We propose to build a vendor-independent model, that allows automation rules or policies, specified using multiple commodity IoT apps to be translated into vendor-independent policy-specification graphs for robust and proactive conflict detection and resolution.

8 CONCLUSION

Emerging consumer IoT infrastructures are characterized by a growing number of heterogeneous devices. VISCR provides a unified policy engine that allows for conflict-free policy specification and enforcement in such environments. VISCR achieves this by unifying policy abstractions, automatically extracting IoT infrastructure topology and converting diverse policy languages such as Groovy-based SmartThings, OpenHAB, IFTTT-based templates, and MUD-based profiles into a vendor-independent graph-based specification. These abstractions enable VISCR to detect rogue policies, bugs, and conflicts. They also allow for easier specification and efficient composition of dynamic policy intents, from users and administrators. In a dataset of 907 IoT market apps with a mix of Groovy, OpeHAB, IFTTT, and MUD-based policies, VISCR detected conflicts in 342 apps, provided resolution mechanisms in under 81 seconds, and adapted to new policies with sub-second latency.

9 ACKNOWLEDGMENTS

This work was partially supported by NSF grants CNS-1642965 and CNS-1514503.

REFERENCES

[1] The Future Smart Home: 500 Smart Objects Will Enable New Business Opportunities., March 2014. http://www.gartner.com/newsroom/id/2839717.

[2] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016, November 2017. https://www.gartner.com/newsroom/id/3598917.

[3] OpenHAB Textual Rules, October 2018. https://www.openhab.org/docs/configuration/rules-dsl.html.

[4] Apple's Homekit. Accessed. March 2017. https://developer.apple.com/homekit/.

[5] Samsung SmartThings Public GitHub Repo. Accessed. 2017. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/smartthings/camera-power-scheduler.src/camera-power-scheduler.groovy.

[6] Manufacturer Usage Description (MUD) Specification, October 2018. https://tools.ietf.org/html/draft-ietf-opsawg-mud-25.

[7] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decoupled-IFTTT: Constraining Privilege in Trigger-Action Platforms for the Internet of Things. *CoRR*, abs/1707.00405, 2017.

[8] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (IoT). In *27th USENIX Security Symposium (USENIX Security 18)*, pages 255–272, Baltimore, MD, 2018. USENIX Association.

[9] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated IoT safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 147–158, Boston, MA, 2018. USENIX Association.

[10] Z Berkay Celik, Gang Tan, and Patrick McDaniel. IoTGUARD: Dynamic Enforcement of Security and Safety Policy in Commodity IoT.

[11] OpenHAB: Rules not running. May 2018. https://community.openhab.org/t/solved-rules-not-running/45776.

[12] OpenHAB: Turn light OFF based on timer. March 2017. https://community.openhab.org/t/turn-light-off-based-on-timer/9558/30.

[13] SmartThings Community Discussions. March 2019. https://community.smartthings.com/c/smartapps.

[14] Garadget IFTTT Errors. March 2019. https://community.garadget.com/t/ifttt-errors/4103.

[15] Apple HomePod Communities. March 2019. https://discussions.apple.com/community/homepod.

[16] IFTTT: Samsung SmartThings. Accessed. 2017. https://ifttt.com/smartthings.

[17] The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History, May 2017. https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities/.

[18] ISTR, Internet Security Threat Report, April 2016. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[19] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM.

[20] Nest Cloud APIs. April 2017. https://developers.nest.com/documentation/cloud/get-started.

[21] Samsung Cloud APIs. April 2017. http://developer.samsung.com/smart-home.

[22] Sklearn: LabelBinarizer. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelBinarizer.html.

[23] Using the elbow method to determine the optimal number of clusters for k-means clustering . https://bl.ocks.org/rpgove/0060ff3b656618e9136b.

[24] J. M. Kang, J. Lee, V. Nagendra, and S. Banerjee. LMS: Label Management Service for intent-driven Cloud Management. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 177–185, May 2017.

[25] Networkx Graph Creation. Accessed. December 2017. https://networkx.github.io/documentation/networkx-1.7/tutorial/tutorial.html.

[26] *Graphviz - Graph Visualization Software*. https://www.graphviz.org/.

[27] IoT TestBench: A micro-benchmark suite to assess the effectiveness of tools designed for IoT apps . https://github.com/IoTBench/IoTBench-test-suite.

[28] Sean Barker, Aditya Mishra, David Irwin, Emmanuel Cecchet, Prashant Shenoy, and Jeannie Albrecht. Smart*: An open data set and tools for enabling research in sustainable homes. *SustKDD, August*, 111:112, 2012.

[29] Smart Home Data Set for Sustainability). December 2017. http://traces.cs.umass.edu/index.php/Smart/Smart.

[30] SFO City scale data set (City facilities). December 2017. https://data.sfgov.org.

[31] Charith Perera, Yongrui Qin, Julio C. Estrella, Stephan Reiff-Marganiec, and Athanasios V. Vasilakos. Fog computing for sustainable smart cities: A survey. *ACM Comput. Surv.*, 50(3):32:1–32:43, June 2017.

