Friendly Fire: Cross-App Interactions in IoT Platforms

MUSARD BALLIU, KTH Royal Institute of Technology, Sweden
MASSIMO MERRO and MICHELE PASQUA, University of Verona, Italy
MIKHAIL SHCHERBAKOV, KTH Royal Institute of Technology, Sweden

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 computations 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 targeting the security and safety of IoT apps. To demonstrate the usefulness of our framework, we define and implement static analyses for enforcing cross-app security and safety, and prove them sound with respect to our semantic conditions. We also leverage real-world apps to validate the practical benefits of our tools based on the proposed enforcement mechanisms.

CCS Concepts: • **Software and its engineering** → *Software safety*; • **Security and privacy** → **Formal security models**; **Information flow control**.

Additional Key Words and Phrases: Cloud-based IoT platform, IoT application security, Cross-app interference

ACM Reference Format:
Musard Balliu, Massimo Merro, Michele Pasqua, and Mikhail Shcherbakov. 2021. Friendly Fire: Cross-App Interactions in IoT Platforms. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (January 2021), 39 pages. https://doi.org/10.1145/3444963

1 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

Authors' addresses: Musard Balliu, musard@kth.se, KTH Royal Institute of Technology, Lindstedtsvägen 5, Stockholm, Sweden, SE-100 44; Massimo Merro, massimo.merro@univr.it; Michele Pasqua, michele.pasqua@univr.it, University of Verona, Strada le grazie 15, Verona, Italy, 37134; Mikhail Shcherbakov, mshc@kth.se, KTH Royal Institute of Technology, Lindstedtsvägen 5, Stockholm, Sweden, SE-100 44.

2471-2566/2021/1-ART1 $15.00
https://doi.org/10.1145/3444963

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 [32] (If This Then That), Zapier [53], and Microsoft Power Automate [39] support thousands of smart devices and services with millions of users running billions 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 [42]) and the smart window (e.g., SmartThings [48]). 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 [25], unexpected information leaks by seemingly harmless apps [49], and sensitive information disclosure by malicious apps [8, 16]. To protect the users against these attacks, defensive mechanism rely on fine-grained access control and capabilities, decentralization [26] or static [9, 16] and dynamic [7, 8] information-flow analysis.

A more subtle vulnerability concerns the unintended or malicious interaction between different apps running on behalf on the same user [18–20, 23, 49]. 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 a 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 [18–20, 23, 43, 49]. These mechanisms perform inter-application program analysis to track dependencies, and (manual or automated) natural 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 between 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? (v) Can we build tool support to help users validating the security and safety issues with cross-app interactions?

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. 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 [18–20, 23, 49], 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* [28], 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. We demonstrate the feasibility of our approach by implementing our enforcement mechanism in a tool prototype. We use our tool in an empirical study analyzing 20,000 unique IFTTT apps with respect to safety of cross-app interactions.

Further, we focus on security 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 [45], 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 security type system by Hunt and Sands [31] for our concurrent IoT setting, and prove it sound for our security condition. The type system also implements a declassification mechanism to allow controlled release of confidential information. Interestingly, the nature of IoT apps allows for a unified treatment of the *Where* and *What* dimensions of declassification [47]. We develop a prototype implementing the proposed type system, dealing also with declassification. The prototype serves to demonstrate that our security type system can handle real-world apps, previously translated in our calculus.

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 2).
- 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 3).
- We extend our framework to accommodate implicit app interactions in order to tackle the challenge of semantic false negatives (Section 3.3).
- We propose a flow-sensitive security types system, enforcing flexible information-flow policies in a system of IoT apps running concurrently (Section 4).
- We implement and validate our enforcement mechanisms for real-world IoT apps (Section 5).

Full proofs of our results can be found in the Appendix.

2 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 IoT apps running concurrently on an IoT platform, and accessing Internet-connected services and devices on behalf of a 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 has its own *local store*, i.e., local view, which may get updated whenever the execution of the app is triggered by a change in the global store.

We start the description of CaITApp with some preliminary notations. We use letters $\mathsf{x}, \mathsf{y}, \mathsf{z} \in \mathsf{Service}$ 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, strings, etc. We assume two special values: $\bot$ and $*$. The former represents an undefined value, while the latter is a placeholder that can be replaced with "any value".

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

$$\begin{array}{lcll} \mathsf{Sys} \ni \mathsf{S} & ::= & \mathsf{S} \parallel \mathsf{S} & \int \text{ parallel composition} \\ & | & \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,] & \int \text{ app} \end{array}$$

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 a process P.

The syntax for *service declarations* is the following:

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

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

The syntax of *processes* describing the code running in an IoT app is the following:

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

We comment on a few interesting constructs. $\mathsf{listen}(\mathsf{L})$ denotes an app listening on a list 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. The formal definition of L is the following:

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

The construct $\mathsf{x} \leftarrow \mathsf{e}$ sets the local variable x (the local view of the global service with the same name x) to the value obtained by the evaluation of an expression e. Expressions e consist of basic values, readings of local variables y, denoted by y, and readings of global variables y, denoted by $\mathsf{read}(\mathsf{y})$, using standard operators. Thus, in the assignment $\mathsf{x} \leftarrow \mathsf{read}(\mathsf{x}) + \mathsf{y}$ the local copy of the service variable x is updated with the sum of the up-to-date value of the global service x (read from the cloud) and the value of the local copy of the service y. The construct $\mathsf{update}(\mathsf{x})$ updates the

value of the service x in the global store with its current value in the local store. $\mathsf{fix}\,\mathbb{X} \bullet \mathsf{P}$ is the standard construct to denote recursion.

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

Actually, the syntax proposed for modeling our apps is a bit too permissive for our intentions. We could rule out ill-formed apps with a simple type system. 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 hold:

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

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

Hereafter, we will always work with well-formed systems. We write $\mathtt{update}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ for the sequential update $\mathtt{update}(\mathsf{x}_1); \mathtt{update}(\mathsf{x}_2); \ldots; \mathtt{update}(\mathsf{x}_n)$ of the global variables $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n$. We also write if b then {P} for if b then {P} else {skip}.

Let us provide two simple examples describing how to model IoT apps in CaITApp.

Example 1. Consider the following two apps. The app Tw2Fb reposts on Facebook any messages posted on Twitter. Similarly, the app Fb2Ld publishes a post on LinkedIn whenever there is a new post on Facebook. We can use three logical services, tw for Twitter, fb for Facebook, and ld for LinkedIn, to formalize the two apps in our language:

$$\mathtt{Tw2Fb}[\,\mathsf{tw}^{\mathsf{R}}; \mathsf{fb}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen(tw)}; \mathsf{tw} \leftarrow \mathsf{read(tw)}; \mathsf{fb} \leftarrow \mathsf{tw}; \mathsf{update(fb)}; \mathbb{X}\,]$$
$$\mathtt{Fb2Ld}[\,\mathsf{fb}^{\mathsf{R}}; \mathsf{ld}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen(fb)}; \mathsf{fb} \leftarrow \mathsf{read(fb)}; \mathsf{ld} \leftarrow \mathsf{fb}; \mathsf{update(ld)}; \mathbb{X}\,]$$

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 two apps are formalized in our language as follows:

$$\mathtt{SmokeAlarm}[\,\mathsf{smoke}^{\mathsf{R}}; \mathtt{alarm}^{\mathsf{W}}; \mathtt{lights}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen(smoke)}; \mathsf{Pld3}\,]$$
$$\mathtt{Sprinks}[\,\mathsf{heat}^{\mathsf{R}}; \mathsf{waterV}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen(heat)}; \mathsf{Pld4}\,]$$

$$\begin{aligned}
\mathsf{Pld3} \stackrel{\text{def}}{=}\ & \mathsf{smoke} \leftarrow \mathsf{read(smoke)};\\
& \mathsf{if}\ (\mathsf{smoke} = \mathit{yes})\ \mathsf{then}\ \{\\
& \quad \mathtt{alarm} \leftarrow \mathit{On};\\
& \quad \mathtt{lights} \leftarrow \mathit{On};\\
& \quad \mathtt{update(alarm, lights)}\\
& \};\mathbb{X}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Pld4} \stackrel{\text{def}}{=}\ & \mathsf{heat} \leftarrow \mathsf{read(heat)};\\
& \mathsf{if}\ (\mathsf{heat} \geq 45)\ \mathsf{then}\ \{\\
& \quad \mathtt{waterV} \leftarrow \mathit{Open};\\
& \quad \mathtt{update(waterV)}\\
& \};\mathbb{X}
\end{aligned}$$

Table 1. Labeled transition semantics for process configurations.

