2019 IEEE 32nd Computer Security Foundations Symposium (CSF)

Securing Cross-App Interactions in IoT Platforms

Musard Balliu
KTH Royal Institute of Technology
Stockholm, Sweden
musard@kth.se

Massimo Merro
University of Verona
Verona, Italy
massimo.merro@univr.it

Michele Pasqua
University of Verona
Verona, Italy
michele.pasqua@univr.it

***Abstract*—IoT platforms enable users to connect various smart devices and online services via reactive apps running on the cloud. These apps, often developed by third-parties, perform simple computations on data triggered by external information sources and actuate the results of computation on external information sinks. Recent research shows that unintended or malicious interactions between the different (even benign) apps of a user can cause severe security and safety risks. These works leverage program analysis techniques to build tools for unveiling unexpected interference across apps for specific use cases. Despite these initial efforts, we are still lacking a semantic framework for understanding interactions between IoT apps. The question of what security policy cross-app interference embodies remains largely unexplored.**

This paper proposes a semantic framework capturing the essence of cross-app interactions in IoT platforms. The framework generalizes and connects syntactic enforcement mechanisms to bisimulation-based notions of security, thus providing a baseline for formulating soundness criteria of these enforcement mechanisms. Specifically, we present a calculus that models the behavioral semantics of a system of apps executing concurrently, and use it to define desirable semantic policies in the security and safety context of IoT apps. To demonstrate the usefulness of our framework, we define static mechanisms for enforcing cross-app security and safety, and prove them sound with respect to our semantic conditions. Finally, we leverage real-world apps to validate the practical benefits of our policy framework.

***Index Terms*—Cloud-based IoT platform, IoT app security, cross-app noninterference**

I. Introduction

IoT platforms provide robust application support for automating the interaction and communication between Internet-connected services and smart physical devices. This interaction is enabled by simple reactive programs known as IoT apps (or applets) running on a cloud-based IoT platform, and sensing and actuating data from services and devices on behalf of a user. These apps, often developed by third-parties, are triggered by external information sources, as in "*if the room temperature exceeds a threshold*", to perform actions on external information sinks, as in "*open the windows*". By exposing devices such as a thermostat and a smart window to the IoT platform via, *e.g.*, REST APIs, IoT apps can be used to implement desirable automations like "*if the room temperature exceeds a threshold then open the windows*".

Driven by the appeal of end-user programming, IoT platforms such as IFTTT [24] (If This Then That), Stringify [34], and Microsoft Flow [29] support thousands of smart devices and services with millions of users running millions of IoT apps. These platforms help users to build powerful automations by connecting IoT devices (e.g., smart homes, security cameras, and voice assistants) to online services (*e.g.*, Google and Dropbox) and social networks (*e.g.*, Instagram and Twitter). For instance, the IFTTT platform allows to execute IoT *applets* that include triggers, actions, and filter code. For the platform to run an applet, users need to provide their credentials to the services associated with its triggers and actions. In the previous applet that opens the window when the temperature exceeds a threshold, the user gives the applet access to the APIs for the temperature device (*e.g.*, a Nest Thermostat [32]) and the smart window (*e.g.*, SmartThings [35]). Additionally, applets may contain filter code for personalization, *e.g.*, for setting the temperature threshold. If present, the filter code is invoked after a trigger has been fired and before an action is dispatched.

Recently, researchers have shown that popular IoT platforms are susceptible to attacks that may cause severe security and safety issues for the end users and the physical devices [5]. Examples of attacks include design flaws due to over privileged permission tokens [18], unexpected information leaks by seemingly harmless apps [36], and sensitive information disclosure by malicious apps [6], [11]. To protect the users against these attacks, defensive mechanism rely on fine-grained access control and capabilities, decentralization [19] or static [7], [11] and dynamic [6] information-flow analysis.

A more subtle vulnerability concerns the unintended or malicious interaction between different apps running on behalf on the same user [13]–[15], [17], [36]. The distinctive feature of IoT apps to affect a shared physical environment such as the room temperature, may enable unintended *cross-app interactions* between IoT apps that are installed by the same user. For instance, in addition to the above-mentioned IoT app "*if the room temperature exceeds a threshold then open the windows*", a user may also install the app "*if I leave my work location then turn on the thermostat at home*". While the user's intention is to use these two apps for separate purposes, the interaction between the latter and the former may open the window while the user is not at home, thus clearing a way for burglary.

Recent research identifies numerous use cases of cross-app interactions that violate specific policies, and suggests tracking dependencies across IoT apps to identify policy violations [13]–[15], [17], [36]. These mechanisms perform inter-application program analysis to track dependencies, and (man-

DOI 10.1109/CSF.2019.00029

ual or automatic) language processing to identify semantically-related language constructs, *e.g.*, the fact that *temperature* and *thermostat* refer to related semantic constructs, despite their syntax being different. While these approaches motivate the need for analyzing security and safety risks in cross-app interactions, foundational questions related to the interaction semantics of apps, security policies, and soundness of enforcement mechanisms remain largely unexplored.

This leads us to the following research questions: (i) What is an appropriate formal model for cross-app interaction vulnerabilities? (ii) Is there a generic policy framework for security and safety that captures the essence of cross-app interactions? (iii) How do we model implicit interactions stemming from IoT-specific features like the physical environment? (iv) Can we harden enforcement mechanisms to prove soundness guarantees in our policy framework?

Contributions: To help answering these questions, we develop a process calculus for specifying and reasoning about cross-app interactions, capturing the core features of apps in IoT platforms like IFTTT and Stringify. We then propose extensional conditions to capture the essence of security and safety requirements in a system of IoT apps executing concurrently. We demonstrate the usefulness of these conditions by considering policies from real-world apps, and discuss how they can be relaxed in order to accommodate more flexible user policies. Further, we show how standard enforcement mechanisms can be adapted to check security and safety of a system of IoT apps, thus providing strong guarantees against vulnerable cross-app interactions. We think that these conditions will provide a semantic baseline for proving soundness of current and future enforcement mechanisms in the domain of IoT apps.

Our key observation is that for a system of apps to reach an unsafe configuration, a cross-app interaction should either lead to an inconsistent state that violates the intended specification for some apps, or engage in an interaction where the action of one app triggers the execution of another app. This is supported by the intuition, as well as existing real-world vulnerabilities [13]–[15], [17], [36], that an end user may consider a system of IoT apps as safe if the runtime behavior of an app in isolation is *bisimilar* to running that app in parallel with other apps in the system. Drawing on Focardi and Martinelli's *Generalized Non Deducibility on Composition* [21], we formalize this intuition to provide a bisimulation-based characterization of *safe cross-app interaction*. Further, we provide a simple syntactic condition and prove it sound for our notion of safe cross-app interaction. We also tackle the challenge of implicit cross-app interactions and propose an extension of our semantic condition. Finally, we envision scenarios where some form of interaction across apps can be considered as safe, and show how it can be modeled in our framework via *priorities*.

Further, we focus on confidentiality and integrity policies of a system of IoT apps and propose a *termination-insensitive* bisimulation-based security condition that accommodates these policies. As standard in information-flow control [33], the condition assumes a security classification of global services and devices, and it ensures that any interference between apps respects the security classification. We propose an extension of the flow-sensitive type system by Hunt and Sands [23] for our concurrent IoT setting, and prove it sound for our security condition.

In summary, the paper provides the following contributions:

- We present a calculus for IoT apps to study security and safety in cross-app interactions. The calculus models closely the behavioral semantics of apps in IoT platforms (Section II).
- Inspired by policy requirements in real apps, we propose an extensional condition for safe cross-app interactions, as well as a syntactic condition to enforce safe interactions (Section III).
- We extend our framework to accommodate implicit app interactions and service priorities in order to tackle the challenge of false negatives and false positives, respectively (Section IV).
- We propose a flow-sensitive security types system, enforcing information-flow policies in a system of IoT apps running concurrently (Section V).

Full proofs of our results and a number of examples of IoT apps modeled in our calculus can be found in the Appendix.

II. A Calculus of IoT Apps

In this section, we define our *Calculus of IoT Apps*, called `CaITApp`, to formally specify and reason about *systems of apps*, *i.e.*, sets of concurrent IoT apps running on an IoT platform, and accessing the Internet-connected services and devices of a given user. The interface between the IoT apps and the external services and physical devices, *e.g.*, Dropbox or home security camera, is defined by APIs that enable communication between the platform and the user services and devices. As common in IoT platforms like IFTTT, the platform itself maintains a *global store* with data from a user's services and devices, which gets updated whenever there is a change in the corresponding services and devices. Each IoT app of a given user has its own *local store*, i.e., local view, which may get updated whenever the execution of that app is triggered by a change in the global store.

We start the description of our calculus with some preliminary notations. We use letters $\mathtt{x}, \mathtt{y}, \mathtt{z} \in \mathsf{Service}$ to denote the IoT platform's (global) view of a user's services and devices. Abusing notation, we call them just services in the following. *Values*, ranged over by $v, w \in \mathsf{Value}$ are basic values, such as booleans, integers, real numbers, strings, etc. We assume two special values: $\bot$ and $*$. The first represents an undefined value, while the second is a placeholder that can be replaced with "any value".

The *syntax* of our *systems* is given by the grammar:

$$\begin{array}{lll} \mathsf{Sys} \ni \mathsf{S} & ::= & \mathsf{S} \parallel \mathsf{S} \qquad \text{∫ parallel composition} \\ & \mid & \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,] \qquad \text{∫ app} \end{array}$$

Here, $\mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ denotes an app with a *unique identifier* $\mathtt{id} \in \mathcal{I}$, using only the global services declared in D, with the associated permissions (read and/or write), and running the process (code) P.

The syntax for service declarations is the following:

$$\begin{array}{lcll} \mathsf{Decl} \ni \mathsf{D} & ::= & \mathsf{D};\mathsf{D} & \int \text{ declaration list} \\ & | & \mathtt{x}^{\mathsf{R}} & \int \text{ service to be used in read} \\ & | & \mathtt{x}^{\mathsf{W}} & \int \text{ service to be used in write} \end{array}$$

In the following, we will write $\mathtt{x}^{\mathsf{RW}}$, as an shorthand for $\mathtt{x}^{\mathsf{R}};\mathtt{x}^{\mathsf{W}}$.

The syntax of our *processes* for describing the code running in our IoT apps is the following:

$$\begin{array}{lcll} \mathsf{Proc} \ni \mathsf{P} & ::= & \mathsf{listen}(\mathsf{L}) & \int \text{ listener} \\ & | & \mathtt{x} \leftarrow \mathsf{e} & \int \text{ set local store} \\ & | & \mathsf{update}(\mathtt{x}) & \int \text{ set global store} \\ & | & \mathsf{if\ b\ then\ } \{\mathsf{P}\} \mathsf{\ else\ } \{\mathsf{P}\} & \int \text{ conditional} \\ & | & \mathsf{skip} & \int \text{ termination} \\ & | & \mathbb{X} & \int \text{ process variable} \\ & | & \mathsf{fix}\,\mathbb{X} \bullet \mathsf{P} & \int \text{ recursion} \\ & | & \mathsf{P};\mathsf{P} & \int \text{ seq. composition} \end{array}$$

Let us comment on the most peculiar constructs. With $\mathsf{listen}(\mathsf{L})$ an app listens on a set of services L whose changes may trigger the app to execute. This is a blocking construct as it progresses only when at least one of the services listed in L changes. L is formally defined as follows:

$$\begin{array}{lcll} \mathsf{VarList} \ni \mathsf{L} & ::= & \mathsf{L};\mathsf{L} & \int \text{ services list} \\ & | & \mathtt{x} & \int \text{ service} \end{array}$$

The construct $\mathtt{x} \leftarrow \mathsf{e}$ sets the local variable $\mathtt{x}$ (the local view of the global service $\mathtt{x}$) with the value obtained by the evaluation of an expression e. Note that, in the expression e we may have both readings on local variables $\mathtt{y}$, simply denoted with $\mathtt{y}$, and readings of global variables $\mathtt{y}$, denoted with $\mathsf{read}(\mathtt{y})$. Thus, in the assignment $\mathtt{x} \leftarrow \mathsf{read}(\mathtt{x}) + \mathtt{y}$ the local copy of the service variable $\mathtt{x}$ is updated with the summation between the up-to-date value of the global service $\mathtt{x}$ (taken from the cloud) and the value read from the local copy of the service $\mathtt{y}$. The construct $\mathsf{update}(\mathtt{x})$ updates the value of the service $\mathtt{x}$ in the global store with its current value in the local store. The process $\mathsf{fix}\,\mathbb{X} \bullet \mathsf{P}$ is the standard construct to denote recursion.

An app is a process silently running in background until a trigger occurs. This latter fires the app payload, consisting of a sequence of actions (potentially dispatched after the execution of some code). Technically speaking, the process running in an app is a recursive process of the form: $\mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathsf{L});\mathtt{payload}$. Intuitively, our apps keep listening on a number of cloud services: when at least one these services changes, the app executes its payload. The payload consists in performing a number of activities, such as checking the state of some cloud service $\mathtt{x}$ via the $\mathsf{read}(\mathtt{x})$ expression, and updating one or more cloud services via the $\mathsf{update}(\mathtt{x})$ construct.

Actually, the syntax proposed for the code of our apps is a bit too permissive with respect to our intentions. We could rule out ill-formed apps with a simple type systems. However, for the sake of simplicity, we prefer to provide the following definition.

Definition 1 (Well-Formedness)**.** An app $\mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ is *well-formed* if the following conditions are satisfied:

- P is of the form $\mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathsf{L});\mathsf{Q}$;
- $\mathtt{x}$ appears in $\mathsf{listen}(\mathsf{L})$ only if $\mathtt{x}^{\mathsf{R}}$ occurs in D;
- the payload Q does not contain listeners;
- $\mathsf{read}(\mathtt{x})$ appears in Q only if $\mathtt{x}^{\mathsf{R}}$ occurs in D;
- $\mathsf{update}(\mathtt{x})$ appears in Q only if $\mathtt{x}^{\mathsf{W}}$ occurs in D.

A system is *well-formed* only if its apps are well-formed.

Hereafter, we will always work with well-formed systems.

Let us provide two simple examples to describe how we can model IoT apps in `CaITApp`.