[32] Secure, Safe and Privacy With IoT Infrastructures Policy Framework and Dataset. Accessed. March 2019. Anonymized for double blinded review.

[33] SFO City scale data set (Locations and Boundaries). December 2017. https://data.sfgov.org/Geographic-Locations-and-Boundaries/List-of-Streets-and-Intersections/pu5n-qu5c.

[34] SFO City scale data set (Traffic Signals). December 2017. https://data.sfgov.org/Transportation/Map-of-Traffic-Signals/8xta-sna8.

[35] SFO City scale data set (City facilities). December 2017. https://data.sfgov.org/City-Infrastructure/Map-of-City-Facilities/bps8-63cu.

[36] John Strassner. *Policy-Based Network Management: Solutions for the Next Generation (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[37] D. C. Verma. Simplifying network administration using policy-based management. *IEEE Network*, 16(2):20–26, Mar 2002.

[38] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 328–341, New York, NY, USA, 2015. ACM.

[39] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[40] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM.

[41] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[42] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM.

[43] C. Trois, M. D. Del Fabro, L. C. E. de Bona, and M. Martinello. A survey on sdn programming languages: Toward a taxonomy. *IEEE Communications Surveys Tutorials*, 18(4):2687–2712, Fourthquarter 2016.

[44] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM.

[45] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM.

[46] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain. An intent-based approach for network virtualization. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 42–50, May 2013.

[47] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical policies for software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 37–42, New York, NY, USA, 2012. ACM.

[48] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 426–439, New York, NY, USA, 2016. ACM.

[49] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A Database-Defined Network. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 5:1–5:7, New York, NY, USA, 2016. ACM.

[50] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 34:1–34:13, New York, NY, USA, 2015. ACM.

[51] OpenDaylight Group Policy. Accessed. 2017. https://wiki.opendaylight.org/view/Group$_p$olicy:Main.

[52] Anu Mercian, Felipe Yrineu, Joon-Myung Kang, Raphael Amorim, Saket M Mahajani, Mario Sanchez and Sujata Banerjee. *Network Intent Composition (NIC) Be Feature Update and Demo: Intent Compilation, Lifecycle Management and Automated Mapping*, Presented in OpenDaylight Summit 2016. http://sched.co/7RBY.

[53] Nanxi Kang, Ori Rottenstreich, Sanjay Rao, and Jennifer Rexford. Alpaca: Compact Network Policies with Attribute-carrying Addresses. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 7:1–7:13, New York, NY, USA, 2015. ACM.

[54] Boulder: Intent based North Bound Interface (NBI). February 2015. https://bit.ly/37w33r9.

[55] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 327–338, New York, NY, USA, 2013. ACM.

[56] M. Pham and D. B. Hoang. SDN applications - The intent-based Northbound Interface realisation for extended applications. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 372–377, June 2016.

[57] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. Supporting Diverse Dynamic Intent-based Policies Using Janus. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 296–309, New York, NY, USA, 2017. ACM.

[58] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. *CoRR*, abs/1512.00822, 2015.

[59] Seyed K. Fayaz and Vyas Sekar. Testing Stateful and Dynamic Data Planes with FlowTest. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 79–84, New York, NY, USA, 2014. ACM.

[60] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing Context-dependent Policies in Stateful Networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 275–289, Berkeley, CA, USA, 2016. USENIX Association.

[61] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 19–24, New York, NY, USA, 2013. ACM.

[62] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J. M. Kang. SFC-Checker: Checking the correct forwarding behavior of Service Function chaining. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 134–140, Nov 2016.

[63] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic Synthesis of NF Models by Program Analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 29–35, New York, NY, USA, 2016. ACM.

[64] Y. Zhang, W. Wu, S. Banerjee, J. M. Kang, and M. A. Sanchez. SLA-verifier: Stateful and quantitative verification for service chaining. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[65] H. Kim and N. Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, February 2013.

[66] Wenbo Ding and Hongxin Hu. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 832–846, New York, NY, USA, 2018. ACM.

[67] Abdullah Al Farooq, Ehab Al-Shaer, Thomas Moyer, and Krishna Kant. IoTC2: A Formal Method Approach for Detecting Conflicts in Large Scale IoT Systems. *CoRR*, abs/1812.03966, 2018.

[68] Emre Göynügür, Sara Bernardini, Geeth de Mel, Kartik Talamadupula, and Murat Şensoy. Policy conflict resolution in IoT via planning. In *Canadian Conference on Artificial Intelligence*, pages 169–175. Springer, 2017.

[69] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1687–1704, Baltimore, MD, 2018. USENIX Association.

[70] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Situational Access Control in the Internet of Things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1056–1073, New York, NY, USA, 2018. ACM.

[71] Meiyi Ma, Sarah Masud Preum, and John A. Stankovic. 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*, IoTDI '17, pages 259–270, New York, NY, USA, 2017. ACM.

[72] M. Ma, S. M. Preum, W. Tarneberg, M. Ahmed, M. Ruiters, and J. Stankovic. Detection of Runtime Conflicts among Services in Smart Cities. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–10, May 2016.