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

$$\text{(SetLocal)}\ \frac{[\![\mathsf{e}]\!](\mathfrak{G}, \phi) = v}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{x} \leftarrow \mathsf{e} \xrightarrow{\tau}_{\triangleright} \langle \mathfrak{G}, \phi[\mathsf{x} \mapsto v] \rangle \triangleright \mathsf{skip}} \qquad \text{(Update)}\ \frac{\mathfrak{G}(\mathsf{x}) \neq \phi(\mathsf{x}) \qquad \phi(\mathsf{x}) = v}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{update(x)} \xrightarrow{\mathsf{x}!v}_{\triangleright} \langle \mathfrak{G}[\mathsf{x} \mapsto v], \phi \rangle \triangleright \mathsf{skip}}$$

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

$$\text{(IfTrue)}\ \frac{[\![\mathsf{b}]\!](\mathfrak{G}, \phi) = \mathbf{tt}}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{if\ b\ then\ \{P_1\}\ else\ \{P_2\}} \xrightarrow{\tau}_{\triangleright} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_1} \qquad \text{(SeqSkip)}\ \frac{-}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{skip; P} \xrightarrow{\tau}_{\triangleright} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}}$$

$$\text{(IfFalse)}\ \frac{[\![\mathsf{b}]\!](\mathfrak{G}, \phi) = \mathbf{ff}}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{if\ b\ then\ \{P_1\}\ else\ \{P_2\}} \xrightarrow{\tau}_{\triangleright} \langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_2} \qquad \text{(Seq)}\ \frac{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda}_{\triangleright} \langle \mathfrak{G}', \phi' \rangle \triangleright \mathsf{P}_1'}{\langle \mathfrak{G}, \phi \rangle \triangleright \mathsf{P}_1; \mathsf{P}_2 \xrightarrow{\lambda}_{\triangleright} \langle \mathfrak{G}', \phi' \rangle \triangleright \mathsf{P}_1'; \mathsf{P}_2}$$

2.1 Labeled Transition Semantics

IoT apps are simple reactive programs interacting with physical devices and services. They can be accessed only via APIs to *cloud platform*, which we call *global store* and denote by $\mathfrak{G} \in \mathbb{S}$, where $\mathbb{S} \stackrel{\text{def}}{=} \mathsf{Service} \to \mathsf{Value} \cup \{\bot\}$ is the set of all total functions from services to values (sometimes, we will use the following notation: $\mathbb{S}_\bot \stackrel{\text{def}}{=} \{\mathfrak{G} \in \mathbb{S} \mid \forall \mathsf{x} \in \mathrm{dom}(\mathfrak{G})\,.\,\mathfrak{G}(\mathsf{x}) \neq \bot\}$). 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 app id consists of all and only those services declared in D. Changes in the global store are shared with all 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, our labeled transition semantics has two kinds of transitions: one for processes and one for systems.

In Table 1 we provide the transition rules for *process configurations* of the form

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

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 $\lambda \in \mathcal{L}$ range over *non-observable* τ-actions and *observable* modifications (writings) of a global service x, written $\mathsf{x}!v$, respectively. In the following, we will write $\mathfrak{C} \xrightarrow{\lambda}_{\triangleright} \mathfrak{C}'$ to denote a transition between process configurations belonging to the set Pconf of system configurations.

We assume a standard *evaluation semantics* for expressions $[\![\mathsf{e}]\!] \in \mathbb{S} \times \mathbb{S} \to \mathsf{Value} \cup \{\bot\}$, inductively defined on the structure of e. We omit the details of its definition. We write $[\![\mathsf{x}]\!](\mathfrak{G}, \phi) \stackrel{\text{def}}{=} \phi(\mathsf{x})$ for the value of a local service in the local store, and $[\![\mathsf{read(x)}]\!](\mathfrak{G}, \phi) \stackrel{\text{def}}{=} \mathfrak{G}(\mathsf{x})$ for the value of a global service in the global store. Observe that a service may be undefined in a given store.

We now discuss a few interesting rules in Table 1. The construct listen(L) is a blocking operator waiting for changes in the cloud on (at least one of) the services contained in L. The semantics of the listen(L) operator is formalized by means of the rule (StopListening) which is fired whenever one of the changes mentioned before occur. The rule (SetLocal) updates 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 update(x). The rule (Update) modifies the value of the service x in the global store with the value v recorded in the local store, yielding an observable action $\mathsf{x}!v$. Whenever these two values

Table 2. Labeled transition semantics for Systems.

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

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

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

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

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

coincide, no observable changes occur in the cloud, and we use the rule (SkipUpdate) to yield a τ-action modeling a non-observable computational step. The remaining rules are straightforward.

In Table 2 we provide the transition rules for *system configurations* of the form

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

where $\mathfrak{G} \in \mathbb{S}$ denotes the global store, whereas $\mathfrak{L} \in \mathcal{I} \to \mathbb{S}$ is a mapping from an app identifier to its local store. The labels $\alpha \in \mathcal{A}$ range over: *non-observable* τ-actions, *observable* modifications (writings) made by the applet `id` on a global service `x`, written $\mathtt{id}:\mathtt{x}!v$, and *observable* changes on a global service `x` made by the external environment, written $\mathtt{x}?v$, 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 system configurations.

We now comment on the transition rules in Table 2. The rules (App) and (AppUpdate) 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 (EnvChange) models changes in the cloud made by the external environment and affecting all apps. Thus, such action is not triggered by some app of the system, but it can be fired nondeterministically at any moment. The rules (ParLeft) and (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 (ParLeft) and (ParRight)).

3 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 common services in a way that the state reached by those services is 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 in A may affect services in the cloud whose changes may subsequently

enable triggers in B. These triggers of B would not have been fired if A did not modify the state of its services in the cloud.

3.1 Semantic Characterization of Safe Cross-app Interactions

In this section, we provide a semantic characterization of safe cross-app interaction based on the 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. More formally, along the lines of Focardi and Martinelli's *Generalized Non Deducibility on Composition (GNDC)* [28], 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 non-genuine 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 S $\parallel$ R might have non-genuine 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 S $\parallel$ R because they are prevented by S; this is a violation of the *availability* of the system R.

In order to formalize these concepts, we define a slight generalization of the *weak asynchronous bisimulation* [3] introduced for the asynchronous fragment of the π-calculus [29, 38]. For that purpose, we adopt a standard notation for *weak transitions*: we write $\stackrel{\tau}{\rightarrow}{}^{*}$ for the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}$, whereas $\stackrel{\hat{\alpha}}{\Rightarrow}$ denotes $\stackrel{\tau}{\rightarrow}{}^{*}$ if $\alpha = \tau$, and $\stackrel{\tau}{\rightarrow}{}^{*}\stackrel{\alpha}{\rightarrow}\stackrel{\tau}{\rightarrow}{}^{*}$ otherwise. Notice that in weak asynchronous bisimulation input actions are made non-observable because in an asynchronous setting the observer cannot directly observe them. Here, we intend to make non-observable (i.e., to hide) modifications enabled by the interacting system in the cloud.

Consider a set H of *hidden actions*, $H \subseteq \mathcal{A} \setminus \{\tau\}$. Then, the following bisimulation compares two system configurations by observing all possible actions except for those 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 $\mathfrak{C}_1 \stackrel{\alpha}{\rightarrow} \mathfrak{C}_1'$ we have the following:

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

We say that two system configurations $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are *hiding bisimilar* with respect to 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 notion of weak bisimulation.

In the following, 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. Our 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 write $\mathfrak{L}_\perp$ for the function $\lambda \mathtt{id} \,.\, \lambda \mathtt{x} \,.\, \perp$ defining the initial local environments of all apps for which no services are 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 if for any $\mathfrak{G} \in \mathbb{S}$ we have: $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathsf{S}} \mathsf{R}$, 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 it additionally holds that: for any $\mathfrak{G} \in \mathbb{S}$, $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathsf{R}} \mathsf{S}$, for $H_\mathsf{R} \stackrel{\text{def}}{=} \mathfrak{upd}(\mathsf{R})$.

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

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

for $H = \{\mathtt{Tw2Fb{:}tw}!v \mid v \in \mathsf{Value}\}$. We can see that, in the compound system on the left, the app `Fb2Ld` may perform a writing on LinkedIn which cannot occur if the app is running in isolation.