Example 1. Consider the following two apps. `Tw2Fb` reposts on Facebook messages received on Twitter. Similarly, when there is a new post on Facebook, the app `Fb2Ld` publishes the post on LinkedIn. In this case, we have three logical services: `tw`, for Twitter, `fb`, for Facebook and `ld`, for LinkedIn. The apps are formalized in our language as follows:

$$\mathtt{Tw2Fb}[\,\mathtt{tw}^{\mathsf{R}};\mathtt{fb}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{tw});\mathsf{pld1}\,]$$

where $\mathsf{pld1} \stackrel{\mathsf{def}}{=} \mathtt{tw} \leftarrow \mathsf{read}(\mathtt{tw});\mathtt{fb} \leftarrow \mathtt{tw};\mathsf{update}(\mathtt{fb});\mathbb{X}$, and

$$\mathtt{Fb2Ld}[\,\mathtt{fb}^{\mathsf{R}};\mathtt{ld}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{fb});\mathsf{pld2}\,]$$

where $\mathsf{pld2} \stackrel{\mathsf{def}}{=} \mathtt{ld} \leftarrow \mathsf{read}(\mathtt{fb});\mathtt{ld} \leftarrow \mathtt{fb};\mathsf{update}(\mathtt{ld});\mathbb{X}$.

Example 2. Consider the following two apps. When smoke is detected, the app `SmokeAlarm` should fire the smoke alarm and turn on the lights. If a given heat threshold is reached, then the app `Sprinks` will open the water valve to activate fire sprinkles. For that we assume five logical services: `smoke`, reporting the presence of smoke, `heat`, reporting the heat level, `waterV`, controlling the water valve, `alarm`, controlling the smoke alarm, and `lights`, managing the lights. The apps are formalized in our language as

$$\mathtt{SmokeAlarm}[\,\mathtt{smoke}^{\mathsf{R}};\mathtt{alarm}^{\mathsf{W}};\mathtt{lights}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{smoke});\mathsf{P3}\,]$$

where:

$$\begin{array}{l} \mathsf{P3} \stackrel{\mathsf{def}}{=} \mathtt{smoke} \leftarrow \mathsf{read}(\mathtt{smoke}); \\ \quad \mathsf{if}\ (\mathtt{smoke} = \mathit{yes})\ \mathsf{then}\ \{ \\ \quad\quad \mathtt{alarm} \leftarrow \mathit{On}; \\ \quad\quad \mathtt{lights} \leftarrow \mathit{On}; \\ \quad\quad \mathsf{update}(\mathtt{alarm}, \mathtt{lights}) \\ \quad \};\mathbb{X} \end{array}$$

and $\mathtt{Sprinks}[\,\mathtt{heat}^{\mathsf{R}};\mathtt{waterV}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{heat});\mathsf{P4}\,]$, where:

$$\begin{array}{l} \mathsf{P4} \stackrel{\mathsf{def}}{=} \mathtt{heat} \leftarrow \mathsf{read}(\mathtt{heat}); \\ \quad \mathsf{if}\ (\mathtt{heat} \geq 45)\ \mathsf{then}\ \{ \\ \quad\quad \mathtt{waterV} \leftarrow \mathit{Open}; \mathsf{update}(\mathtt{waterValve}) \\ \quad \};\mathbb{X} \end{array}$$

$$(\text{StopListening})\ \frac{\mathsf{L} = \mathtt{x}_1; \ldots; \mathtt{x}_n \qquad \exists i \in [1,n]\,.\,\mathfrak{G}(\mathtt{x}_i) \neq \phi(\mathtt{x}_i)}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{listen}(\mathsf{L}) \xrightarrow{\tau} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{skip}}$$

$$(\text{SetLocal})\ \frac{[\![\mathsf{e}]\!](\mathfrak{G}, \phi) = v}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathtt{x} \leftarrow \mathsf{e} \xrightarrow{\tau} \langle \mathfrak{G}, \phi[\mathtt{x} \hookleftarrow v] \rangle \triangleright \mathsf{skip}}$$

$$(\text{SkipUpdate})\ \frac{\mathfrak{G}(\mathtt{x}) = \phi(\mathtt{x})}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{update}(\mathtt{x}) \xrightarrow{\tau} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{skip}}$$

$$(\text{Update})\ \frac{\mathfrak{G}(\mathtt{x}) \neq \phi(\mathtt{x}) \qquad \phi(\mathtt{x}) = v}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{update}(\mathtt{x}) \xrightarrow{\mathtt{x}!v} \langle \mathfrak{G}[\mathtt{x} \hookleftarrow v], \phi \rangle \triangleright \mathsf{skip}}$$

$$(\text{IfTrue})\ \frac{[\![\mathsf{b}]\!](\mathfrak{G}, \phi) = \mathtt{tt}}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{if\ b\ then\ } \{\mathsf{P}_1\} \mathsf{\ else\ } \{\mathsf{P}_2\} \xrightarrow{\tau} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_1}$$

$$(\text{IfFalse})\ \frac{[\![\mathsf{b}]\!](\mathfrak{G}, \phi) = \mathtt{ff}}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{if\ b\ then\ } \{\mathsf{P}_1\} \mathsf{\ else\ } \{\mathsf{P}_2\} \xrightarrow{\tau} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_2}$$

$$(\text{Fix})\ \frac{-}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{fix}\,\mathbb{X} \bullet \mathsf{P} \xrightarrow{\tau} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}\left\{^{\mathsf{fix}\,\mathbb{X} \bullet \mathsf{P}}/_{\mathbb{X}}\right\}}$$

$$(\text{SeqSkip})\ \frac{-}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{skip}; \mathsf{P} \xrightarrow{\tau} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}}$$

$$(\text{Seq})\ \frac{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}', \phi' \rangle \triangleright \mathsf{P}_1'}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_1; \mathsf{P}_2 \xrightarrow{\lambda} \langle \mathfrak{G}', \phi' \rangle \triangleright \mathsf{P}_1'; \mathsf{P}_2}$$

TABLE I
LABELED TRANSITION SEMANTICS FOR PROCESSES

We end this section with a couple of *notations and abbreviations* that will be used in the rest of the paper. We will write $\mathsf{update}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$ as an abbreviation for the sequential update of the global variables $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n$, namely: $\mathsf{update}(\mathtt{x}_1); \mathsf{update}(\mathtt{x}_2); \ldots; \mathsf{update}(\mathtt{x}_n)$. We write $\mathsf{if\ b\ then\ } \{\mathsf{P}\}$ as an abbreviation for $\mathsf{if\ b\ then\ } \{\mathsf{P}\} \mathsf{\ else\ } \{\mathsf{skip}\}$.

A. Labeled Transition Semantics

IoT Apps are simple applications interacting with physical and logical services that can be accessed only via a *cloud platform*, that we call *global store*, denoted with $\mathfrak{G} \in \mathbb{S}$, where $\mathbb{S} \stackrel{\text{def}}{=} \mathsf{Service} \rightharpoonup \mathsf{Value}$ is the set of all partial functions from services to values. Every app $\mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ retains a *local view* of the cloud platform that must be consistent with the app's declaration D, meaning that the domain of the *local store* of id consists of all and only those services declared in D. Changes in the global store are shared with all parallel apps of the system associated to the same user/account; however, these modifications do not directly affect the apps' local view of the store. Indeed, a local store can be modified only explicitly by its related app payload.

Since our syntax distinguishes between processes and systems of apps, in our labeled transition semantics we have two different kinds of transitions: one for processes and a second one for systems.

$$(\text{App})\ \frac{\mathfrak{L}(\mathtt{id}) = \phi \quad \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P} \xrightarrow{\tau} \langle \mathfrak{G}, \phi' \rangle \triangleright \mathsf{P}'}{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,] \xrightarrow{\tau} \langle \mathfrak{G}, \mathfrak{L}[\mathtt{id} \hookleftarrow \phi'] \rangle \triangleright \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}'\,]}$$

$$(\text{AppUpdate})\ \frac{\mathfrak{L}(\mathtt{id}) = \phi \quad \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P} \xrightarrow{\mathtt{x}!v} \langle \mathfrak{G}', \phi \rangle \triangleright \mathsf{P}'}{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,] \xrightarrow{\mathtt{id}:\mathtt{x}!v} \langle \mathfrak{G}', \mathfrak{L} \rangle \triangleright \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}'\,]}$$

$$(\text{EnvChange})\ \frac{-}{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S} \xrightarrow{\mathtt{x}?v} \langle \mathfrak{G}[\mathtt{x} \hookleftarrow v], \mathfrak{L} \rangle \triangleright \mathsf{S}}$$

$$(\text{ParLeft})\ \frac{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S}_1 \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{S}_1' \qquad \alpha \in \{\tau, \mathtt{id}:\mathtt{x}!v\}}{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{S}_1' \parallel \mathsf{S}_2}$$

$$(\text{ParRight})\ \frac{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S}_2 \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{S}_2' \qquad \alpha \in \{\tau, \mathtt{id}:\mathtt{x}!v\}}{\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{S}_1 \parallel \mathsf{S}_2'}$$

TABLE II
LABELED TRANSITION SEMANTICS FOR SYSTEMS

In Table I we provide the transition rules for *process configurations* of the form

$$\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P} \xrightarrow{\lambda} \langle \mathfrak{G}', \phi' \rangle \triangleright \mathsf{P}'\,,$$

where $\mathfrak{G} \in \mathbb{S}$ denotes the global store while $\phi \in \mathbb{S}$ is the local store in which the process P is running. The labels $\lambda \in \{\tau, \mathtt{x}!v\}$ denote: *nonobservable* actions and *observable* modifications (writings) of a global service x, respectively. We assume a standard *evaluation semantics* for expressions $[\![\mathsf{e}]\!] \in \mathbb{S} \times \mathbb{S} \to \mathsf{Value}$, inductively defined on the structure of the expression e. We omit the details of its definition. Here, it is only important to notice that we have one expression for looking-up variables in the local store, $[\![\mathtt{x}]\!](\mathfrak{G}, \phi) \stackrel{\text{def}}{=} \phi(\mathtt{x})$, and a different expression for looking-up variables in the global store: $[\![\mathsf{read}(\mathtt{x})]\!](\mathfrak{G}, \phi) \stackrel{\text{def}}{=} \mathfrak{G}(\mathtt{x})$.

Now, let us comment on the most interesting rules of Table I. The construct $\mathsf{listen}(\mathsf{L})$ is a blocking operator waiting for changes in the cloud on (at least on of) those services contained in L. The semantics of the $\mathsf{listen}(\mathsf{L})$ operator is formalized by means of the rule (StopListening). The transition rule (SetLocal) serves to update the local store via an assignment to (the local copy of the) service x; this assignment will affect the global service x only if followed by an $\mathsf{update}(\mathtt{x})$. By an application of the rule (Update) the construct $\mathsf{update}(\mathtt{x})$ modifies the value associated to the service x in the global store with the value recorded in the local store. The remaining rules are standard.

In Table II we provide the transition rules for *system configurations* of the form

$$\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S} \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{S}'\,,$$

where $\mathfrak{G} \in \mathbb{S}$ denotes the global store, whereas $\mathfrak{L} \in \mathcal{I} \to \mathbb{S}$ is a map associating to any app identifier its local store. The labels $\alpha \in \{\tau, \mathtt{id}:\mathtt{x}!v, \mathtt{x}?v\}$ denote: *nonobservable* actions, *observable* modifications (writings) made by the applet id on a global service x, and *observable* changes on a global

service $\mathtt{x}$ made by the external environment, respectively. In the following, we will write $\mathfrak{C} \xrightarrow{\alpha} \mathfrak{C}'$ to denote a transition between system configurations belonging to the set Sconf of all possible system configurations.

Now, let us comment on the transition rules of Table II. Both rules (App) and $(\mathrm{AppUpdate})$ serve to lift actions from processes to apps (and hence systems of apps). Observe that in case of updates on the cloud services, we are interested in annotating the label of the action with the name of the app performing the write operation; this will be useful when defining *safe cross-app interactions* (Definition 3). The rule $(\mathrm{EnvChange})$ models changes in the cloud made by the external environment and affecting all apps. Thus, this action is not triggered by some app of the system and it can be fired nondeterministically at any moment. Rules $(\mathrm{ParLeft})$ and $(\mathrm{ParRight})$ are standard. Note that, for convenience, action $\mathtt{x}?v$ is allowed only to complete systems as it does not propagate through parallel composition (it is not admitted in rules $(\mathrm{ParLeft})$ and $(\mathrm{ParRight})$).

III. Defining Safe Cross-App Interactions

In order to capture harmful interactions in systems of apps, we formalize a notion of *safe cross-app interaction* based on a bisimulation-based behavioral semantics for our systems.

Intuitively, two apps may interact with each other by acting on some common services in such a way that the state reached by those services is somehow inconsistent (think of a thermostat or a valve when activated by different apps designed with different specifications in mind). However, this is not the only way to yield unsafe interactions between two apps: an app A might interact with a second app B if the execution of some actions of A may affect services in the cloud whose changes may subsequently trigger activities of B. Those activities of B would not have been fired if A would not have modified the state of its services on the cloud.

A. Semantic characterization of safe cross-app interactions

In this section, we provide a semantic characterization of safe cross-app interaction based on some appropriate notion of bisimulation.

Intuitively, we would like to say that a system of apps S does not interact with a system R if the runtime behavior of R when running in parallel with S does not differ from its behavior when running in isolation. A bit more formally, along the lines of Focardi and Martinelli's *Generalized Non Deducibility on Composition (GNDC)* [21], we would like to say that a system S does not interact with a system R if

$$\mathsf{S} \parallel \mathsf{R} \simeq_{\mathsf{S}} \mathsf{R}$$

for some appropriate bisimilarity $\simeq_{\mathsf{S}}$ that hides those (observable) actions of S that modify the services in the cloud (the global store). Notice that the bisimilarity $\simeq_{\mathsf{S}}$ should only suppress the capability of the observer to detect writing actions on the cloud services executed by S; however, these writings must be executed, so that indirect interactions via the cloud between the two systems can be observed if they trigger a nongenuine behavior in R.