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

- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathtt{SmokeAlarm} \parallel \mathtt{Sprinks} \approx_{H_1} \mathtt{SmokeAlarm}$, where $H_1 = \{\mathtt{Sprinks{:}waterValve}!v \mid v \in \mathsf{Value}\}$;
- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathtt{SmokeAlarm} \parallel \mathtt{Sprinks} \approx_{H_2} \mathtt{Sprinks}$, $H_2 = \{\mathtt{SmokeAlarm{:}alarm}!v, \mathtt{SmokeAlarm{:}lights}!v\}$ and $v \in \mathsf{Value}$.

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 noninteracting even if they write on the same services, provided that no causalities exist among the two writings.

Example 5. Consider an app `SimPres` that turn on the lights for 10 minutes every half an hour to simulates the presence of the user at home during the night. Consider a second app `eSaver` turning off lights during the day to save energy whenever there is no motion for at least 5 minutes.

$$\mathtt{SimPres}[\,\mathtt{user}^{\mathsf{R}}; \mathtt{timeH}^{\mathsf{R}}; \mathtt{timeM}^{\mathsf{R}}; \mathtt{lights}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{user}; \mathtt{time}); \mathsf{Pld5}\,]$$
$$\mathtt{eSaver}[\,\mathtt{none}^{\mathsf{R}}; \mathtt{timeH}^{\mathsf{R}}; \mathtt{lights}^{\mathsf{RW}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathtt{none}; \mathtt{lights}); \mathsf{Pld6}\,]$$

$\mathsf{Pld5} \stackrel{\text{def}}{=}$
```
user ← read(user);
if (0 < read(timeH) < 7 ∧ user = away) then {
  if (read(timeM) = 30) then {
    lights ← On10minsOff; update(lights)
  }
};𝕏
```

$\mathsf{Pld6} \stackrel{\text{def}}{=}$
```
none ← read(none);
if (8 < read(timeH) < 18) then {
  if (none ≥ 5 ∧ lights = On) then {
    lights ← Off; update(lights)
  }
};𝕏
```

Notice that there is no interaction between these two apps, although 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 $H_1 = \{\texttt{SimPres:lights!}v \mid v \in \mathsf{Value}\}$;
- $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{SimPres} \parallel \texttt{eSaver} \approx_{H_2} \texttt{SimPres}$, with $H_2 = \{\texttt{eSaver:lights!}v \mid v \in \mathsf{Value}\}$.

3.2 A Simple Proof Technique for Safe Cross-app Interactions

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 $\mathfrak{act}(\texttt{id}) \stackrel{\text{def}}{=} \{\mathsf{x} \in \mathsf{Service} \mid \mathsf{x}^{\mathsf{W}} \in \mathsf{D}\}$. 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 $\mathfrak{act}(\mathsf{S}) \stackrel{\text{def}}{=} \bigcup_{1 \le i \le n} \mathfrak{act}(\texttt{id}_i)$.

Similarly, the triggers of an app $\texttt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ are given by the services on which the app currently listens or makes 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 $\mathfrak{trg}(\texttt{id}) \stackrel{\text{def}}{=} \{\mathsf{x} \in \mathsf{Service} \mid \mathsf{x}^{\mathsf{R}} \in \mathsf{D}\}$. 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 $\mathfrak{trg}(\mathsf{S}) \stackrel{\text{def}}{=} \bigcup_{1 \le i \le n} \mathfrak{trg}(\texttt{id}_i)$.

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 execution of S does 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 $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{trg}(\mathsf{R}) = \varnothing$. More generally, we say that the two systems S and R are *syntactically noninteracting with respect to each other*, written $\mathsf{S} \nleftrightarrow \mathsf{R}$, when, in addition, it also holds that $\mathfrak{trg}(\mathsf{S}) \cap \mathfrak{act}(\mathsf{R}) = \varnothing$.

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 $\mathfrak{act}(\texttt{Tw2Fb}) \cap \mathfrak{trg}(\texttt{Fb2Ld}) = \{\texttt{facebook}\} \neq \emptyset$;
- $\texttt{SmokeAlarm} \nleftrightarrow \texttt{Sprinks}$ holds.

Thus, Definition 6 provides an easy-to-verify syntactic condition to check our semantic-based notion of safe-cross interaction formalized in Definition 3.

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.

As we can see, 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. While existing techniques, e.g., model checking [41], can be used to improve the permissiveness of the analysis, our goal is to illustrate how our semantic condition enables formal proofs of soundness of these techniques.

3.3 Implicit Safe-cross App Interactions

We now study the challenge posed by *implicit cross-app interactions* that arises whenever two (physical) services, e.g., temperature and thermostat, are semantically related, though they differ syntactically. In such cases, our semantic-based definition of safe cross-app interaction may consider an interaction as safe, while this is not the case in practice. We propose an extension of our language semantics, as well as both our semantic and syntactic conditions to reason about such cases.

Actually, 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 *implicit interactions*. Consider the example below.

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

$$\mathtt{Win}[\,\mathtt{temp}^{\mathrm{R}};\mathtt{win}^{\mathrm{W}} \bowtie \mathsf{fix}\,\mathbb{X}\bullet \mathsf{listen}(\mathtt{temp});\mathsf{Pld7}\,]$$

where, $\mathsf{Pld7} \stackrel{\text{def}}{=} \mathtt{temp} \leftarrow \mathsf{read}(\mathtt{temp});\mathsf{if}\ (\mathtt{temp} > 22)\ \mathsf{then}\ \{\mathtt{win} \leftarrow \mathit{Open};\mathsf{update}(\mathtt{win})\};\mathbb{X}$.
Now, suppose there is a second app Therm, managing the thermostat of the room, such that when the temperature is below 17 degrees the thermostat is set to increase the temperature by 3 degrees.

$$\mathtt{Therm}[\,\mathtt{temp}^{\mathrm{R}};\mathtt{therm}^{\mathrm{W}} \bowtie \mathsf{fix}\,\mathbb{X}\bullet \mathsf{listen}(\mathtt{temp});\mathsf{Pld8}\,]$$

where, $\mathsf{Pld8} \stackrel{\text{def}}{=} \mathtt{temp} \leftarrow \mathsf{read}(\mathtt{temp});\mathsf{if}\ (\mathtt{temp} < 17)\ \mathsf{then}\ \{\mathtt{therm} \leftarrow \mathit{+3};\mathsf{update}(\mathtt{therm})\};\mathbb{X}$.

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 that room.

However, since this out-of-band information is not considered by our formalization, according to Definition 3 we would have a kind of *semantic false negative* as the app Therm is not directly interacting with the app Win. Formally, for $H = \mathfrak{upd}(\mathtt{Therm}) = \{\mathtt{Therm}{:}\mathtt{therm}!v \mid v \in \mathsf{Value}\}$, we have: $\langle \mathfrak{G}, \mathfrak{Q}_{\perp} \rangle \triangleright \mathtt{Win} \parallel \mathtt{Therm} \approx_H \mathtt{Win}$, for any $\mathfrak{G} \in \mathbb{S}$.

Note that *causal dependence* 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 provide extra information about causal dependence.

However, we can assume that, when designing our system of apps we actually get, out of band, a set of causal dependencies to improve the precision of our analysis ruling out a number of semantic 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 y may be affected by changes occurring at the service 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 with respect to the service 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 x changes on the cloud then the service y can be somehow affected. We represent this abstract

information by means of *nondeterministic updates* assigning to 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 causal dependence 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 1 different from (Update);
- rule (Update) is replaced by the following transition rule:

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

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 semantic false negatives due to implicit interactions: any change on a service x affects any service in the set $\mathfrak{clo}(\Delta, \mathsf{x})$ 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 CaITApp. We say that S is *noninteracting with* R *under* Δ when for any $\mathfrak{G} \in \mathbb{S}$ we have $\langle \mathfrak{G}, \mathfrak{L}_{\perp} \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{S}} \mathsf{R}$, 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 that for any $\mathfrak{G} \in \mathbb{S}$ it holds $\langle \mathfrak{G}, \mathfrak{L}_{\perp} \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \overset{\Delta}{\approx}_{H_\mathsf{R}} \mathsf{S}$, 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 $\mathfrak{clo}(\Delta, \mathfrak{act}(\mathsf{S})) \cap \mathfrak{trg}(\mathsf{R}) = \varnothing$. More generally, we say that the two systems S and R are *syntactically noninteracting with respect to each other under* Δ, written $\mathsf{S} \overset{\Delta}{\nleftrightarrow} \mathsf{R}$, when, in addition, it holds that $\mathfrak{trg}(\mathsf{S}) \cap \mathfrak{clo}(\Delta, \mathfrak{act}(\mathsf{R})) = \varnothing$.