Basically, in the scenario above, if bisimilarity breaks then the system S does interact with the correct execution of R in at least one of the following ways:

- The compound system $\mathsf{S} \parallel \mathsf{R}$ might have nongenuine traces containing observables (originating from the R component) that cannot be reproduced by R in isolation; here the interaction affects the *integrity* of the behavior of R.
- The system R might have execution traces containing observables that cannot be reproduced by the compound system $\mathsf{S} \parallel \mathsf{R}$ because they are prevented S; this is a violation of the *availability* of the system R.

In order to formalize the concepts described above, we define a slight generalization of the *weak asynchronous bisimulation* [3] introduced for the asynchronous fragment of the π-calculus [22], [28]. In that bisimulation, input actions are made not observable because in an asynchronous setting the observer cannot directly observe them. Here, we intend to hide modifications on cloud enabled by the interacting system.

Consider a set H of *hidden actions*, $H \subset \mathcal{A} \setminus \{\tau\}$, Then, the following bisimulation compares two system configurations by observing all possible actions except those occurring in H.

Definition 2 (Hiding Bisimulation)**.** Given a set of actions $H \subseteq \mathcal{A} \setminus \{\tau\}$, the symmetric relation $\mathcal{R} \subseteq \mathsf{Sconf} \times \mathsf{Sconf}$ is a *hiding bisimulation* parametric on H if and only if, whenever $\mathfrak{C}_1 \mathcal{R} \mathfrak{C}_2$ and $\mathfrak{C}_1 \xrightarrow{\alpha}\!\!\triangleright \mathfrak{C}_1'$ we have the following:

- if $\alpha \notin H$ then $\mathfrak{C}_2 \stackrel{\hat{\alpha}}{\Rightarrow} \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \mathcal{R} \mathfrak{C}_2'$;
- if $\alpha \in H$ then
  - either $\mathfrak{C}_2 \stackrel{\hat{\alpha}}{\Rightarrow} \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \mathcal{R} \mathfrak{C}_2'$
  - or $\mathfrak{C}_2 \stackrel{\tau}{\rightarrow\!\!\triangleright}{}^{*} \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \mathcal{R} \mathfrak{C}_2'$.

We say that two system configurations $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are *hiding bisimilar w.r.t.* the set of actions H, written $\mathfrak{C}_1 \approx_H \mathfrak{C}_2$, if $\mathfrak{C}_1 \mathcal{R} \mathfrak{C}_2$ for some hiding bisimulation $\mathcal{R}$ parametric on H.

Obviously, for $H = \varnothing$ we get the standard bisimulation.

In the following, for the sake of simplicity, given two system configurations $\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S}$ and $\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{R}$, we will write

$$\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S} \approx_H \mathsf{R}$$

as an abbreviation for $\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S} \approx_H \langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{R}$.

Now, we can use our hiding bisimilarity to formalize a *semantic-based* notion of safe cross-app interaction. As said before, out intention is to hide only those actions that may cause an update on the cloud. Thus, given an arbitrary system $\mathsf{S} \stackrel{\text{def}}{=} \prod_{i=1}^{n} \mathtt{id}_i[\,\mathsf{D}_i \bowtie \mathsf{P}_i\,]$, we define the set of possible actions of S that may modify the state of the cloud services:

$$\mathfrak{upd}(\mathsf{S}) \stackrel{\text{def}}{=} \bigcup_{1 \le i \le n} \{\mathtt{id}_i{:}\mathtt{x}!v \mid \mathtt{x}^{\mathsf{W}} \in \mathsf{D}_i\}\,.$$

In the following, we let $\mathfrak{L}_\perp$ be the function $\lambda \mathtt{id}\,.\,\lambda \mathtt{x}\,.\,\perp$ used to define an initial local environments for all apps in which all services are not initialized.

Definition 3 (Safe cross-app interaction)**.** Let S and R be two systems of apps in CaITApp. We say that S is *noninteracting* with R when

$$\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathsf{S}} \mathsf{R}\,,$$

for $H_\mathsf{S} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{S})$. We say that the two systems S and R *do not interact with each other* if in addition to the previous requirement it holds that

$$\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathsf{R}} \mathsf{S}\,,$$

for $H_\mathsf{R} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{R})$.

Example 3. Consider the two apps Tw2Fb and Fb2Ld introduced in Example 1. Here, we have a potentially unwanted interaction between the two apps as Tw2Fb may trigger Fb2Ld: a message of Twitter will be posted on Facebook by Tw2Fb, then the app Fb2Ld will post that message on LinkedIn. In fact, according to Definition 3, the app Tw2Fb may interact with the app Fb2Ld as:

$$\exists \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{Tw2Fb} \parallel \texttt{Fb2Ld} \not\approx_H \texttt{Fb2Ld}$$

for $H = \{\texttt{Tw2Fb:tw!}v \mid v \in \mathsf{Value}\}$. It is easy to see that on the left-hand-side compound system the app Fb2Ld might do a writing on the cloud service LinkedIn that would not occur if the app was running in isolation.

Example 4. Consider the two apps SmokeAlarm and Sprinks introduced in Example 2. According to Definition 3, the two apps do not interact with each other, as for any $\mathfrak{G} \in \mathbb{S}$ we have the following:

- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{SmokeAlarm} \parallel \texttt{Sprinks} \approx_{H_1} \texttt{SmokeAlarm}$, with $H_1 \stackrel{\text{def}}{=} \{\texttt{Sprink:waterValve!}v \mid v \in \mathsf{Value}\}$;
- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{SmokeAlarm} \parallel \texttt{Sprinks} \approx_{H_2} \texttt{Sprinks}$, with $H_2 \stackrel{\text{def}}{=} \{\texttt{SmokeAlarm:alarm!}v, \texttt{SmokeAlarm:lights!}v\}$, for any $v \in \mathsf{Value}$.

In the example above the two apps do not interact with each other simply because they work on different services. However, according to Definition 3, two apps may be non-interacting even if they write on the same services, provided that no causalities exist among the two writings.

Example 5. Suppose to have an app SimPres that simulates the presence of the user when it is not at home during the night, turning on lights for 10 minutes every half an hour. Then, consider a second app eSaver turning off lights during the day to save energy whenever there is no motion for at least 5 minutes. The two apps are defined as follows:

$$\texttt{SimPres}[\,\texttt{user}^\mathsf{R};\texttt{time}^\mathsf{R};\texttt{lights}^\mathsf{W} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\texttt{user};\texttt{time});\mathsf{P}_5\,],$$

where:

$$\begin{array}{l}\mathsf{P}_5 \stackrel{\text{def}}{=} \texttt{user} \leftarrow \mathsf{read}(\texttt{user});\\ \quad \mathsf{if}\ (0 < \mathsf{read}(\texttt{time.H}) < 7 \wedge \texttt{user} = \mathit{away})\ \mathsf{then}\ \{\\ \quad\quad \mathsf{if}\ \mathsf{read}(\texttt{time.M}) = 30\ \mathsf{then}\ \{\\ \quad\quad\quad \texttt{lights} \leftarrow \mathit{On10minsOff}; \mathsf{update}(\texttt{lights})\\ \quad\quad \}\\ \quad \};\mathbb{X}\end{array}$$

$$\texttt{eSaver}[\,\texttt{none}^\mathsf{R};\texttt{time}^\mathsf{R};\texttt{lights}^\mathsf{RW} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\texttt{none};\texttt{lights});\mathsf{P}_6\,],$$

where:

$$\begin{array}{l}\mathsf{P}_6 \stackrel{\text{def}}{=} \texttt{none} \leftarrow \mathsf{read}(\texttt{none});\\ \quad \mathsf{if}\ (8 < \mathsf{read}(\texttt{time.H}) < 18)\ \mathsf{then}\ \{\\ \quad\quad \mathsf{if}\ (\texttt{none} \geq \mathit{5mins} \wedge \texttt{lights} = \mathit{On})\ \mathsf{then}\ \{\\ \quad\quad\quad \texttt{lights} \leftarrow \mathit{Off}; \mathsf{update}(\texttt{lights})\\ \quad\quad \}\\ \quad \};\mathbb{X}\end{array}$$

Notice that that there is no interaction between these two apps, even if they write on the same global service lights. Actually, those writings occur in different periods of the day and can never interact. Thus, according to Definition 3, for any $\mathfrak{G} \in \mathbb{S}$ we have the following:

- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{SimPres} \parallel \texttt{eSaver} \approx_{H_1} \texttt{eSaver}$, with $H_1 \stackrel{\text{def}}{=} \{\texttt{SimPres:lights!}v \mid v \in \mathsf{Value}\}$;
- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{SimPres} \parallel \texttt{eSaver} \approx_{H_2} \texttt{SimPres}$, with $H_2 \stackrel{\text{def}}{=} \{\texttt{eSaver:lights!}v \mid v \in \mathsf{Value}\}$.

B. A proof technique for safe cross-app interaction

Although the notion of safe cross-app interaction in Definition 3 is very intuitive, it is actually quite hard to verify due to the universal quantification over all possible global stores.

In this section, we provide *syntactic conditions*, easy to verify, that allow us to enforce the semantic condition of safe cross-app interaction. In order to do that, we have to formally specify: (i) what are the potential actions that an app may perform, (ii) what are the services whose changes may trigger an app.

In our calculus CaITApp, the actions potentially performed by an app $\texttt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ are given by the services declared in write modality.

Definition 4 (Actions)**.** Given an app $\texttt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$, we define $\mathfrak{act}(\texttt{id}) \stackrel{\text{def}}{=} \{\mathtt{x} \in \mathsf{Service} \mid \mathtt{x}^\mathsf{W} \in \mathsf{D}\}$. More generally, in a system of apps $\mathsf{S} \stackrel{\text{def}}{=} \prod_{i=1}^{n} \texttt{id}_i[\,\mathsf{D}_i \bowtie \mathsf{P}_i\,]$ we define $\mathfrak{act}(\mathsf{S}) \stackrel{\text{def}}{=} \bigcup_{1 \leq i \leq n} \mathfrak{act}(\texttt{id}_i)$.

Similarly, the triggers of an app $\texttt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ are given by the services on which the app currently listen or make a read from the global store, namely the services declared in read modality.

Definition 5 (Triggers)**.** Given an app $\texttt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$, we define $\mathfrak{trg}(\texttt{id}) \stackrel{\text{def}}{=} \{\mathtt{x} \in \mathsf{Service} \mid \mathtt{x}^\mathsf{R} \in \mathsf{D}\}$. More generally, in a system of apps $\mathsf{S} \stackrel{\text{def}}{=} \prod_{i=1}^{n} \texttt{id}_i[\,\mathsf{D}_i \bowtie \mathsf{P}_i\,]$ we define $\mathfrak{trg}(\mathsf{S}) \stackrel{\text{def}}{=} \bigcup_{1 \leq i \leq n} \mathfrak{trg}(\texttt{id}_i)$.

Now, everything is in place to provide a syntactic condition for safe cross-app interaction, where a system S is said not to interact with a system R when:

- the two systems do not write on common cloud services;
- the execution of S may not trigger any app of R.

Formally,

Definition 6 (Syntax-based safe cross-app interaction)**.** The system S is said to be *syntactically noninteracting* with the

system R, written $\mathsf{S} \nrightarrow \mathsf{R}$, when both the following conditions hold:

1) $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{act}(\mathsf{R}) = \varnothing$;
2) $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{trg}(\mathsf{R}) = \varnothing$.

More generally, we say that the two systems S and R are *syntactically noninterfering w.r.t. each other*, written $\mathsf{S} \nleftrightarrow \mathsf{R}$, when besides the two conditions above it also holds:

3) $\mathfrak{trg}(\mathsf{S}) \cap \mathfrak{act}(\mathsf{R}) = \varnothing$.

Now, if we consider the apps in Examples 1, 2 and 5, it is easy to verify that:

- $\texttt{Fb2Ld} \nrightarrow \texttt{Tw2Fb}$ holds;
- $\texttt{Tw2Fb} \nrightarrow \texttt{Fb2Ld}$ does not hold because $\mathfrak{act}(\texttt{Tw2Fb}) \cap \mathfrak{trg}(\texttt{Fb2Ld}) \neq \emptyset$;
- $\texttt{SmokeAlarm} \nleftrightarrow \texttt{Sprinks}$ holds;
- $\texttt{SimPres} \nleftrightarrow \texttt{eSaver}$ does not hold because they can both modify the service `lights`, that is $\mathfrak{act}(\texttt{SimPres}) \cap \mathfrak{act}(\texttt{eSaver}) = \{\texttt{lights}\} \neq \emptyset$.

Thus, Definition 6 provides an easy-to-verify syntactic sufficient condition for semantic-based condition.

Theorem 1 (Soundness)**.** *Let* S *and* R *be two systems of apps in* `CaITApp`*. Let* $H_{\mathsf{S}} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{S})$ *and* $H_{\mathsf{R}} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{R})$*. Then:*

- $\mathsf{S} \nrightarrow \mathsf{R}$ *implies* $\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_{\mathsf{S}}} \mathsf{R}$*;*
- $\mathsf{S} \nleftrightarrow \mathsf{R}$ *implies*
  - $\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_{\mathsf{S}}} \mathsf{R}$
  - $\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_{\mathsf{R}}} \mathsf{S}$.

The details of the proof can be found in the Appendix.

Obviously, Definition 6 provides us with a sufficient but not necessary condition to derive soundness for cross-app interactions, as shown, for instance, by the two apps `SimPres` and `eSaver` in Example 5.

IV. Implicit Interactions and Priorities

In this section, we study: (i) the challenge posed by *implicit cross-app interactions*, and (ii) the possibility of having *priorities* between different services.

The former arises whenever two services, *e.g.*, temperature and thermostat, are semantically related, though they differ syntactically. This may lead to both the semantic condition and the enforcement mechanism deeming an interaction as safe, while this is not the case in practice. We propose an extension of our language semantics, as well as a semantic condition and a syntactic one to reason about such cases.

The latter may be useful when our semantic condition in Definition 3 becomes too restrictive for use cases where the end user is willing to accept some interactions on specific services, *e.g.*, social networks like Facebook, while avoiding interactions on other services, e.g, the temperature. By leveraging a lattice order of priorities between services, we discuss how our condition can be extended to enable such flexibility.