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* 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 $\Delta = \{(\texttt{therm}, \texttt{temp})\}$, 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: $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \texttt{Win} \parallel \texttt{Therm} \not\approx_H \texttt{Win}$, for $H = \mathfrak{upd}(\texttt{Therm}) = \{\texttt{Therm:therm!}v \mid v \in \texttt{Value}\}$.

4 SECURING CROSS-APP INTERACTIONS

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

In this section, we assume a *security policy* $\Sigma \in \text{Service} \to SL$, which associates a *security level* $\sigma \in SL$, taken from some complete lattice $\langle SL, \preccurlyeq, \sqcup, \sqcap, \top, \bot \rangle$, with each service used by our system of apps. As expected, the lattice consists of a set of security levels SL, an ordering relation $\preccurlyeq$, the join $\sqcup$ and meet $\sqcap$ operators, as well as a top security level $\top$ and a bottom security level $\bot$.

For simplicity, in the examples the security levels will be high (H), or *confidential*, and low (L), or *public*, although the theory is developed for a generic complete lattice 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 securely flow from a service x to a service y whenever $\Sigma(\mathsf{x}) \preccurlyeq \Sigma(\mathsf{y})$.

As usual, a security policy Σ 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 $\mathfrak{G} \equiv_{\Sigma,\sigma} \mathfrak{G}'$, if they agree on the values associated to all services with security level lower than, or equal to, σ.

Definition 9 (σ-equivalent global stores)**.** Let $\Sigma \in \text{Service} \to SL$ be a security policy. 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 $\mathfrak{G} \equiv_{\Sigma,\sigma} \mathfrak{G}'$, whenever $\Sigma(\mathsf{x}) \preccurlyeq \sigma$ entails $\mathfrak{G}(\mathsf{x}) = \mathfrak{G}'(\mathsf{x})$, for any $\mathsf{x} \in \text{Service}$.

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 non-interference we consider σ-equivalent stores in $\mathbb{S}_\perp \stackrel{\text{def}}{=} \{\mathfrak{G} \in \mathbb{S} \mid \forall \mathsf{x} \in \text{dom}(\mathfrak{G}) \,.\, \mathfrak{G}(\mathsf{x}) \neq \bot\}$, 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 permissiveness 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 process actions λ and system actions α, according to the cloud services involved. Thus, we define $\Sigma(\mathsf{x}!v) = \Sigma(\mathtt{id}{:}\mathsf{x}!v) = \Sigma(\mathsf{x}?v) = \Sigma(\mathsf{x})$.

However, in order to capture a semantic notion of noninterference that is not sensitive to information leaks due to program termination[1], we propose a modification of our hiding bisimilarity inspired by the *termination-insensitive i-bisimulation* proposed by Demange and Sands [21]. For this purpose, given a set of *non-observable actions* H, we will write $\mathfrak{C}_1 \Uparrow_H$ if and only if $\mathfrak{C}_1 \in D \stackrel{\text{def}}{=} \{\mathfrak{C} : (\exists \alpha \in \mathcal{A}.\, \mathfrak{C} \stackrel{\alpha}{\longrightarrow} \mathfrak{C}') \wedge (\mathfrak{C} \stackrel{\alpha}{\longrightarrow} \mathfrak{C}' \text{ entails } \alpha \in H \wedge \mathfrak{C}' \in D)\}$, that is $\mathfrak{C}_1$ belongs to the set of divergent configurations that can always and only perform either τ-actions or high-level actions.

Furthermore, we refine our LTS semantics as follows. We denote with $\rightarrow_H$ the relation involving any possible τ or high-level action, i.e., $\rightarrow_H \stackrel{\text{def}}{=} \bigcup \left\{ \stackrel{\alpha}{\rightarrow} \;\middle|\; \alpha \in H \right\}$ and with $\Rightarrow_H$ its transitive closure,

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

i.e., $\Rightarrow_H \stackrel{\text{def}}{=} \rightarrow_H^*$. Finally, $\stackrel{\alpha}{\Rightarrow}_H \stackrel{\text{def}}{=} \Rightarrow_H \stackrel{\alpha}{\longrightarrow} \Rightarrow_H$ means that we can perform an arbitrary, possibly empty, sequence of τ- or high-level actions, but at least one α action must be present. In this manner, our notion of weak transition $\Rightarrow_H$ treats high-level actions as non-observable τ-actions.

Definition 10 (Termination-Insensitive Hiding Bisimulation)**.** Given a set of *non-observable actions* actions $H \subseteq \mathcal{A} \cup \{\tau\}$, a 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 $\mathfrak{C}_1 \stackrel{\alpha}{\longrightarrow} \mathfrak{C}_1'$ we have the following:

- if $\alpha \in H$ then $\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \mathcal{R} \mathfrak{C}_2'$
- if $\alpha \notin H$ then
  - either $\mathfrak{C}_2 \stackrel{\alpha}{\Rightarrow}_H \mathfrak{C}_2'$, for some $\mathfrak{C}_2'$ such that $\mathfrak{C}_1' \mathcal{R} \mathfrak{C}_2'$
  - or $\mathfrak{C}_2 \Uparrow_H$.

We say that two system configurations $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are *termination-insensitive hiding bisimilar* with respect to the set of non-observable 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.

In the first clause, as high-level actions are non-observable, different high-level actions can be matched with each other. In the second clause, we deal with low-level actions in a manner similar to what is done by Demange and Sands in their termination-insensitive *i*-bisimulation [21].

Definition 11 (σ-level noninterference)**.** Let $\Sigma \in \mathsf{Service} \to \mathit{SL}$ be a security policy, and $\sigma \in \mathit{SL}$ be a security level. Let S be a system of apps and $\hat{H} = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\}$ the set of actions with clearance greater than σ. We say that S is *σ-level interference-free* whenever:

$$\forall \mathfrak{G}, \mathfrak{G}' \in \mathbb{S}_\perp \,.\, \mathfrak{G} \equiv_{\Sigma,\sigma} \mathfrak{G}' \;\Rightarrow\; \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx_H^{\text{ti}} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}, \text{for } H = \hat{H} \cup \{\tau\}\,.$$

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 `tw` will flow into the public service `ld`. In fact, it is not hard to find two L-equivalent global stores $\mathfrak{G}$ and $\mathfrak{G}'$ such that $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \not\approx_H^{\text{ti}} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$. 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 11 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 security type system* of Hunt and Sands [31], adapted to our setting. In Table 3 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 a security level $\sigma \in \mathit{SL}$ that is higher or different from the pc (i.e., $\sigma \not\preccurlyeq pc$) to services at security level $\sigma' \in \mathit{SL}$ lower or equal to the pc (i.e., $\sigma' \preccurlyeq pc$). Here, pc denotes the usual "program counter" level and serves to eliminate implicit 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, similarly to Hunt and Sands [31], Γ keeps track of the security levels of local variables which hold before the execution of P, whereas Γ' provides us with the security levels of local variables after the execution of P. Again, pc denotes the "program counter" level and the derivation rules ensure that only services with security types not lower than pc may be changed by P. Note that we use Γ also to bind process variables to program counters: in this manner, we ensure that in recursive processes of the form $\mathsf{fix}\,\mathbb{X} \bullet \mathsf{P}$ the security level of recursive calls in P coincides with the security level of the recursive process itself.

Table 3. Security type system.