A. Countering implicit interactions

The semantic-based condition given in Definition 3 works quite well when dealing with logical services like Facebook or Twitter as in Example 1. However, when stepping to physical services, *i.e.*, services affecting the physical environment, such as the temperature of a room, we may end up accepting as safe a system of apps in the presence of some kind *implicit interactions*. Consider the example below.

Example 6. Let `Win` be an app managing the window of a room, depending on the temperature detected: when the temperature is above 22 degrees then the window must be opened. Formally,

$$\texttt{Win}[\,\texttt{temp}^{\mathsf{R}}; \texttt{win}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\texttt{temp}); \mathsf{pld7}\,]$$

with

$$\begin{aligned}\mathsf{pld7} \stackrel{\text{def}}{=}\ & \texttt{temp} \leftarrow \mathsf{read}(\texttt{temp});\\ & \mathsf{if}\ (\texttt{temp} > 22)\ \mathsf{then}\ \{\texttt{win} \leftarrow \textit{Open}; \mathsf{update}(\texttt{win})\};\\ & \mathbb{X}\end{aligned}$$

Now, suppose to have a second app `Therm`, managing the thermostat of the room, such that when the temperature is below 17 degrees the thermostat is set to rise the temperature by 3 degrees. Formally,

$$\texttt{Therm}[\,\texttt{temp}^{\mathsf{R}}; \texttt{therm}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\texttt{temp}); \mathsf{pld8}\,]$$

with

$$\begin{aligned}\mathsf{pld8} \stackrel{\text{def}}{=}\ & \texttt{temp} \leftarrow \mathsf{read}(\texttt{temp});\\ & \mathsf{if}\ (\texttt{temp} < 17)\ \mathsf{then}\ \{\texttt{therm} \leftarrow \textit{+3}; \mathsf{update}(\texttt{therm})\};\\ & \mathbb{X}\end{aligned}$$

When running these two apps in parallel, we may have an *implicit interaction*, as the app `Therm` may indirectly trigger the app `Win`. This is because, we know, out of band, that the temperature of the room should somehow change according to the changes made on the thermostat of the room.

However, since this out-of-band information is not considered by our formalization, according to Definition 3 we would have a kind of *false negative* as the app `Therm` is not directly interacting with the app `Wind`. Formally, for $H \stackrel{\text{def}}{=} \mathfrak{upd}(\texttt{Therm}) = \{\texttt{Therm:therm!}v \mid v \in \mathsf{Value}\}$, we have:

$$\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{Win} \parallel \texttt{Therm} \approx_H \texttt{Win},$$

for any $\mathfrak{G} \in \mathbb{S}$.

Note that *causality dependencies* between services, such as those asserting that thermostat changes may affect the temperature, are not part of the specification of an app (or of a system of apps). This information comes from the physics of the real system managed via apps. Thus, by no means we can capture this kind of *implicit interactions* unless we have information about causality dependencies.

However, we can assume that, when designing our system of apps we actually get, out of band, a set of causality dependencies to improve the precision of our analysis ruling out a number of false negatives. For the sake of simplicity, we define a *dependency policy* as a binary relation $\Delta \subseteq \mathsf{Service} \times \mathsf{Service}$ such that $(\mathtt{x}, \mathtt{y}) \in \Delta$ when the service $\mathtt{y}$ may be affected by changes occurring at the service $\mathtt{x}$. Clearly, dependencies can be composed, hence we will consider the reflexive and

transitive closure of Δ in order to capture all possible dependencies associated to a service. We write $\mathfrak{clo}(\Delta, \mathtt{x})$ to denote the reflexive and transitive closure of the dependency policy Δ w.r.t. the service $\mathtt{x}$. More generally, given a set of services $X \subseteq \mathsf{Service}$ we define $\mathfrak{clo}(\Delta, X) \stackrel{\text{def}}{=} \bigcup_{\mathtt{x} \in X} \mathfrak{clo}(\Delta, \mathtt{x})$.

Here, it is important to notice that when we act on the thermostat of the room we actually do not know exactly how the temperature will change (again, this depends on the physics, *e.g.*, on the wall isolation of the heated room). Thus, the dependency policy Δ records only abstract information relating pairs of services. More precisely, if $(\mathtt{x}, \mathtt{y}) \in \Delta$ we may assume that each time the service $\mathtt{x}$ changes on the cloud then the service $\mathtt{y}$ can be somehow affected. We represent this abstract information by means of *nondeterministic updates* assigning to $\mathtt{y}$ the special value $*$, meaning "any value". Ideally, the special value $*$ satisfies any boolean expression containing it. For instance, $* \leq n$ is true for any n.

Now, using this extra out-of-band information Δ on the causality dependency between services, we can easily define a labeled transitions semantics $\xrightarrow{\alpha}_{\Delta}$, parametric on the set Δ:

- $\mathfrak{C}_1 \xrightarrow{\alpha}_{\Delta} \mathfrak{C}_1$ if $\mathfrak{C}_1 \xrightarrow{\alpha} \mathfrak{C}_1$ is derived by an application of any rule of Table I different from Update;
- rule Update is replaced by the following transition rule:

$$\frac{\mathfrak{G}(\mathtt{x}) \neq \phi(\mathtt{x}) \qquad \phi(\mathtt{x}) = v \qquad \mathfrak{clo}(\Delta, \mathtt{x}) = \{\mathtt{y}_1, \ldots, \mathtt{y}_n\}}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{update}(\mathtt{x}) \xrightarrow{\mathtt{x}!v}_{\Delta} \langle \mathfrak{G}[\mathtt{x} \leftarrow v, \mathtt{y}_1 \leftarrow *, \ldots, \mathtt{y}_n \leftarrow *], \phi \rangle \triangleright \mathsf{skip}}$$

Now, we can refine Definition 3 making it parametric on a dependency policy Δ. Basically, we use our hiding bisimilarity defined on top of the parametric LTS $\xrightarrow{\alpha}_{\Delta}$, denoted with $\overset{\Delta}{\approx}_H$. In this manner, we can rely on the dependency policy Δ to capture false negatives due to implicit interactions: any change on a service $\mathtt{x}$ affects any service in the set $\mathfrak{clo}(\Delta, \mathtt{x})$ via nondeterministic assignments that will always trigger apps listening at these services.

Definition 7 (Safe cross-app interaction under dependencies)**.** Let Δ be a dependency policy. Let S and R be two systems of apps in $\mathtt{CaITApp}$. We say that S is *noninteracting with* R *under* Δ when

$$\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{S}} \mathsf{R}$$

where $H_\mathsf{S} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{S})$. We say that the two systems S and R *do not interact with each other under* Δ if in addition to the requirement above we have

$$\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{R}} \mathsf{S}$$

where $H_\mathsf{R} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{R})$.

Now, in order to provide a consistent reformulation of Theorem 1 to capture semantics-based noninteraction parametric on a dependency policy Δ, we have to reformulate Definition 6 to take into account the presence of the dependency policy.

Definition 8 (Syntax-based safe cross-app interaction under dependencies)**.** Let Δ be a dependency policy. The system S is said to be *syntactically noninteracting under* Δ with the system R, written $\mathsf{S} \overset{\Delta}{\nrightarrow} \mathsf{R}$, when both the following conditions hold:

1) $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{act}(\mathsf{R}) = \varnothing$;
2) $\mathfrak{clo}(\Delta, \mathfrak{act}(\mathsf{S})) \cap \mathfrak{trg}(\mathsf{R}) = \varnothing$.

More generally, we say that the two systems S and R are *syntactically noninteracting* w.r.t. *each other under* Δ, written $\mathsf{S} \overset{\Delta}{\nleftrightarrow} \mathsf{R}$, when besides the two conditions above we also have:

3) $\mathfrak{trg}(\mathsf{S}) \cap \mathfrak{clo}(\Delta, \mathfrak{act}(\mathsf{R})) = \varnothing$.

Now, if we consider the apps in Example 6 with $\Delta = \{(\mathtt{therm}, \mathtt{temp})\}$ we have:

- $\mathtt{Win} \overset{\Delta}{\nrightarrow} \mathtt{Therm}$ holds;
- $\mathtt{Therm} \overset{\Delta}{\nrightarrow} \mathtt{Win}$ does not hold because $\mathfrak{trg}(\mathtt{Win}) = \{\mathtt{temp}\}$, $\mathfrak{clo}(\Delta, \mathfrak{act}(\mathtt{Therm})) = \{\mathtt{therm}, \mathtt{temp}\}$ and hence $\mathfrak{clo}(\Delta, \mathfrak{act}(\mathtt{Therm})) \cap \mathfrak{trg}(\mathtt{Win}) \neq \emptyset$.

Finally, we can reformulate Theorem 1 to prove that Definition 8 provides a sufficient condition to capture semantic-based noninteraction under a given dependency policy Δ.

Theorem 2 (Soundness under dependencies)**.** *Let Δ be a dependency policy. Let S and R be two systems of apps in $\mathtt{CaITApp}$. Let $H_\mathsf{S} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{S})$ and $H_\mathsf{R} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{R})$. Then:*

- $\mathsf{S} \overset{\Delta}{\nrightarrow} \mathsf{R}$ *implies* $\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{S}} \mathsf{R}$;
- $\mathsf{S} \overset{\Delta}{\nleftrightarrow} \mathsf{R}$ *implies*
  - $\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{S}} \mathsf{R}$
  - $\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{R}} \mathsf{S}$.

The details of the proof can be found in the Appendix.

Thanks to Theorem 2, for $\Delta = \{(\mathtt{therm}, \mathtt{temp})\}$, we can now correctly capture the semantic-based interaction between the apps of Example 6 as there is $\mathfrak{G} \in \mathbb{S}$ such that

$$\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathtt{Win} \parallel \mathtt{Therm} \not\approx_H \mathtt{Win}$$

for $H \stackrel{\text{def}}{=} \mathfrak{upd}(\mathtt{Therm}) = \{\mathtt{Therm{:}therm}!v \mid v \in \mathsf{Value}\}$.

B. Increasing flexibility via priorities

We envision use cases where users are aware of potential interactions on some services, while disallowing interactions on other services. For instance, an app managing the fire alarm may have higher priority than an app posting messages to Facebook. This relative importance of services can be specified with a lattice of service priorities. Suppose to have a *priority policy* Π, which associates a *priority level* $\mathfrak{p}$, taken from some complete lattice $\langle \mathrm{PL}, \sqsubseteq \rangle$, with each service used by our system of apps.

Now, we can extend our condition of hiding bisimilarity to define safety of cross-app interference up to a given priority level. In fact, given a priority level $\mathfrak{p}$, we can formally ensure that two systems have the same behaviour only when looking at observables on services at priority level greater than, or equal to $\mathfrak{p}$.

Definition 9 (Safe cross-app interaction up to priorities)**.** Let S and R be two systems of apps in $\mathtt{CaITApp}$. We say that S is *noninteracting with* R *up to the priority level* $\mathfrak{p}$ when:

$$\forall \mathfrak{G} \in \mathbb{S} \,.\, \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathfrak{p}} \mathsf{R}$$

where $H_{\mathfrak{p}} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{S}) \cup \{\mathtt{id}{:}\mathtt{x}!v \in \mathfrak{upd}(\mathsf{R}) \mid \mathfrak{p} \not\sqsupseteq \Pi(\mathtt{x})\}$.

Note that when Π maps each service to the same priority level, the definition above coincides with Definition 3. To enforce the condition above, we can easily define a syntax-based condition similar to the one given in Definition 6.

V. Securing Cross-App Interactions

It is not hard to imagine that services accessed via an IoT platform may have different security clearance. For instance, a service to access a smart security camera should definitely not leak any kind of information to a second service that is used to share nice pictures among friends.

In this section, we assume a *security policy* Σ, which associates a *security level* σ, taken from some complete lattice $\langle SL, \preccurlyeq \rangle$, with each service used by our system of apps. For the sake of simplicity, in the examples, the security levels will simply be high (H), or *secret*, and low (L), or *public*, although the theory is developed for a generic complete lattice $\langle SL, \preccurlyeq \rangle$, of security levels. The goal is to achieve classical *noninterference* results stating that a system of apps is interference-free if its low-level services are not affected by changes occurring at its high-level services. Thus, information can safely flow from a service $\mathtt{x}$ to a service $\mathtt{y}$ whenever $\Sigma(\mathtt{x}) \preccurlyeq \Sigma(\mathtt{y})$. In the following, we use $\prec$ to denote the nonreflexive restriction of $\preccurlyeq$ (*i.e.*, $\delta \prec \sigma$ iff $\delta \preccurlyeq \sigma$ and $\delta \neq \sigma$).

As usual, a security policy induces an equivalence relation between stores. Given two (global) stores $\mathfrak{G}, \mathfrak{G}' \in \mathbb{S}$, we say that they are σ-equivalent, written $\mathfrak{G} \equiv_\sigma \mathfrak{G}'$, if they agree on the values associated to all services with security level lower than, or equal to, σ.

Definition 10 (σ-equivalent stores)**.** Let $\mathfrak{G}, \mathfrak{G}' \in \mathbb{S}$ be two stores and $\sigma \in SL$ be a security level. We say that $\mathfrak{G}$ and $\mathfrak{G}'$ are *σ-equivalent*, written $\mathfrak{G} \equiv_\sigma \mathfrak{G}'$, whenever:

$$\forall \mathtt{x} \in \mathsf{Service} \,.\, \Sigma(\mathtt{x}) \preccurlyeq \sigma \Rightarrow \mathfrak{G}(\mathtt{x}) = \mathfrak{G}'(\mathtt{x}) \,.$$

Now, we can formalize a bisimulation-based notion of noninterference parametric on some security level $\sigma \in SL$. Intuitively, the runtime behavior at security level σ (or lower) of an interference-free system does not change when executed in two different σ-equivalent stores $\mathfrak{G}$ and $\mathfrak{G}'$, though it may differ on services with security clearance higher than σ. Actually, in our notion of noninterference we consider σ-equivalent stores in $\mathbb{S}_\perp \stackrel{\text{def}}{=} \{\mathfrak{G} \in \mathbb{S} \mid \forall \mathtt{x} \in \mathrm{dom}(\mathfrak{G}) \,.\, \mathfrak{G}(\mathtt{x}) \neq \perp\}$, as the mere initialization of an high-level service might activate a listener in an applet, thus leaking information about the *occurrence/presence* of a high event. We ignore presence leaks in order to increase premissivenes of our enforcement mechanism.