Typing rules for expressions

$$\text{(Const)}\ \frac{-}{\Gamma \vdash v : \bot} \qquad \text{(Var)}\ \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \text{(Read)}\ \frac{\Sigma(x) = \sigma}{\Gamma \vdash \mathsf{read}(x) : \sigma} \qquad \text{(Expr)}\ \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash e_1\ \mathsf{op}\ e_2 : \sigma_1 \sqcup \sigma_2}$$

Typing rules for processes and subtyping

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

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

$$\text{(IfElse)}\ \frac{\Gamma_1 \vdash b : \sigma \qquad pc \sqcup \sigma \vdash \Gamma_1\{P_1\}\Gamma_2 \qquad pc \sqcup \sigma \vdash \Gamma_1\{P_2\}\Gamma_2}{pc \vdash \Gamma_1\{\mathsf{if}\ b\ \mathsf{then}\ \{P_1\}\ \mathsf{else}\ \{P_2\}\}\Gamma_2} \qquad \text{(Seq)}\ \frac{pc \vdash \Gamma_1\{P_1\}\Gamma_2 \quad pc \vdash \Gamma_2\{P_2\}\Gamma_3}{pc \vdash \Gamma_1\{P_1; P_2\}\Gamma_3}$$

$$\text{(Sub.Proc)}\ \frac{pc' \vdash \Gamma_1'\{P\}\Gamma_2' \qquad pc \preccurlyeq pc' \qquad \Gamma_1 \preccurlyeq \Gamma_1' \qquad \Gamma_2' \preccurlyeq \Gamma_2}{pc \vdash \Gamma_1\{P\}\Gamma_2}$$

Typing rules for systems

$$\text{(App)}\ \frac{pc \vdash \Gamma_1\{P\}\Gamma_2}{pc \vdash \mathtt{id}[\,D \bowtie P\,]} \qquad \text{(Par)}\ \frac{pc \vdash S_1 \qquad pc \vdash S_2}{pc \vdash S_1 \parallel S_2}$$

Here, we wish to remark that, unlike batch-job noninterference models [51], 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 (explicit or implicit) 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 allowed only if x is associated to an "original" security level (given by Σ) not lower than the current security level of the corresponding local variable x (given by Γ) and the security level of the program counter (given by pc), i.e., $pc \sqcup \Gamma(x) \preccurlyeq \Sigma(x)$. 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.

Example 8. Consider a *malicious app* listening on a confidential location service loc and updating a public log service log. For simplicity, we assume that a location is represented as a positive integer. The app implements the following payload Pld:[2]

[2]We recall that we write the code "if b then {P}" as an abbreviation for "if b then {P} else {skip}".

1. $\texttt{loc} \leftarrow \mathsf{read}(\texttt{loc}); \texttt{log} \leftarrow 0;$
2. $\mathsf{fix}\,\mathbb{X} \bullet \mathsf{if}\ (\texttt{loc} > 0)\ \mathsf{then}\ \{$
   $\texttt{log} \leftarrow \texttt{log} + 1; \texttt{loc} \leftarrow \texttt{loc} - 1;$
   $\mathbb{X}\};$
3. $\mathsf{update}(\texttt{log});$

The payload copies the confidential value from variable loc to variable log via an implicit flow and it exfiltrates the value via an update operation update(log) on the public service log. We leverage the typing rules in Table 3 to show that the app is correctly rejected by the type system.

(1) The security policy for services is $\Sigma(\texttt{loc}) = \mathsf{H}$ and $\Sigma(\texttt{log}) = \mathsf{L}$ and the initial policy for variables is set to the lowest security level, i.e., $\Gamma(\texttt{loc}) = \Gamma(\texttt{log}) = \Gamma(\mathbb{X}) = \mathsf{L}$, since these variables contain no confidential information initially. Moreover, we assume the attacker can observe data at security level L, hence $pc = \mathsf{L}$. We now show that $\mathsf{L} \vdash \Gamma \left\{ \mathsf{Pld} \right\} \Gamma'$ cannot typecheck, for any Γ'. We can use rule (Seq) to typecheck the statements in lines (1.–3.) separately.
(2) For the first two assignment statements in line 1., an application of rule (Assign) and rules (Const) and (Read) yields a typing environment Γ_1 such that $\Gamma_1(\texttt{loc}) = \mathsf{H}$ and $\Gamma_1(\texttt{log}) = \mathsf{L}$, while $pc = \mathsf{L}$. Next, we typecheck the statements between lines 2. and 3. using the typing environment Γ_1 and $pc = \mathsf{L}$.
(3) Rule (Fix) requires typechecking the conditional statement under its scope in a security environment Γ_2 such that $\Gamma_2(\mathbb{X}) = pc$, hence $\Gamma_2 = \Gamma_1$ since $pc = \mathsf{L}$. We then use rule (IfElse) to typecheck the conditional statement with $pc = \mathsf{H}$ since $\Gamma_2(\texttt{loc} > 0) = \mathsf{H}$. The typing rule (Assign) will yield $\Gamma_3(\texttt{loc}) = \mathsf{H}$ and $\Gamma_3(\texttt{log}) = \mathsf{H}$. Finally, the type checking will fail when applying rule (Pvar) since $\Gamma_3(\mathbb{X}) = \mathsf{L}$ and $pc = \mathsf{H}$.
(4) Alternatively, we can apply rule (Sub.Proc) prior to applying rule (Fix) in step (3) and raise the program counter level $pc = \mathsf{H}$. Then rule (Fix) requires typechecking the conditional statement in a security environment Γ_2 such that $\Gamma_2(\mathbb{X}) = \mathsf{H}$. At this point, we can apply the same rules as in the previous case and obtain a security environment Γ_3 such that $\Gamma_3(\texttt{loc}) = \Gamma_3(\texttt{log}) = \Gamma_3(\mathbb{X}) = \mathsf{H}$ and $pc = \mathsf{H}$.
(5) Finally, the application of rule (Update) on the statement in line 3. will fail independently of the level of pc, since $\Gamma_3(\texttt{log}) = \mathsf{H}$ and $\Sigma(\texttt{log}) = \mathsf{L}$, and $pc \sqcup \mathsf{H} = \mathsf{H} \not\leq \mathsf{L}$, thus correctly rejecting the program.

Now, everything is in place to show that the security type system in Table 3 is sound with respect to the security condition formalized in Definition 10. The proof can be found in the appendix.

Theorem 3 (Soundness of security types)**.** *Let* $\Sigma \in \mathsf{Service} \to \mathit{SL}$ *be a security policy,* S *be a system of apps, and* $H = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\} \cup \{\tau\}$ *be the set of all possible non-observable system actions at the attacker's security clearance* $\sigma \in \mathit{SL}$*. Let* $\mathfrak{G}_a, \mathfrak{G}_b \in \mathbb{S}_\perp$ *be two arbitrary stores such that* $\mathfrak{G}_a \equiv_\sigma \mathfrak{G}_b$*. If* $\sigma \vdash \mathsf{S}$ *then* $\langle \mathfrak{G}_a, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx^{\mathrm{ti}}_H \langle \mathfrak{G}_b, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$*.*

As we will see in the Section 5, our security type system does not face any permissiveness issues (i.e., false positives) for the apps considered in our benchmarks. Nevertheless, as expected, the type system is not complete, as we can see from the following example.

Example 9. Consider an app with payload

$$\texttt{Lserv} \leftarrow 0; \mathsf{if}\ (\texttt{Hserv} = 5)\ \mathsf{then}\ \{\texttt{Lserv} \leftarrow 0\}\ \mathsf{else}\ \{\mathsf{skip}\}; \mathsf{update}(\texttt{Lserv})$$

where Hserv is considered confidential, i.e., it has type H, and Lserv is considered public, i.e., it has type L. Clearly, this app is noninterferent, since the value of Lserv does not depend on Hserv

(it is always 0 when we perform the update). However, our type system flags this app as insecure, since the type of `Lserv` is H when reaching the update. This is because of the assignment to `Lserv` inside a conditional guard of type H. Since the initial type of `Lserv` is L, the type system cannot apply the typing rule for the update, hence the app is rejected.

In conclusion, we expect our analysis to scale well and produce a minimal false-positive rate for user-automation IoT platforms like IFTTT. In these platforms, the code consist of simple snippets matching the syntax of `CaITApp` closely [8]. 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*.

4.1 Declassification

In many real-world applications, noninterference is a too strong policy: sometimes, controlled release of sensitive information is deliberately allowed. A classic example is a password checking program which compares the actual password with a guessed password to authenticate a user. This program contains a sensitive information flow from the actual password to the output on a public channel, in order to inform a (potentially untrusted) user whether or not the authentication succeeds. The program is usually accepted as secure, since leaking the entire password in this manner is computationally hard. In this setting, this information can be *declassified*, i.e., it can be safely disclosed although it is not part of the flow relation $\preccurlyeq$.