Our general notion of hiding bisimilarity can be used to hide (but not to suppress) actions involving changes affecting high-level services. In the following, with an abuse of notation, we extend Σ to assign security levels to system actions α, according to the cloud services affected by α. Thus, we define $\Sigma(\mathtt{id}{:}\mathtt{x}!v) = \Sigma(\mathtt{x})$ and $\Sigma(\alpha) = \perp$ for $\alpha \in \{\tau, \mathtt{x}?v\}$.[1]

However, in order to capture a semantic notion of non-interference that is not sensitive to information leaks due to *program termination*[2] we propose a modification of our hiding bisimilarity inspired by the *termination-insensitive i-bisimulation* proposed by Demange and Sands [16]. For this purpose, given a set of hidden actions H, we will write $\mathfrak{C}_1 \Uparrow_H$ if and only if $\mathfrak{C}_1 \in D \stackrel{\text{def}}{=} \{\mathfrak{C} : (\exists \alpha \in \mathcal{A} . \mathfrak{C} \stackrel{\alpha}{\rightarrow\!\triangleright} \mathfrak{C}') \wedge (\mathfrak{C} \stackrel{\alpha}{\rightarrow\!\triangleright} \mathfrak{C}' \Rightarrow \alpha \in H \cup \{\tau\} \wedge \mathfrak{C}' \in D)\}$, that is $\mathfrak{C}_1$ belongs to the set of *high-level divergent configurations* that can always and only perform either τ-actions or high-level actions.

Definition 11 (Termination-Insensitive Hiding Bisimulation)**.** Given a set of actions $H \subseteq \mathcal{A} \setminus \{\tau\}$, the symmetric relation $\mathcal{R} \subseteq \mathsf{Sconf} \times \mathsf{Sconf}$ is a *termination-insensitive hiding bisimulation* parametric on H if and only if, whenever $\mathfrak{C}_1 \;\mathcal{R}\; \mathfrak{C}_2$ and $\mathfrak{C}_1 \stackrel{\alpha}{\rightarrow\!\triangleright} \mathfrak{C}_1'$ we have the following:

- if $\alpha \notin H$ then
  - either $\mathfrak{C}_2 \stackrel{\hat{\alpha}}{\Rightarrow} \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \;\mathcal{R}\; \mathfrak{C}_2'$
  - or $\mathfrak{C}_2 \Uparrow_H$
- if $\alpha \in H$ then
  - either $\mathfrak{C}_2 \stackrel{\hat{\alpha}}{\Rightarrow} \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \;\mathcal{R}\; \mathfrak{C}_2'$
  - or $\mathfrak{C}_2 \stackrel{\tau}{\rightarrow\!\triangleright}{}^{*} \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \;\mathcal{R}\; \mathfrak{C}_2'$.

We say that two system configurations $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are *termination-insensitive hiding bisimilar w.r.t.* the set of actions H, written $\mathfrak{C}_1 \approx_H^{\text{ti}} \mathfrak{C}_2$, if $\mathfrak{C}_1 \;\mathcal{R}\; \mathfrak{C}_2$ for some termination-insensitive hiding bisimulation $\mathcal{R}$ parametric on H.

Definition 12 (σ-level noninterference)**.** Let S be a system of apps and $H_\sigma \stackrel{\text{def}}{=} \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\}$ the set of actions with clearance greater than σ. We say that S is *σ-level interference-free* whenever:

$$\forall \mathfrak{G}, \mathfrak{G}' \in \mathbb{S}_\perp \,.\, \mathfrak{G} \equiv_\sigma \mathfrak{G}' \;\Rightarrow\; \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx_{H_\sigma}^{\text{ti}} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \,.$$

Example 7. Consider the classic two-points lattice $\{\mathsf{L}, \mathsf{H}\}$, used for the system of apps $\mathsf{S} \stackrel{\text{def}}{=} \mathtt{Tw2Fb} \parallel \mathtt{Fb2Ld}$ of Example 1 such that: $\Sigma(\mathtt{tw}) = \Sigma(\mathtt{fb}) = \mathsf{H}$ and $\Sigma(\mathtt{ld}) = \mathsf{L}$. Obviously, the compound system is exposed to a security interference, as confidential information posted on $\mathtt{tw}$ will flow into the public service $\mathtt{ld}$. In fact, it is not hard to find two L-equivalent global stores $\mathfrak{G}$ and $\mathfrak{G}'$ such that $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \not\approx_{H_\mathsf{L}}^{\text{ti}} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$. This would not be the case if it was $\Sigma(\mathtt{ld}) = \mathsf{H}$. In that case, the bisimilarity would hold for any pair of L-equivalent stores.

Again, Definition 12 has a universal quantification on two global environments and then it requires the verification of a nontrivial bisimilarity. So, its verification is hard to achieve.

In order to provide a syntactic sufficient condition for security noninterference we resort to a type system, *parametric in the security policy* Σ, and inspired by the *flow-sensitive*

[1] We recall that the action $\mathtt{x}?v$ is not "originating" from our system but it denotes a modification of the service $\mathtt{x}$ made by the external environment.

[2] Termination leaks have well-known information-theoretic bounds [4], and they are usually ignored in order to increase permissiveness for static analysis that do not consider for program termination.

Typing rules for expressions

$$(\textsc{Var})\ \frac{\Gamma(\mathsf{x}) = \sigma}{\Gamma \vdash \mathsf{x} : \sigma} \qquad (\textsc{Read})\ \frac{\Sigma(\mathsf{x}) = \sigma}{\Gamma \vdash \mathsf{read}(\mathsf{x}) : \sigma} \qquad (\textsc{Expr})\ \frac{\Gamma \vdash \mathsf{e}_1 : \sigma_1 \qquad \Gamma \vdash \mathsf{e}_2 : \sigma_2}{\Gamma \vdash \mathsf{e}_1\ \mathsf{op}\ \mathsf{e}_2 : \sigma_1 \sqcup \sigma_2}$$

Typing rules for processes

$$(\textsc{Skip})\ \frac{-}{pc \vdash \Gamma\{\mathsf{skip}\}\Gamma} \qquad (\textsc{Assign})\ \frac{\Gamma \vdash \mathsf{e} : \sigma}{pc \vdash \Gamma\{\mathsf{x} \leftarrow \mathsf{e}\}\Gamma[\mathsf{x} \mapsto \sigma \sqcup pc]} \qquad (\textsc{Fix})\ \frac{pc \vdash \Gamma\{\mathsf{P}\}\Gamma'}{pc \vdash \Gamma\{\mathsf{fix}\,\mathbb{X} \bullet \mathsf{P}\}\Gamma'}$$

$$(\textsc{Update})\ \frac{\Gamma(\mathsf{x}) \preccurlyeq \Sigma(\mathsf{x})}{pc \vdash \Gamma\{\mathsf{update}(\mathsf{x})\}\Gamma} \qquad (\textsc{Pvar})\ \frac{-}{pc \vdash \Gamma\{\mathbb{X}\}\Gamma} \qquad (\textsc{Listen})\ \frac{-}{pc \vdash \Gamma\{\mathsf{listen}(\mathsf{L})\}\Gamma}$$

$$(\textsc{IfElse})\ \frac{\Gamma \vdash \mathsf{b} : \sigma \qquad pc \sqcup \sigma \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_2 \qquad pc \sqcup \sigma \vdash \Gamma_1\{\mathsf{P}_2\}\Gamma_2}{pc \vdash \Gamma\{\mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \{\mathsf{P}_1\}\ \mathsf{else}\ \{\mathsf{P}_2\}\}\Gamma_2} \qquad (\textsc{Seq})\ \frac{pc \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_2 \qquad pc \vdash \Gamma_2\{\mathsf{P}_2\}\Gamma_3}{pc \vdash \Gamma_1\{\mathsf{P}_1; \mathsf{P}_2\}\Gamma_3}$$

Typing rules for systems and subtyping

$$(\textsc{App})\ \frac{pc \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2}{pc \vdash \mathsf{id}[\mathsf{D} \bowtie \mathsf{P}]} \qquad (\textsc{Par})\ \frac{pc \vdash \mathsf{S}_1 \qquad pc \vdash \mathsf{S}_2}{pc \vdash \mathsf{S}_1 \parallel \mathsf{S}_2} \qquad (\textsc{Sub.Proc})\ \frac{pc \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2 \qquad pc' \preccurlyeq pc \qquad \Gamma_1' \preccurlyeq \Gamma_1 \qquad \Gamma_2 \preccurlyeq \Gamma_2'}{pc' \vdash \Gamma_1'\{\mathsf{P}\}\Gamma_2'}$$

TABLE III
SECURITY TYPE SYSTEM

security type system of Hunt and Sands [23], adapted to our setting. In Table III we provide the typing rules.

Intuitively, a judgment of the form $pc \vdash \mathsf{S}$ says that the system S does not contain information flows from services at security level higher than pc to services at security level lower than or equal to pc. Here, pc denotes the usual "program counter" level and serves to eliminate indirect information flows. We write $\vdash \mathsf{S}$ to denote $pc \vdash \mathsf{S}$ when pc is the least security level, *i.e.*, the bottom element of the lattice SL.

Since the syntax of our calculus is in two levels we also have a different kind of judgments for processes running inside an app: $pc \vdash \Gamma\{\mathsf{P}\}\Gamma'$. Here, as in Hunt and Sands [23], Γ describes the security levels of services which hold before execution of P while Γ' describe the security level of those services after execution of P. Again, pc denotes the "program counter" level and the derivation rules ensure that only services which end up (in Γ') with types greater than or equal to pc may be changed by P.

Here, we wish to remark that, unlike batch-job noninterference models [37], a security information flow occurs in our setting only if a low-level service x is subject to an information flow and then x is "published" within the same app on the cloud via an update construct. In fact, the update operator is the only "exit gate" for potential (direct or indirect) information flows created within an app. This requires some care in the definition of the typing rule (Update). Basically, we impose that the update of a global service x is possible only if x is associated to an "original" security level (given by Σ) higher than or equal to the security level derived by its use (given by Γ) in the previous instructions within the app. On the other hand, we consider harmless those information flows that remain confined within an app because no update publishes their effects; this situations will not be ruled out by our type system. Finally, notice the difference between the two typing rules (Var) and (Read) as they serve for typing accesses to local views of services and global services, respectively.

As expected, system (and process) well-typedness is preserved at runtime.

Proposition 1 (Subject reduction)**.** *Let Σ be a security policy, S a system of apps, and $\sigma \in SL$ a security level. If $pc \vdash \mathsf{S}$ and $\langle \mathfrak{G}, \mathfrak{L} \rangle \triangleright \mathsf{S} \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{S}'$ then $pc \vdash \mathsf{S}'$.*

Subject reduction is a crucial ingredient to prove that well-typedness is a sufficient condition to ensure noninterference.

Theorem 3 (Soundness of security types)**.** *Let Σ be a security policy, S a system of apps and $H_\sigma \stackrel{\text{def}}{=} \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\}$ the set of all possible high-level actions. If $\vdash \mathsf{S}$ then*

$$\forall \mathfrak{G}, \mathfrak{G}' \in \mathbb{S}_\perp \,.\, \mathfrak{G} \equiv_\sigma \mathfrak{G}' \Rightarrow \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx^{\text{ti}}_{H_\sigma} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}\,.$$

Proof. (*Sketch*) The proof proceeds by contradiction. Consider two configurations $\mathfrak{C} \stackrel{\text{def}}{=} \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$ and $\mathfrak{C}' \stackrel{\text{def}}{=} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$ for some well-typed system S, and differing for two σ-equivalent global stores $\mathfrak{G}$ and $\mathfrak{G}'$, respectively. If $\mathfrak{C}$ and $\mathfrak{C}'$ are not bisimilar then whenever we try to build up a termination-insensitive hiding bisimulation $\mathcal{R}$ containing the pair $(\mathfrak{C}, \mathfrak{C}')$, the bisimulation game will stop in a pair of configurations $(\mathfrak{C}_i, \mathfrak{C}_i')$, with $\mathfrak{C}_i$ (*resp.*, $\mathfrak{C}_i'$) derivative of $\mathfrak{C}$ (*resp.*, $\mathfrak{C}'$), such that $\mathfrak{C}_i$ can perform some action α that cannot be (weakly) mimicked by $\mathfrak{C}_i'$ (or vice versa). Since both $\mathfrak{C}_i$ and $\mathfrak{C}_i'$ are derivatives of well-typed configurations (actually well-typed systems), by subject reduction (Proposition 1) the corresponding systems are both well-typed.

Now, since our termination-insensitive hiding bisimilarity neglects high-level actions, by a case analysis on the distinguishing action α, the only possibility is that α denotes the update of a low-level service derived by the evaluation of some high-level service, the only ones on which the two global stores may differ. But this would lead to a security information

flow that is prevented by the type systems as both $\mathfrak{C}_i$ and $\mathfrak{C}'_i$ are well-typed. Thus, this action is not admissible.

As a consequence, since there are no distinguishing system actions α, the two original configurations $\mathfrak{C}$ and $\mathfrak{C}'$ must be bisimilar.

Full details of the proof can be found in the Appendix. □

Example 8. Let us show that system S of Example 7 is cannot be typed in our type system. The process P running inside the applet `Tw2Fb` is typable: $\mathsf{L} \vdash \Gamma\{\mathsf{P}\}\Gamma$, for $\Gamma = [\mathtt{tw} \mapsto \mathsf{H}, \mathtt{fb} \mapsto \mathsf{H}, \mathtt{ld} \mapsto \mathsf{L}]$. As a consequence, the app is typable as well: $\vdash \mathtt{Tw2Fb}[\mathtt{tw}^{\mathsf{R}}; \mathtt{fb}^{\mathsf{W}} \bowtie \mathsf{P}]$. On the other hand, the process Q running within the applet `Fb2Ld` is not typable. In fact, when building the derivation tree for the typing, we have to type the assignment $\mathtt{ld} \leftarrow \mathtt{fb}$ as $\mathsf{L} \vdash \Gamma_1\{\mathtt{ld} \leftarrow \mathtt{fb}\}\Gamma_2$, for $\Gamma_1 = [\mathtt{tw} \mapsto \mathsf{H}, \mathtt{fb} \mapsto \mathsf{H}, \mathtt{ld} \mapsto \mathsf{L}]$ and $\Gamma_2 = [\mathtt{tw} \mapsto \mathsf{H}, \mathtt{fb} \mapsto \mathsf{H}, \mathtt{ld} \mapsto \mathsf{H}]$. But then, the subsequent update $\mathtt{update}(\mathtt{ld})$ cannot be typed, since the current typing of `ld` is H while the initial typing of `ld` is $\Sigma(\mathtt{id}) = \mathsf{L}$. Thus, the app $\mathtt{Fb2Ld}[\mathtt{fb}^{\mathsf{R}}; \mathtt{ld}^{\mathsf{W}} \bowtie \mathsf{Q}]$ is not typable and, in turn, the parallel composition S is not typable.