Following the approach of *delimited release* [46], we extend our language with a declassification primitive to support controlled release of sensitive information for IoT services. The new policy targets the *what* dimension of declassification [47]. This is more suitable for our setting since declassification applies to information from initial state of global services. Formally, we introduce a construct $\mathsf{declassify}(\mathsf{e}, \sigma)$ into expressions' syntax. Intuitively, $\mathsf{declassify}(\mathsf{e}, \mathsf{L})$ means that the expression e, potentially containing sensitive data H, can be declassified to the (lower) security level L. Note that this construct is used only in the security type system. The evaluation semantics of $\mathsf{declassify}(\mathsf{e}, \sigma)$ is equal to the evaluation semantics of e, namely, $[\![\mathsf{declassify}(\mathsf{e}, \sigma)]\!](\mathfrak{G}, \phi) = [\![\mathsf{e}]\!](\mathfrak{G}, \phi)$. Furthermore, we assume that declassification constructs cannot be nested, namely, in $\mathsf{declassify}(\mathsf{e}, \sigma)$ the expression e cannot contain other instances of the declassification construct. As we allow information release only at the entry-points of the apps, declassification can only involve variables of global services. This means that service instances under declassification must be bonded by a read construct. For example, $\mathsf{declassify}(\mathsf{read}(\mathsf{x}), \mathsf{L}) + \mathsf{y}$ is a legal expression, while $\mathsf{declassify}(\mathsf{x}, \mathsf{L}) + \mathsf{y}$ is not. We assume that expressions in the scope of a *declassify* construct are legal, which can be enforced easily with simple syntactic checks.

In order to define noninterference up to declassification, we need a notion of global-store equivalence which accounts for the declassified expressions. Intuitively, we need to consider not only stores with equivalent public services but also stores with equivalent declassified expressions.

The evaluation semantics $[\![\mathsf{e}]\!]$ for expressions is parametric on the global and local stores but we do not need the local information of services in order to define the (global) stores equivalence. Hence, we use a new semantics for expressions $(\!|\mathsf{e}|\!)$ ignoring the local value of services, namely it assigns to local variables the undefined value. Formally, the semantic function $(\!|\mathsf{e}|\!) \in \mathbb{S} \to \mathsf{Value}$ is defined as $(\!|\mathsf{e}|\!)\mathfrak{G} \stackrel{\text{def}}{=} [\![\mathsf{e}]\!](\mathfrak{G}, \lambda \mathsf{x} \,.\, \bot)$.

Definition 12 (σ-equivalent stores up to declassification)**.** Let $\Sigma \in \mathsf{Service} \to \mathit{SL}$ be a security policy. Let $\mathfrak{G}, \mathfrak{G}' \in \mathbb{S}$ be two σ-equivalent stores for some security level $\sigma \in \mathit{SL}$. Assume n declassification points, $\mathsf{d} \stackrel{\text{def}}{=} \mathsf{declassify}(\mathsf{e}_1, \sigma_1), \ldots, \mathsf{declassify}(\mathsf{e}_n, \sigma_n)$, for expressions e_i and security levels σ_i, with $1 \leq i \leq n$. We say that $\mathfrak{G}$ and $\mathfrak{G}'$ are *σ-equivalent up to declassification* d, written $\mathfrak{G} \equiv^{\mathsf{d}}_{\Sigma,\sigma} \mathfrak{G}'$, only if $\sigma_i \preccurlyeq \sigma$ entails $(\!|\mathsf{e}_i|\!)\mathfrak{G} = (\!|\mathsf{e}_i|\!)\mathfrak{G}'$, for any $1 \leq i \leq n$.

Now, we are ready to define noninterference up to declassification by reformulating Definition 11 using the declassified version of the global stores equivalence.

Definition 13 (σ-level noninterference up to declassification)**.** Let S be a system of apps and $H \stackrel{\text{def}}{=} \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\}$ be the set of actions with clearance greater than a fixed $\sigma \in SL$. Suppose that S contains within it n declassification points $\mathsf{d} \stackrel{\text{def}}{=} \mathsf{declassify}(\mathsf{e}_1, \sigma_1), \ldots, \mathsf{declassify}(\mathsf{e}_n, \sigma_n)$. We say that S is *$\sigma$-level interference-free up to declassification* d whenever:

$$\forall \mathfrak{G}, \mathfrak{G}' \in \mathbb{S}_\perp \,.\, \mathfrak{G} \equiv^{\mathsf{d}}_{\Sigma,\sigma} \mathfrak{G}' \;\Rightarrow\; \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx^{\mathrm{ti}}_{H} \langle \mathfrak{G}', \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}\,.$$

We remark that Definition 11 implies Definition 13 and for systems without the declassify primitives the two definitions coincide.

In order to check noninterference with declassification we extend our security type system. In particular, we add a typing rule for the declassification construct. In the following rule, $\mathtt{LocVars}(\mathsf{e})$ returns the local occurrences of the services in the expression e, namely all services' occurrences which are not under the scope of a read construct. For instance, $\mathtt{LocVars}(\mathsf{x} + \mathsf{read}(\mathsf{y}))$ returns the singleton $\{\mathsf{x}\}$.

$$(\text{Declassify})\ \frac{\mathtt{LocVars}(\mathsf{e}) = \varnothing}{\Gamma \vdash_D \mathsf{declassify}(\mathsf{e}, \sigma) : \sigma}$$

This rule basically says that the type system will reject any attempt to declassify a local service, which is forbidden by our initial assumptions. Observe that our typing rule for declassification is not subject to *laundering* attacks [46], since variables representing global services are *read-only*. In fact, this invariant in combination with the premise of the rule ensures that no variable in the scope of a *declassify* construct is ever modified in the program.

The other typing rules for expressions, processes and systems remain unchanged. We use the symbol $\vdash_D$ in place of $\vdash$ when we refer to the extended security type system with declassification.

We illustrate the new rule with an example. Consider a public service x, a confidential service y, and a predicate par capturing the parity of a given value. The assignment $\mathsf{x} \leftarrow \mathsf{par}(\mathsf{read}(\mathsf{y}))$ is not safe, since we have a forbidden information flow between y and x. Indeed our type system types the expression $\mathsf{par}(\mathsf{read}(\mathsf{y}))$ as H and it will stop if we try to update x after the assignment. Instead, in the assignment $\mathsf{x} \leftarrow \mathsf{declassify}(\mathsf{par}(\mathsf{read}(\mathsf{y})), \mathsf{L})$ we are able to deduce that there is no forbidden flow between y and x, since $\mathsf{declassify}(\mathsf{par}(\mathsf{read}(\mathsf{y})), \mathsf{L})$ is typed as L.

Example 10. Consider the two following apps PicToDb and DbLogger. The first automatically saves on Dropbox every picture taken by the camera of the user's smartphone. The second logs every event happening to the user's Dropbox account. They are formalized in CaITApp as follows:

$$\mathtt{PicToDb}[\,\mathsf{camera}^{\mathsf{R}};\mathsf{dropbox}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathsf{camera}); \mathsf{Pld9}\,]$$

$$\mathtt{DbLogger}[\,\mathsf{dropbox}^{\mathsf{R}};\mathtt{time}^{\mathsf{R}};\mathsf{dbLog}^{\mathsf{W}} \bowtie \mathsf{fix}\,\mathbb{X} \bullet \mathsf{listen}(\mathsf{dropbox}); \mathsf{Pld10}\,]$$

$$\begin{aligned}
\mathsf{Pld9} \stackrel{\text{def}}{=}\ & \mathsf{camera} \leftarrow \mathsf{read}(\mathsf{camera});\\
& \mathsf{dropbox} \leftarrow \mathsf{camera};\\
& \mathsf{update}(\mathsf{dropbox});\\
& \mathbb{X}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{Pld10} \stackrel{\text{def}}{=}\ & \mathsf{dbLog} \leftarrow \mathsf{read}(\mathsf{dbLog}) + \mathsf{read}(\mathtt{time}) +\\
& \qquad \mathsf{nameOf}(\mathsf{read}(\mathsf{dropbox}));\\
& \mathsf{update}(\mathsf{dbLog});\\
& \mathsf{dropbox} \leftarrow \mathsf{read}(\mathsf{dropbox});\\
& \mathbb{X}
\end{aligned}$$

Suppose that the services camera and dropbox are confidential, while the services dbLog and time are public. Clearly, we have a forbidden information flow between camera and dbLog. Indeed, the

system PicToDb || DbLogger is not noninterferent. Even if we do not want to reveal publicly our pictures, it could be reasonable to disclose just their names. In this case, we can declassify the expression nameOf(read(dropbox)) as follows:

$$\begin{aligned}\mathsf{Pld10}' \stackrel{\text{def}}{=}\ & \mathsf{dbLog} \leftarrow \mathsf{read}(\mathsf{dbLog}) + \mathsf{read}(\mathtt{time}) + \mathsf{declassify}(\mathsf{nameOf}(\mathsf{read}(\mathsf{dropbox})), \mathsf{L});\\ & \mathsf{update}(\mathsf{dbLog}); \mathsf{dropbox} \leftarrow \mathsf{read}(\mathsf{dropbox}); \mathbb{X}\end{aligned}$$

The system PicToDb || DbLogger is now typable with our (extended) security type system and, indeed, it is considered secure.