We remark that our security type system did not face any permissiveness issues (i.e., false positives) for the apps considered in the paper. We expect our analysis to scale well and produce a minimal false-positive rate for user-automation IoT platforms like IFTTT and Stringify. In these platforms, the code consist of simple snippets matching the syntax of `CaITApp` closely [6]. For other IoT platforms like SmartThings, the code can be more complex (in fact, SmartThings apps are implemented in Groovy), hence our analysis would face classical challenges for type-based approaches due to complex language features, e.g., *aliasing*.

VI. Related Work

Security and safety risks in the IoT domain have been the subject of a large array of research studies. We refer to recent surveys for an overview of the area [2], [5], [12]. Here, we compare our contributions with closely related works on security and safety analysis of IoT apps, information-flow control, and formal models for IoT.

Securing IoT apps: Recent research points out the security and safety risks arising in the context of IoT apps. Surbatovich et al. [36] study a dataset of 20K IFTTT applets and provide an empirical evaluation of potential secrecy and integrity violations, including violations due to cross-app interactions. Celik et al. [13], [14] propose static and dynamic enforcement mechanisms for unveiling cross-app interference vulnerabilities. Ding et al. [17] propose a framework that combines device physical channel analysis and static analysis to generate all potential interaction chains among applications in an IoT environment. They leverage natural language processing to identify services that have similar semantics, and propose a risk-based approach to classify the actual risks of the discovered interaction chains. Chi et al. [15] propose a systematic categorization of threats arising from unintentional or malicious interaction of apps in IoT platforms. To detect cross-app interference, they use symbolic execution techniques to analyze the apps's implementation. Nguyen et al. [31] design IoTSan, a system that uses model checking to reveal cross-app interaction flows. All the above-mentioned works provide an excellent motivation for our foundational contributions. Our policy framework can be used to validate soundness and permissiveness of these verification techniques.

Another line of work focuses on enforcement mechanisms for checking security and safety of single IoT apps. Fernandes et al. [18] present FlowFence, an approach building secure IoT apps via information-flow tracking and controlled declassification. Celik et al. [11] leverage static taint tracking to identify sensitive data leaks in IoT apps. Bastys et al. [6], [7] identify new attack vectors in IFTTT applets and show that 30% of applets from their dataset can be subject to such attacks. As a countermeasure, they investigate static and dynamic information-flow tracking via security types. Fernandes et al. [19] propose the use of decentralization and fine-grained authentication tokens to limit privileges and prevent unauthorized actions. In contrast, our work targets security and safety issues in cross-app interactions, and it focuses on the formal underpinnings of these approaches.

Information-flow control: Several works propose information-flow control for enforcing confidentiality and integrity policies in emerging domains like IoT. We refer to a survey by Focardi and Gorrieri [20] for an overview on information-flow properties in process algebra. Our semantic condition of safe cross-app interaction draws inspiration from Focardi and Martinelli's notion of Generalized Non Deducibility on Composition (GNDC) [21], applied along the lines of [27]. Volpano and Smith [37] study a *flow-insensitive* type system for imperative languages. Because in our language the communication betweeen services is handled via explicit *update* statements, a flow-insensitive type system would be too restrictive and reject more secure programs. Hunt and Sands [23] propose a flow-sensitive type system for an imperative language. Our work extends their type system to ensure security for a system of apps running concurrently. Similarly to our definition of termination-insensitive hiding bisimulation, Demange and Sands [16] propose a weakening of low bisimulation conditions to ignore leaks arising from program termination. In contrast, the execution of our app's payload affects the global store via a well-defined interface, i.e., listeners and update statements, which makes our systems of apps more amenable for enforcing security and safety properties.

There are a few approaches that carry out information-flow analysis on discrete/continuous models for cyber-physical systems. Akella et al. [1] proposed an approach to perform information flow analysis, including both trace-based analysis and automated analysis through process algebra specifications. This approach has been used to verify process algebra models of a gas pipeline system and a smart electric power grid system. Wang [38] propose Petri-net models to verify *nondeducibility security properties* of a natural gas pipeline system. More recently, Bohrer and Platzer [10] introduce dHL,

a hybrid logic for verifying cyber-physical hybrid-dynamic information flows, communicating information through both discrete computation and physical dynamics, ensuring security in presence of attackers that observe *continuously-changing values* in continuous time.

Formalizing IoT semantics: IoT semantics has been subject to several works aiming at capturing subtle IoT-specific notions like time and device state. Newcomb et al. [30] propose IOTA, a calculus for the domain of home automation. Based on the core formalism of IOTA, the authors develop an analysis for detecting whenever an event can trigger two conflicting actions, and an analysis for determining the root cause of (non)occurrence of an event. Lanese et al. [25] propose a calculus of mobile IoT devices interacting with the physical environment by means of sensors and actuators. The calculus does not allow any representation of the physical environment, while it is equipped with an end-user bisimilarity in which end-users may: (i) provide values to sensors, (ii) check actuators, and (iii) observe the mobility of smart devices. End-user bisimilarity is not preserved by parallel composition. Compositionality is recovered by strengthening its discriminating power. Lanotte and Merro [26] extend and generalize the work of [25] in a timed setting by providing a bisimulation-based semantic theory that is suitable for compositional reasoning. Bodei et al. [9] propose an untimed process calculus, IoT-LYSA, supporting a control flow analysis that safely approximates the abstract behavior of IoT systems. Essentially, they track how data spread from sensors to the logic of the network, and how physical data are manipulated. The calculus adopts asynchronous multi-party communication among nodes taking care of node proximity. The dynamics of the calculus is given in terms of a reduction relation. In [8], the same authors extend their work to infer *quantitative measures* to establish the cost of possibly security countermeasures, in terms of time and energy. In contrast, our calculus models constructs that are relevant for our security and safety analysis of cross-app interactions in IoT platforms, while ignoring details of the physical environment.

VII. Conclusion

IoT platforms empower users by connecting a wide array of otherwise unconnected services and devices. These platforms routinely execute IoT apps that have access to sensitive information of their users. Because different apps of a user may affect a common physical or logical environment, their interaction (even for benign apps) can cause severe security and safety risks for that user.

Motivated by this setting, we proposed a generic foundational framework for securing cross-app interactions. We presented an extensional condition that captures the essence of safe cross-app interactions, as well as implicit interactions and priorities. Moreover, we studied an extensional condition for confidentiality and integrity properties of a system of apps, and proposed a flow-sensitive security type system to enforce such condition.

Because of the simplicity of the execution model and the relatively-small code size, IoT apps offer a promising avenue for integrating formal techniques studied by the community to real-world products. In future work, we plan to implement our techniques for IoT platforms like Android Things and Node-RED, and investigate their impact on users' security and safety.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. The first author was partly funded by the Swedish Research Council (VR) under the project JointForce, and by the Swedish Foundation for Strategic Research (SSF) under the project TrustFull. The second author has been partially supported by the project "Dipartimenti di Eccellenza 2018-2022", funded by the Italian Ministry of Education, Universities and Research (MIUR). The third author has been supported by the Joint Project 2017 "Security Static Analysis for Android Things", jointly funded by the University of Verona and JuliaSoft Srl.

References

[1] R. Akella, H. Tang, and B. M. McMillin. Analysis of information flow security in cyber-physical systems. *International Journal of Critical Infrastructure Protection*, 3(3-4):157–173, 2010.

[2] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security evaluation of home-based iot deployments. In *IEEE S&P*, pages 208–226. IEEE Computer Society, 2019.

[3] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, volume 5283 of *LNCS*, pages 333–348. Springer, 2008.

[5] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, September/October 2019.

[6] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *ACM CCS*, pages 1102–1119. ACM, 2018.

[7] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking information flow via delayed output - Addressing privacy in IoT and emailing apps. In *NordSec*, volume 11252 of *LNCS*, pages 19–37. Springer, 2018.

[8] C. Bodei, S. Chessa, and L. Galletta. Measuring security in IoT communications. *Theoretical Computer Science*, 764:100–124, 2019.

[9] C. Bodei, P. Degano, G. Ferrari, and L. Galletta. Tracing where IoT data are collected and aggregated. *Logical Methods in Computer Science*, 13(3):1–38, 2017.

[10] B. Bohrer and A. Platzer. A Hybrid, Dynamic Logic for Hybrid-Dynamic Information Flow. In *LICS*, pages 115–124. ACM, 2018.

[11] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity IoT. In *USENIX*, pages 1687–1704. USENIX Association, 2018.

[12] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *CoRR*, arXiv:1809.06962, 2018.

[13] Z. B. Celik, P. McDaniel, and G. Tan. Soteria: Automated iot safety and security analysis. In *USENIX*, pages 147–158, 2018.

[14] Z. B. Celik, G. Tan, and P. D. M. and. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*. The Internet Society, 2019.

[15] H. Chi, Q. Zeng, X. Du, and J. Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. *CoRR*, abs/1808.02125, 2018.

[16] D. Demange and D. Sands. All Secrets Great and Small. In *ESOP*, volume 5502 of *LNCS*, pages 207–221, 2009.

[17] W. Ding and H. Hu. On the safety of IoT device physical interaction control. In *ACM CCS*, pages 832–846. ACM, 2018.

[18] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX*, pages 531–548, 2016.

[19] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*. The Internet Society, 2018.
[20] R. Focardi and R. Gorrieri. Classification of security properties (Part I: Information flow). In *FOSAD*, pages 331–396. Springer-Verlag, 2001.
[21] R. Focardi and F. Martinelli. A Uniform Approach for the Definition of Security Properties. In *FM*, volume 1708 of *LNCS*, pages 794–813. Springer, 1999.
[22] K. Honda and M. Tokoro. An object calculus for asynchronous communications. In *ECOOP*, volume 512 of *LNCS*. Springer, 1991.
[23] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL*, pages 79–90. ACM, 2006.
[24] IFTTT: IF This Then That. https://ifttt.com, 2019.
[25] I. Lanese, L. Bedogni, and M. Di Felice. Internet of Things: a process calculus approach. In *SAC*, pages 1339–1346. ACM, 2013.
[26] R. Lanotte and M. Merro. A semantic theory of the Internet of Things. *Information and Computation*, 259(1):72–101, 2018.
[27] D. Macedonio and M. Merro. A semantic analysis of key management protocols for wireless sensor networks. *Science of Computer Programming*, 81:53–78, 2014.
[28] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
[29] Microsoft Flow. https://flow.microsoft.com/en-us/, 2019.
[30] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan. IOTA: A calculus for internet of things automation. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward!, pages 119–133, 2017.
[31] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel. IoTSan: Fortifying the Safety of IoT Systems. In *CoNEXT*, pages 191–203. ACM, 2018.
[32] Nest Thermostat. https://ifttt.com/services/nest_thermostat, 2019.
[33] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
[34] Stringify. https://www.stringify.com/, 2019.
[35] SmartThings. https://ifttt.com/smartthings, 2019.
[36] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, pages 1501–1510. ACM, 2017.
[37] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
[38] J. Wang and H. Yu. Analysis of the Composition of Non-Deducibility in Cyber-Physical Systems. *Applied Mathematics & Information*, 8:3137–3143, 2014.

APPENDIX

A. Proofs

Proof of Theorem 1. It suffices to prove the first part of the theorem (for the second part, just swap S with R). Assume $\mathsf{S} \nrightarrow \mathsf{R}$, *i.e.*, $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{act}(\mathsf{R}) = \varnothing = \mathfrak{act}(\mathsf{S}) \cap \mathfrak{trg}(\mathsf{R})$, then we have to prove that for any global store $\mathfrak{G}$ we have: $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathsf{S}} \mathsf{R}$, with $H_\mathsf{S} = \mathfrak{upd}(\mathsf{S})$. The proof is by contradiction.

Suppose S and R be two syntactically noninteracting systems such that $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \not\approx_{H_\mathsf{S}} \mathsf{R}$. By definition of bisimulation, this means that whenever we try to build up a hiding bisimulation $\mathcal{R}$ containing the pair $(\mathfrak{C}, \mathfrak{C}')$, with $\mathfrak{C} \stackrel{\text{def}}{=} \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R}$ and $\mathfrak{C}' \stackrel{\text{def}}{=} \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{R}$, the bisimulation game will stop with a pair of configurations $(\mathfrak{C}_i, \mathfrak{C}'_i)$, with $\mathfrak{C}_i$ (*resp.*, $\mathfrak{C}'_i$) derivative of $\mathfrak{C}$ (*resp.*, $\mathfrak{C}'$), where $\mathfrak{C}_i$ can perform an action α that cannot be (weakly) mimicked by $\mathfrak{C}'_i$ (or vice versa).

Let proceed by case analysis on the action α that would distinguish the two configurations.

– $\alpha = \tau$. We notice that τ-actions cannot distinguish the two systems as we adopted a weak notion of bisimulation.

– $\alpha = \mathtt{x}?v$. This action can be only derived by an application of rule (EnvChange) in Table II. However, as already pointed out, this action denotes a modification of the cloud made by the external observer. Thus, this action does not depend on the actual configuration and can always be performed by both configurations.

– $\alpha = \mathtt{id}{:}\mathtt{x}!v$. We have two sub-cases.

- $\mathtt{id}$ is an applet of the system S. In this case, $\alpha \in H_\mathsf{S}$, and by definition of hiding bisimulation this action can always be mimicked by an arbitrary number (possibly 0) of τ-actions.
- $\mathtt{id}$ is an applet of the system R. In this case, $\alpha \notin H_\mathsf{S}$. As α is the distinguishing action, it follows that the app $\mathtt{id}$ reaches different states in the two configurations $\mathfrak{C}_i$ and $\mathfrak{C}'_i$ leading to two possible situations: (i) the writing on $\mathtt{x}$ is possible in $\mathfrak{C}_i$ but not in $\mathfrak{C}'_i$; (ii) the writing on $\mathtt{x}$ is possible in both configurations but with different values. Since both $\mathsf{S} \parallel \mathsf{R}$ and R start in the same global store (the local store is not initialized in both cases), the system R could exhibit different behaviors if and only if it is affected by S. In particular, it means that S must modify a service that R listens on, or a service that R reads from the global store. Recall that there is no direct information passing between applets, so the only way for applets to interact is via the global store. However, we assumed that $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{trg}(\mathsf{R})$ is empty, hence this situation is not possible.

As it does not exist a distinguishing action α, it follows that the original configurations $\mathfrak{C}$ and $\mathfrak{C}'$ must be hiding bisimilar, with $H_\mathsf{S} = \mathfrak{upd}(\mathsf{S})$. □

Proof of Theorem 2. The proof goes along the same lines of that of Theorem 1. Indeed, the setting of the bisimulation is the same but the syntactic condition is stronger than the one of Theorem 1: $\mathfrak{act}(\mathsf{S}) \subseteq \mathfrak{clo}(\Delta, \mathfrak{act}(\mathsf{S}))$ and the relation $\stackrel{\Delta}{\nleftrightarrow}$ (see Definition 8) is more restrictive than $\nleftrightarrow$ (see Definition 6). □

Now, let us provide a technical lemma which is useful to prove the subject reduction property of the type system defined in Table III.

Lemma 1. *Let Σ be a security policy, P a process, and $\sigma \in SL$ a security level. If $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$ and $\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P} \stackrel{\lambda}{\rightarrow} \langle \mathfrak{G}', \mathfrak{L}' \rangle \triangleright \mathsf{P}'$ then there is Γ'_1 such that $\sigma \vdash \Gamma'_1\{\mathsf{P}'\}\Gamma_2$.*

Proof. The proof is by rule induction on the transitions rules defining the semantics of processes (Table I). In that table, all transition rules are axioms (base cases of the induction) except for the rule (Seq) (the only inductive case). Let us proceed by case analysis on

$$\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P} \stackrel{\lambda}{\rightarrow} \langle \mathfrak{G}', \phi' \rangle \triangleright \mathsf{P}'.$$

We examine the most significant cases.

- Rule (SetLocal). In this case, we have $\mathsf{P} = \mathtt{x} \leftarrow \mathsf{e}$, $\mathsf{P}' = \mathsf{skip}$, $\Gamma_1 \vdash \mathsf{e} : \delta$, for some δ, and $\Gamma_2 = \Gamma_1[\mathtt{x} \mapsto \delta \sqcup \sigma]$.

Then, setting $\Gamma_1' = \Gamma_2$, we have $\sigma \vdash \Gamma_1'\{\mathsf{skip}\}\Gamma_2$ by an application of the typing rule (Skip).

- Rule (Update). In this case, we have $\mathsf{P} = \mathsf{update}(\mathtt{x})$, $\mathsf{P}' = \mathsf{skip}$, $\Sigma(\mathtt{x}) = \delta$, for some δ, $\Gamma_1 \vdash \mathtt{x} : \delta'$, for some $\delta' \preccurlyeq \delta$ and $\Gamma_1 = \Gamma_2$. Then, setting $\Gamma_1' = \Gamma_1 = \Gamma_2$, we have $\sigma \vdash \Gamma_1'\{\mathsf{skip}\}\Gamma_2$ by an application of the typing rule (Skip).
- Rule (IfTrue). In this case, $\mathsf{P} = \mathsf{if\ b\ then}\ \{\mathsf{P}_1\}\ \mathsf{else}\ \{\mathsf{P}_2\}$, $\mathsf{P}' = \mathsf{P}_1$ and $\Gamma_1 \vdash \mathsf{b} : \delta$, for some δ. Then, by definition of the the typing rule (IfElse) we have that $\sigma \sqcup \delta \vdash \Gamma_1'\{\mathsf{P}_1\}\Gamma_2$, for some Γ_1'. Finally, since $\sigma \preccurlyeq \sigma \sqcup \delta$, by an application of sub-typing we derive $\sigma \vdash \Gamma_1'\{\mathsf{P}_1\}\Gamma_2$.
- Rule (Seq). In this case, $\mathsf{P} = \mathsf{P}_1;\mathsf{P}_2$, $\sigma \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_3$, $\sigma \vdash \Gamma_3\{\mathsf{P}_2\}\Gamma_2$, $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}'', \phi''\rangle \triangleright \mathsf{P}_1'$ and $\mathsf{P}' = \mathsf{P}_1';\mathsf{P}_2$. Since the derivation tree for $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}'', \phi''\rangle \triangleright \mathsf{P}_1'$ is smaller than the derivation tree for $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P} \xrightarrow{\lambda} \langle \mathfrak{G}', \phi'\rangle \triangleright \mathsf{P}'$, and $\sigma \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_3$, by inductive hypothesis, we have that $\sigma \vdash \Gamma_1''\{\mathsf{P}_1'\}\Gamma_3$, for some Γ_1''. Thus, setting $\Gamma_1' = \Gamma_1''$, since $\sigma \vdash \Gamma_3\{\mathsf{P}_2\}\Gamma_2$, by an application of the typing rule (Seq) it follows that $\sigma \vdash \Gamma_1''\{\mathsf{P}_1';\mathsf{P}_2\}\Gamma_2$.

□

Proof of Proposition 1. Given a security policy Σ, a system of apps S and a security level $\sigma \in SL$, we have to prove that if $\sigma \vdash \mathsf{S}$ and $\langle \mathfrak{G}, \mathfrak{L}\rangle \triangleright \mathsf{S} \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}'\rangle \triangleright \mathsf{S}'$ then $\sigma \vdash \mathsf{S}'$.

The proof is by rule induction on the transitions rules defining the semantics of systems (Table II). We proceed by case analysis on why

$$\langle \mathfrak{G}, \mathfrak{L}\rangle \triangleright \mathsf{S} \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}'\rangle \triangleright \mathsf{S}'.$$

- Rule (App). In this case, we have $\mathsf{S} = \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$, $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, $\langle \mathfrak{G}, \mathfrak{L}(\mathtt{id})\rangle \triangleright \mathsf{P} \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}'(\mathtt{id})\rangle \triangleright \mathsf{P}'$ and $\mathsf{S}' = \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}'\,]$. By an application of Lemma 1, we derive that $\sigma \vdash \Gamma_1'\{\mathsf{P}'\}\lambda_2$, for some Γ_1'. By an application of the typing rule (App) we derive $\sigma \vdash \mathsf{S}'$.
- Rule (AppUpdate). Similar to the previous case.
- Rule (EnvChange). In this case, $\mathsf{S} = \mathsf{S}'$ and we trivially have $\sigma \vdash \mathsf{S}'$.
- Rule (ParLeft). In this case, we have $\mathsf{S} = \mathsf{S}_1 \parallel \mathsf{S}_2$, $\langle \mathfrak{G}, \mathfrak{L}\rangle \triangleright \mathsf{S}_1 \xrightarrow{\alpha} \langle \mathfrak{G}'', \mathfrak{L}''\rangle \triangleright \mathsf{S}_1'$, $\sigma \vdash \mathsf{S}_1$ and $\mathsf{S}' = \mathsf{S}_1' \parallel \mathsf{S}_2$. Since the derivation tree for $\langle \mathfrak{G}, \mathfrak{L}\rangle \triangleright \mathsf{S}_1 \xrightarrow{\alpha} \langle \mathfrak{G}'', \mathfrak{L}''\rangle \triangleright \mathsf{S}_1'$ is smaller than the derivation tree for $\langle \mathfrak{G}, \mathfrak{L}\rangle \triangleright \mathsf{S} \xrightarrow{\alpha} \langle \mathfrak{G}', \mathfrak{L}'\rangle \triangleright \mathsf{S}'$, and $\sigma \vdash \mathsf{S}_1$, by inductive hypothesis it follows that $\sigma \vdash \mathsf{S}_1'$. Thus, by an application of the typing rule (Par) it follows that $\sigma \vdash \mathsf{S}_1' \parallel \mathsf{S}_2$.
- Rule (ParRight). Similar to the previous case.

□

Proof of Theorem 3. Assume that $\vdash \mathsf{S}$ and $\mathfrak{G} \equiv_\sigma \mathfrak{G}'$, then we have to prove that $\langle \mathfrak{G}, \mathfrak{L}_\perp\rangle \triangleright \mathsf{S} \approx^{\mathrm{ti}}_{H_\sigma} \langle \mathfrak{G}', \mathfrak{L}_\perp\rangle \triangleright \mathsf{S}$, with $H_\sigma = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\}$. The proof is by contradiction.

Suppose there exist two stores $\mathfrak{G}, \mathfrak{G}' \in \mathbb{S}_\perp$ such that $\mathfrak{G} \equiv_\sigma \mathfrak{G}'$ and $\langle \mathfrak{G}, \mathfrak{L}_\perp\rangle \triangleright \mathsf{S} \not\approx^{\mathrm{ti}}_{H_\sigma} \langle \mathfrak{G}', \mathfrak{L}_\perp\rangle \triangleright \mathsf{S}$. This implies that whenever we try to build up a termination-insensitive hiding bisimulation $\mathcal{R}$ containing the pair $(\mathfrak{C}, \mathfrak{C}')$, with $\mathfrak{C} \stackrel{\text{def}}{=} \langle \mathfrak{G}, \mathfrak{L}_\perp\rangle \triangleright \mathsf{S}$ and $\mathfrak{C}' \stackrel{\text{def}}{=} \langle \mathfrak{G}', \mathfrak{L}_\perp\rangle \triangleright \mathsf{S}$, the bisimulation game will stop in a pair of configurations $(\mathfrak{C}_i, \mathfrak{C}_i')$, with $\mathfrak{C}_i$ (*resp.*, $\mathfrak{C}_i'$) derivative of $\mathfrak{C}$ (*resp.*, $\mathfrak{C}'$), where $\mathfrak{C}_i$ can perform an action α that cannot be (weakly) mimicked by $\mathfrak{C}_i'$ (or vice versa). Since both $\mathfrak{C}_i$ and $\mathfrak{C}_i'$ are derivatives of well-typed configurations (actually well-typed systems), by subject reduction (Proposition 1) the corresponding systems are both well-typed.

Let proceed by case a analysis on the action α that would distinguish the two configurations.

– $\alpha = \tau$. We notice that τ-actions cannot be a problem as we adopted a weak notion of bisimulation which can also mimic τ-actions.

– $\alpha = \mathtt{x}?v$. This action can be only derived by an application of rule (EnvChange) in Table II. However, as already pointed out, this action denotes a modification of the cloud made by the external observer. Thus, this action does not depend on the actual configuration and can always be performed by both configurations.

– $\alpha = \mathtt{id}{:}\mathtt{x}!v$. We have two sub-cases.

- $\Sigma(\mathtt{x}) \not\preccurlyeq \sigma$. That is α is a high-level action. In this case, $\alpha \in H_\sigma$, and by definition of our bisimulation this action can always be mimicked by an arbitrary number (possibly 0) of τ-actions.
- $\Sigma(\mathtt{x}) \preccurlyeq \sigma$. That is α is a low-level action, or more precisely an action with security level smaller than or equal to σ. As α is the distinguish action, it follows that the app id reaches different states in the two configurations $\mathfrak{C}_i$ and $\mathfrak{C}_i'$ leading to two possible situations: (i) the low-level writing on x is possible in $\mathfrak{C}_i$ but not in $\mathfrak{C}_i'$; (ii) the low-level writing on x is possible in both configurations but with different values.

  Let us consider the case (i) first. Since the initial global stores are σ-equivalent, $\mathfrak{G} \equiv_\sigma \mathfrak{G}'$, and potential updates of the global stores via the transition rule (EnvChange) will update the two stores consistently, it follows that $\mathfrak{G}_i \equiv_\sigma \mathfrak{G}_i'$, where $\mathfrak{G}_i$ (*resp.*, $\mathfrak{G}_i'$) is the global store of $\mathfrak{C}_i$ (*resp.*, $\mathfrak{C}_i'$). Thus, *one possibility* for the app id to reach two different states in the configurations $\mathfrak{C}_i$ and $\mathfrak{C}_i'$, such that $\mathfrak{C}_i$ performs the update and $\mathfrak{C}_i'$ does not perform the update (or vice versa), is that it passed by a conditional statement that was evaluated differently in the two cases. Since $\mathfrak{G}_i \equiv_\sigma \mathfrak{G}_i'$, it follows that the guard of the conditional involved some high-level services which are the only ones that can differ in the two global stores. However, this situation is prevented by our type system as the typing rule (IfElse) ensures that we cannot have low-level updates in the two branches of the conditional, after the evaluation of a high-level guard (the high-level type is pushed into the pc of the branches and, eventually, to the low-level update). The *other possibility* is that there was a conditional statement that was evaluated differently in the two configurations: one branch diverges (without performing any low update) while the other does not. Then, after the conditional we have a low-level update. But in this case the two configurations are bisimilar, by definition of our termination-insensitive bisimulation.

Let us consider now the case (ii), *i.e.* when the low-level writing on x is possible in both configurations $\mathfrak{C}_i$ and $\mathfrak{C}'_i$ but with different values v and v', respectively. This situation would be possible when the writing on x in the app id is preceded by an assignment $\mathtt{x} \leftarrow \mathtt{e}$ and the evaluations of e in the two configurations leads to v and v', respectively, with $v \neq v'$. Since $\mathfrak{G}_i \equiv_\sigma \mathfrak{G}'_i$ it follows that the expression e must have a high-level security type, that is, $\Gamma \vdash \mathtt{e} : \delta$, with $\sigma \prec \delta$.