The following theorem extends the soundness results to systems of apps that support declassification policies. The proof can be found in the appendix.

Theorem 4 (Soundness of security types with declassification)**.** *Let* $\Sigma \in \mathsf{Service} \to \mathit{SL}$ *be a security policy, Let* $H = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\} \cup \{\tau\}$ *be the set of all possible non-observable system actions at the attacker's security clearance* $\sigma \in \mathit{SL}$*. Let* S *be a system of apps containing n declassification points* $\mathsf{d} \stackrel{\text{def}}{=} \mathsf{declassify}(\mathsf{e}_1, \sigma_1), \ldots, \mathsf{declassify}(\mathsf{e}_n, \sigma_n)$*, Let* $\mathfrak{G}_a, \mathfrak{G}_b \in \mathbb{S}_\perp$ *be two arbitrary stores such that* $\mathfrak{G}_a \equiv^{\mathsf{d}}_{\Sigma,\sigma} \mathfrak{G}_b$*. If* $\sigma \vdash_D \mathsf{S}$ *then* $\langle \mathfrak{G}_a, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx^{\mathsf{ti}}_H \langle \mathfrak{G}_b, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$.

5 IMPLEMENTATION AND VALIDATION

This section presents the implementation and validation of our enforcement mechanisms for safety and security of cross-app interactions. We first implement a tool prototype to detect cross-app interactions for a set of user-installed apps and visualize potential violations, using the syntactic conditions given in Definition 6. We use the tool in a large-scale empirical study to analyze about 280,000 IFTTT applets for unsafe cross-app interactions.

We also implement our security type system (Table 3), together with its extension to declassification, in another tool prototype written in Java. We then validate the prototype on a set of real-world IoT apps, taken from the benchmark dataset of Bastys et al. [8] and translated to our language CaITApp. As expected, the prototype does not signal false negatives for our test cases (indeed the prototype is based on a sound core, as proven in Theorems 3 and 4). Actually, we did not encounter false positives either, although the type system is not complete, as explained in Section 4. Incompleteness can be exploited only with specially crafted code, which is unlikely in real-world apps. Our tools and the experiments are publicly available at https://bitbucket.org/yuske/friendlyfiretools.

5.1 Safety

We provide a tool to analyze a set of IFTTT apps to detect potential cross-app interactions. The tool is developed in C# and .NET Core - a cross-platform version of .NET [24]. The goal of our analysis is to detect whenever an action of one app enables a trigger of another app. The main challenge in detecting such interactions consists in matching the actions and triggers in a way that captures semantic interactions between the two. Indeed, a purely syntactic match of triggers and actions would miss many possible interactions.

To achieve this, we manually analyzed the set of all triggers and actions (1,426 triggers and 891 actions), and assigned a unique label to an action-trigger pair whenever the action could enable the trigger. Specifically, we reviewed the descriptions of actions and triggers, as well as the information about their corresponding services. For example, the action "Post a tweet with image" of the service "Twitter" may enable several triggers of the same service, e.g., "New tweet by you", "New tweet by you with hashtag", "New tweet from search", "New tweet by anyone in area", etc. An interaction may also arise between action and triggers that belong to different services. For example, the action

Table 4. Cross-app interactions in the top 10 most popular apps.

Service:Trigger → Service:Action	Users	Interactions
Instagram: Any new photo by you → Twitter: Post a tweet with image	553,734	544
Weather Underground: Today's weather report → Notifications: Send a notification from the IFTTT app	485,545	0
RSS Feed: New feed item → Email: Send me an email	344,597	564
Weather Underground: Tomorrow's forecast calls for → Notifications: Send a notification from the IFTTT app	319,419	0
iOS Contacts: Any new contact → Google Drive: Add row to spreadsheet	312,266	0
RSS Feed: New feed item matches → Email: Send me an email	285,607	564
Android SMS: New SMS received matches search → Android Device: Set ringtone volume	259,684	702
Location: You enter an area → Notifications: Send a notification from the IFTTT app	242,929	0
Location: You enter an area → Android Device::Turn on WiFi	221,826	0
Amazon Alexa: Say a specific phrase → Phone Call (US only): Call my phone	220,757	112

"Send me an email" of the service "Email" may enable triggers on receiving emails in the "Gmail" and "Office 365 Mail" services. In our analysis we linked 357 actions to triggers on the same service and 8 actions to triggers on different services. Observe that such interactions cannot be detected by purely syntactic comparison of an action's name with a trigger's name. On the other hand, our analysis is conservative since a cross-app interaction may ultimately depend on the settings of an app for a particular user, e.g., the tracked hashtags for the trigger "New tweet by you with hashtag". As a result we create a *knowledge base* of action-trigger pairs that may lead to cross-app interactions.

We used this knowledge base to analyze the number of cross-app interactions in a real-world dataset of about 280,000 IFTTT apps. The dataset contains 19,305 unique apps, namely apps making use of only one trigger-action pair, which we used to analyze potential cross-app interactions. The algorithm is quite simple: Given a set of apps, it visits each app in the set and matches its action with the triggers of the remaining apps leveraging the knowledge base of action-trigger pairs. If there is a match, the algorithm detects an interaction between the two apps.

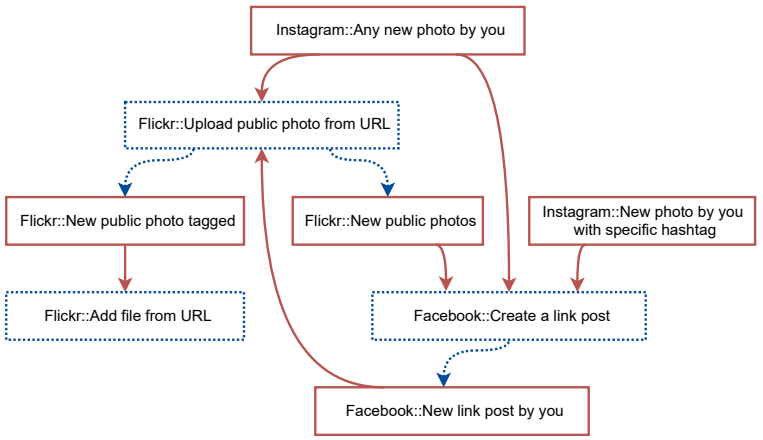
We first investigated the number of possible interactions between apps, which resulted in 1,815,707 cross-app interactions. We then ranked the apps according to their popularity (number of installs), and computed the interactions for the top 10 most popular apps, as reported in Table 4. For instance, the most popular app that posts a tweet whenever a user adds a new photo on Instagram (553,734 installs) may interact with 544 apps from our dataset of 19,305 unique apps. These are all the apps with triggers matching the action "Post a tweet with image" and all the apps with actions matching the trigger "Any new photo by you".

While these results show that the dataset contains a large number of interactions, it is unlikely that a single user installs all these apps. In a more realistic setting, an average IFTTT user installs only a small number of apps. In order to understand possible interactions for a single user, we extracted samples of apps and analyzed them separately. Prior work by Surbatovich et al. [49] suggests that in 2015 an IFTTT user would run in average about 20 apps per day. The actual number would vary by user and may have increased since the time of this study. Therefore, we follow the sampling strategy suggested by Surbatovich et al. [49] and sample subsets of 20, 30, and 40 apps consisting of the n most-frequently adopted apps, and m apps selected randomly from the remaining apps. In our experiments, each subset contains 10 apps selected randomly and 10, 20 and

Table 5. Analysis of cross-app interactions.

	Mean	Median	Standard deviation
Top 10 Random 10	1.7	1	1.8
Top 20 Random 10	3.1	2	2.9
Top 30 Random 10	8.7	8	4.2

Fig. 1. Visualization of cross-app interactions.



30 most-frequently adopted apps respectively. We analyzed the number of interactions for each sample and repeated the experiment 500 times for each sampling strategy.

We summarize our results in Table 5. We observe that a user account with the top 10 popular apps and 10 other apps may contain 1.7 cross-app interaction in average. The average number of cross-app interactions increases to 3.1 and 8.7 in user accounts with 20 and 30 popular apps, respectively. These results show that there is a risk for cross-app interactions in real-world apps.