A similar situation would be possible also if there was a conditional whose guard is typed at high-level and the service x is assigned in both branches of the conditional (with different values v and v'). Here, x would take different values even if the assignments involve only low-level expressions, since the guard has type greater than σ, similarly to what happens in case (i). However, our type system would type x as $\delta \succ \sigma$ and, via the typing rule (Update), it does not admit low-level writings on x as $\Sigma(\mathtt{x}) \preccurlyeq \sigma \prec \delta = \Gamma(\mathtt{x})$. Thus, also this case is not admissible.

As it does not exist a distinguishing action α, it follows that the original configurations $\mathfrak{C}$ and $\mathfrak{C}'$ must be bisimilar, with $H_\sigma = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\}$. □

B. Further examples of IoT apps

In the following, we provide further examples of apps that can be modeled in our calculus CaITApp. These examples are inspired by existing real-world apps, also studied in the literature.

Example 9. Consider an app with the following specification. When the user enters a given geographical area the app sends an e-mail saying "entering area". Similarly, when the user exits the given area, the app sends an e-mail saying "exiting area". Suppose a services gps reporting the current position of the user in terms of gps coordinates: gps.Lat and gps.Long. Suppose also a service emailA, sending e-mails to a given address. The geographical area is given by a center, with coordinates centerLat and centerLong, and a radius called side. The area is the square with edges $\langle \mathsf{centerLat} \pm \mathsf{side}, \mathsf{centerLong} \pm \mathsf{side} \rangle$. The app is:

$$\mathtt{Area}[\,\mathtt{gps}^{\mathsf{R}}; \mathtt{emailA}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{gps}); \mathsf{pld9}\,]$$

where

$$\begin{aligned}
\mathsf{pld9} \stackrel{\mathsf{def}}{=}\ & \mathsf{if}\ (\mathsf{ExitArea})\ \mathsf{then}\ \{ \\
& \quad \mathtt{emailA} \leftarrow \mathit{goingOut}; \mathsf{update}(\mathtt{emailA}) \\
& \} \ \mathsf{else}\ \{ \\
& \quad \mathsf{if}\ (\mathsf{EnterArea})\ \mathsf{then}\ \{ \\
& \quad\quad \mathtt{emailA} \leftarrow \mathit{goingIn}; \mathsf{update}(\mathtt{emailA}) \\
& \quad \} \\
& \}; \\
& \mathtt{gps.Lat} \leftarrow \mathsf{read}(\mathtt{gps.Lat}); \\
& \mathtt{gps.Long} \leftarrow \mathsf{read}(\mathtt{gps.Long}); \\
& \mathbb{X}
\end{aligned}$$

where ExitArea is a macro for the boolean test:

$$\begin{aligned}
& \mathtt{gps.Lat} - \mathsf{centerLat} \leq \mathsf{side} \wedge \\
& \mathtt{gps.Long} - \mathsf{centerLong} \leq \mathsf{side} \wedge \\
& \mathsf{read}(\mathtt{gps.Lat}) - \mathsf{centerLat} > \mathsf{side} \wedge \\
& \mathsf{read}(\mathtt{gps.Long}) - \mathsf{centerLong} > \mathsf{side}
\end{aligned}$$

and EnterArea is a macro for the boolean test:

$$\begin{aligned}
& \mathtt{gps.Lat} - \mathsf{centerLat} > \mathsf{side} \wedge \\
& \mathtt{gps.Long} - \mathsf{centerLong} > \mathsf{side} \wedge \\
& \mathsf{read}(\mathtt{gps.Lat}) - \mathsf{centerLat} \leq \mathsf{side} \wedge \\
& \mathsf{read}(\mathtt{gps.Long}) - \mathsf{centerLong} \leq \mathsf{side}
\end{aligned}$$

Example 10. Consider following app. When the lights are turned on the app should brew coffee and set the heater to 22 degrees. Suppose to have three services: ligth, managing the lights, coffeeM, managing the coffee machine, and heater, managing the heater. The app is formalized as follows:

$$\mathtt{Welcome}[\,\mathtt{light}^{\mathsf{R}}; \mathtt{coffeeM}^{\mathsf{W}}; \mathtt{heater}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{light}); \mathsf{P}_{10}\,]$$

where

$$\begin{aligned}
\mathsf{P}_{10} \stackrel{\mathsf{def}}{=}\ & \mathtt{lights} \leftarrow \mathsf{read}(\mathtt{lights}); \\
& \quad \mathsf{if}\ (\mathtt{lights} = \mathit{On})\ \mathsf{then}\ \{ \\
& \quad\quad \mathtt{coffeeM} \leftarrow \mathit{doCoffee}; \\
& \quad\quad \mathtt{heater} \leftarrow 22; \\
& \quad\quad \mathsf{update}(\mathtt{coffeeM}, \mathtt{heater}) \\
& \quad \}; \mathbb{X}
\end{aligned}$$

Example 11. Imagine to have the following app. If a water leak is detected, then the app should shut off the main water supply valve. We assume two services: leakDetect, reporting whether there is a water leak or not, and mainValve, managing the main water valve. The app is formalized as follows:

$$\mathtt{Leak}[\mathtt{leakDetect}^{\mathsf{R}}; \mathtt{mainValve}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{leakDetect}); \mathsf{P}_{11}]$$

where

$$\begin{aligned}
\mathsf{P}_{11} \stackrel{\mathsf{def}}{=}\ & \mathsf{if}\ (\mathsf{read}(\mathtt{leakDetect}) = \mathit{yes})\ \mathsf{then}\ \{ \\
& \quad \mathtt{mainValve} \leftarrow \mathit{Off}; \\
& \quad \mathsf{update}(\mathtt{mainValve}) \\
& \}; \mathbb{X}
\end{aligned}$$

Example 12. Imagine to have an Lamp1, such that if the floor lamp is turned on and the home is in *sleep* mode, then the app should turn off the lamp after 5 minutes. We assume three services: flamp, managing the floor lamp, homeMode, reporting the home mode, and alarm, controlling the burglar alarm. The app is formalized as follows:

$$\mathtt{Lamp1}[\mathtt{flamp}^{\mathsf{RW}}; \mathtt{homeMode}^{\mathsf{R}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{flamp;homeMode}); \mathsf{P}_{12}\,]$$

where

$$\mathsf{P}_{12} \stackrel{\text{def}}{=} \texttt{flamp} \leftarrow \mathsf{read}(\texttt{flamp});$$
$$\mathsf{if}\ (\texttt{flamp.state} = \mathit{On} \wedge \texttt{homeMode} = \mathit{sleep})\ \mathsf{then}\ \{$$
$$\texttt{flamp.ctrl} \leftarrow \mathit{OffDelay5};$$
$$\mathsf{update}(\texttt{flamp})$$
$$\};\mathbb{X}$$

Example 13. Imagine to have a second app, Lamp2, such that if the floor lamp is turned on, and some object is moving, and it is midnight, and the lamp is tunred on for more than 10 minutes, then the burglar alarm should be fired. We assume three services: flamp, managing the floor lamp, motion, reporting whether something is moving or not, and time, reporting the current time. The app is formalized as follows:

$$\texttt{Lamp2}[\,\mathsf{D}_2 \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\texttt{flamp};\texttt{motion};\texttt{time});\mathsf{P}_{13}\,]$$

where
$\mathsf{D}_2 \stackrel{\text{def}}{=} \texttt{flamp}^{\mathsf{RW}};\texttt{motion}^{\mathsf{R}};\texttt{time}^{\mathsf{R}};\texttt{alarm}^{\mathsf{W}}$, and

$$\mathsf{P}_{13} \stackrel{\text{def}}{=} \texttt{flamp} \leftarrow \mathsf{read}(\texttt{flamp});$$
$$\texttt{motion} \leftarrow \mathsf{read}(\texttt{motion});$$
$$\texttt{time} \leftarrow \mathsf{read}(\texttt{time});$$
$$\mathsf{if}\ (\mathsf{cond})\ \mathsf{then}\ \{$$
$$\mathsf{if}\ (\texttt{flamp.curStateTime} \geq 10)\ \mathsf{then}\ \{$$
$$\texttt{alarm} \leftarrow \mathit{On};$$
$$\mathsf{update}(\texttt{alarm})$$
$$\}$$
$$\};\mathbb{X}$$

where cond is the following boolean test:

$$\texttt{flamp.state} = \mathit{On} \wedge \texttt{motion} = \mathit{yes} \wedge \texttt{time.H} = 0 = \texttt{time.M}.$$

Example 14. Imagine to have an app Forward, forwarding messages from the e-mail address emailA to the e-mail address emailB. The app is formalized as follows:

$$\texttt{Forward}[\,\texttt{emailA}^{\mathsf{R}};\texttt{emailB}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\texttt{emailA});\mathsf{P}_{14}\,]$$

where

$$\mathsf{P}_{14} \stackrel{\text{def}}{=} \texttt{emailA} \leftarrow \mathsf{read}(\texttt{emailA});$$
$$\texttt{emailB} \leftarrow \texttt{emailA};$$
$$\mathsf{update}(\texttt{emailB});\mathbb{X}$$

Let us now comment on potential interactions and or interferences between the apps defined above.

Semantic vs. syntax-based interactions: According to Definition 3, the apps Welcome and Leak do not interact with each other, since for every $\mathfrak{G} \in \mathbb{S}$ we have:

- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{Welcome} \parallel \texttt{Leak} \approx_{H_w} \texttt{Leak}$
- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{Welcome} \parallel \texttt{Leak} \approx_{H_l} \texttt{Welcome}$

where $H_w = \mathfrak{upd}(\texttt{Welcome})$ and $H_l = \mathfrak{upd}(\texttt{Leak})$.

However, if we add also the app SmokeAlarm, defined in Example 2, then we do have an interaction. In fact, there exists $\mathfrak{G} \in \mathbb{S}$ such that:

$$\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{SmokeAlarm} \parallel \texttt{Welcome} \parallel \texttt{Leak} \not\approx_{H_s} \texttt{Welcome} \parallel \texttt{Leak}$$

where $H_s = \mathfrak{upd}(\texttt{SmokeAlarm})$. Here, the app SmokeAlarm may interact with the compound system Welcome $\parallel$ Leak. Intuitively, when some smoke is detected the app SmokeAlarm turns the lights on. As a consequence, the app Welcome is triggered: the app brews a coffee and set the heater to 22 degrees.

According to Definition 6, this interaction can be captured syntactically as: SmokeAlarm $\not\rightharpoonup$ Welcome $\parallel$ Leak does not hold because $\mathfrak{act}(\texttt{SmokeAlarm}) = \{\texttt{alarm}, \texttt{lights}\}$, $\mathfrak{trg}(\texttt{Welcome} \parallel \texttt{Leak}) = \{\texttt{lights}, \texttt{leakDetect}\}$, and hence $\mathfrak{act}(\texttt{SmokeAlarm})) \cap \mathfrak{trg}(\texttt{Welcome} \parallel \texttt{Leak}) \neq \varnothing$.

Interactions under dependencies: According to Definition 3, the app Lamp1 is noninteracting with the app Lamp2, even if there is an obvious interplay between them. This lack in Definition 3 is due to the fact we cannot express that the action *OffDelay5* turns flamp.state to *Off* after 5 minutes. This can be fixed by using a dependency policy $\Delta \stackrel{\text{def}}{=} \{(\texttt{flamp.ctrl}, \texttt{flamp.state})\}$. In fact, according to Definition 7, there exists $\mathfrak{G} \in \mathbb{S}$ such that:

$$\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{Lamp1} \parallel \texttt{Lamp2} \stackrel{\Delta}{\not\approx}_{H_1} \texttt{Lamp2}$$

with $H_1 \stackrel{\text{def}}{=} \{\texttt{Lamp1:flamp.ctrl!}v \in \mathcal{A} \mid v \in \mathsf{Value}\}$. Intuitively, if the app Lamp1 turns off the lights then the app Lamp2 cannot be triggered. Again, this interaction can be captured syntactically, via Definition 8, as Lamp1 $stackrel\Delta\not\rightharpoonup$ Lamp2 does not hold because $\{\texttt{flamp.ctrl}, \texttt{flamp.state}\} \subseteq \mathfrak{clo}(\Delta, \mathfrak{act}(\texttt{Lamp1}))$, $\{\texttt{flamp.state}\} \subseteq \mathfrak{trg}(\texttt{Lamp2})$, and hence $\mathfrak{clo}(\Delta, \mathfrak{act}(\texttt{Lamp1})) \cap \mathfrak{trg}(\texttt{Lamp2}) \neq \varnothing$.

Interference and noninterference: Suppose that the email account A, associated to the address emailA, is confidential, *i.e.*, $\Sigma(\texttt{emailA}) = \mathsf{H}$, while the email account B, associated to the address emailB, is public, *i.e.*, $\Sigma(\texttt{emailB}) = \mathsf{L}$. For instance, emails associated to the account A can be red only by one, privileged, administrator while emails associated to the account B can be freely read by any user.

Clearly, the system $\mathsf{S} \stackrel{\text{def}}{=} \texttt{Area} \parallel \texttt{Forward}$ is not secure as we have a confidentiality leak from emailA to emailB. Indeed, it is not hard to find two L-equivalent global stores $\mathfrak{G}$ and $\mathfrak{G}'$ such that $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \not\approx^{\mathsf{ti}}_{H_\mathsf{L}} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$. Again, we can capture this interference by syntactic means: the system S cannot be typed by our security type system because the process P_{14} running within the applet Forward is not typable. In fact, when building the derivation tree for the typing, we have to type the assignment $\texttt{emailB} \leftarrow \texttt{emailA}$ as $\mathsf{L} \vdash \Gamma_1\{\texttt{emailB} \leftarrow \texttt{emailA}\}\Gamma_2$, for $\Gamma_1 = [\texttt{emailA} \mapsto \mathsf{H}, \texttt{emailB} \mapsto \mathsf{L}, \ldots]$ and $\Gamma_2 = [\texttt{emailA} \mapsto \mathsf{H}, \texttt{emailB} \mapsto \mathsf{H}, \ldots]$. But then, the subsequent command update(emailB) cannot be typed, since the current typing of emailB is H while the initial typing of emailB is $\Sigma(\texttt{emailB}) = \mathsf{L}$. Thus, the app Forward is not typable and, in turn, the parallel composition S is not typable.