We also provide a tool to visualize a graph of interactions between apps. The tool can help users to analyze cross-app interactions by specifying a set of apps installed in their IFTTT accounts. Figure 1 depicts an example of a graph generated by our tool, where the red rectangles denote triggers, blue dotted rectangles denote actions. A red arrow represents the trigger and the action of a user's app, while the blue dotted arrow shows the action-trigger pair that may cause a possible interaction.

5.2 Security

We have developed a prototype implementing the rules of the security type system in Table 3. The prototype is written in Java and uses an ANTLR [44] grammar to parse `CaITApp` programs. It takes as input a set of apps and a list of security labels for each service appearing in the apps, and outputs "non-interfering whenever the security typing is successful, i.e., there are no harmful information flows in the set of apps. Otherwise, if the security typing fails, the prototype outputs "interference as well as the list of potentially harmful information flows. Thanks to the compositionality of the type system, the security analysis is performed on each app singularly and then extended to the set of apps (see the rule (Par) in Table 3).

Consider the two apps `PicToDb` and `DbLogger` introduced in the Example 10. They use the confidential services `camera` and `dropbox` and the public services `time` and `dblog`. The first app `PicToDB`

Table 6. Information flow analysis.

App name	Secure	Typable	Public Sink
popular3rdParty1A	✗	✗	twitterContent
popular3rdParty1B	✗	✗	twitterContent
popular3rdParty2A	✗	✗	googleSpreadSheet
popular3rdParty2B	✗	✗	googleSpreadSheet
popular3rdParty3A	✓	✓	–
popular3rdParty3B	✗	✗	dropboxFile
popular3rdParty4A	✗	✗	iosAlbumPhoto
popular3rdParty4B	✗	✗	iosAlbumPhoto
popular3rdParty5A	✗	✗	googleCalendarEv
popular3rdParty5B	✗	✗	googleCalendarEv
popular3rdParty6A	✓	✓	–
popular3rdParty6B	✓	✓	–
popular3rdParty7A	✗	✗	googleSpreadSheet
popular3rdParty7B	✗	✗	googleSpreadSheet
popular3rdParty8A	✗	✗	facebookPost
popular3rdParty8B	✗	✗	facebookPost
popular3rdParty9A	✗	✗	googleSpreadSheet
popular3rdParty9B	✗	✗	googleSpreadSheet
popular3rdParty10A	✗	✗	twitter
popular3rdParty10B	✗	✗	twitter
popular3rdParty11A	✓	✓	–
popular3rdParty11B	✓	✓	–
popular3rdParty12A	✗	✗	email
popular3rdParty12B	✗	✗	email
popular3rdParty13A	✗	✗	emailBody

App	Secure	Typable	Public Sink
popular3rdParty13B	✗	✗	emailBody
forumExample1A	✗	✗	IFTTTnotifMsg
forumExample1B	✗	✗	IFTTTnotifMsg
forumExample2A	✗	✗	richNotificationUrl
forumExample2B	✗	✗	richNotificationUrl
forumExample4A	✗	✗	postBlogger
forumExample4B	✗	✗	postBlogger
forumExample5A	✗	✗	email
forumExample5B	✗	✗	email
forumExample6A	✓	✓	–
forumExample6B	✗	✗	email
forumExample7A	✓	✓	–
forumExample7B	✓	✓	–
Area	✗	✗	emailA
DbLogger	✗	✗	dblog
aSaver	✗	✗	lights
Forward	✗	✗	emailB
Leak	✓	✓	–
PicToDb	✓	✓	–
SimPres	✗	✗	lights
SmokeAlarm	✗	✗	alarm
Sprinks	✓	✓	–
Therm	✓	✓	–
Welcome	✓	✓	–
Win	✓	✓	–

is secure, while the second app `DbLogger` is not, since it does not satisfy `L`-level noninterference (Definition 11). Our prototype is able to type the first app, while it fails for the second one. In particular, the tool signals an insecure attempt to update the service `dblog` of the app `DbLogger`. This reflects the fact that the app contains a flow from a confidential service (`dropbox` in this case) to the public service `dblog`, and the latter is committed to the cloud.

To validate the prototype, we have adapted and analyzed 38 samples taken from the benchmarks of Bastys et al. [8] and 12 samples taken from the apps presented in this paper, for a total of 50 samples. The former benchmarks contain a selection of popular real-world IFTTT apps as well as apps from online forums. We extended this dataset by inferring code from the apps' descriptions, and then translated it into `CaITApp`. We have manually checked the information flows and then compared the results with the outputs given by our tool. The comparison is reported in Table 6.

The tool does not report false positives, accepting all secure apps. We believe that for *reasonable* IFTTT apps, our type system will not exhibit false positives. As we have seen in Section 4, the type system is not complete, but a false positive requires specifically crafted code, that is unlikely to be present in real-world apps.

Declassification. We have also extended the prototype to support declassification via the construct $\mathtt{declassify}(\mathtt{e}, \sigma)$ of Section 4.1. Consider again the example app `DbLogger`, which is insecure with respect to Definition 11 and, consequently, rejected by our prototype without declassification. However, the app is leaking only the name of the user's photo which might be acceptable by the user. We have modified the app in a way that the existing information flow is now allowed, as shown in Example 10. Specifically, we have declassified the expression $\mathtt{nameOf}(\mathtt{read}(\mathtt{dropbox}))$ via the annotation $\mathtt{declassify}(\mathtt{nameOf}(\mathtt{read}(\mathtt{dropbox})), \mathtt{L})$. Now the app satisfies `L`-level noninterference up to declassification (Definition 13) and it is promptly accepted as secure by our prototype.

To validate the extended prototype, we considered the same 50 samples, but we declassified some services concerning *time* and *location*. For example, in IFTTT a malicious app can use timing information at the granularity of millisecond to leak sensitive information [7], hence timing information is considered confidential. On the other hand, coarser-grained time readings, e.g., at the granularity of second or hour, are too imprecise to leak sensitive information and can be safely

Table 7. Information flow analysis with declassification.

App name	Secure	Typable	Public Sink
popular3rdParty1A	✗	✗	twitterContent
popular3rdParty1B	✗	✗	twitterContent
popular3rdParty2A	✗	✗	googleSpreadSheet
popular3rdParty2B	✗	✗	googleSpreadSheet
popular3rdParty3A	✓	✓	–
popular3rdParty3B	✗	✗	dropboxFile
popular3rdParty4A	✗	✗	iosAlbumPhoto
popular3rdParty4B	✗	✗	iosAlbumPhoto
popular3rdParty5A	✓	✓	–
popular3rdParty5B	✗	✗	googleCalendarEv
popular3rdParty6A	✓	✓	–
popular3rdParty6B	✓	✓	–
popular3rdParty7A	✗	✗	googleSpreadSheet
popular3rdParty7B	✗	✗	googleSpreadSheet
popular3rdParty8A	✗	✗	facebookPost
popular3rdParty8B	✗	✗	facebookPost
popular3rdParty9A	✗	✗	googleSpreadSheet
popular3rdParty9B	✗	✗	googleSpreadSheet
popular3rdParty10A	✓	✓	–
popular3rdParty10B	✓	✓	–
popular3rdParty11A	✓	✓	–
popular3rdParty11B	✓	✓	–
popular3rdParty12A	✗	✗	email
popular3rdParty12B	✗	✗	email
popular3rdParty13A	✗	✗	emailBody

App	Secure	Typable	Public Sink
popular3rdParty13B	✗	✗	emailBody
forumExample1A	✓	✓	–
forumExample1B	✗	✗	IFTTTnotifMsg
forumExample2A	✗	✗	richNotificationUrl
forumExample2B	✗	✗	richNotificationUrl
forumExample4A	✗	✗	postBlogger
forumExample4B	✗	✗	postBlogger
forumExample5A	✗	✗	email
forumExample5B	✗	✗	email
forumExample6A	✓	✓	–
forumExample6B	✗	✗	email
forumExample7A	✓	✓	–
forumExample7B	✓	✓	–
Area	✓	✓	–
DbLogger	✓	✓	–
aSaver	✓	✓	–
Forward	✗	✗	emailB
Leak	✓	✓	–
PicToDb	✓	✓	–
SimPres	✓	✓	–
SmokeAlarm	✗	✗	alarm
Sprinks	✓	✓	–
Therm	✓	✓	–
Welcome	✓	✓	–
Win	✓	✓	–

declassified. Similarly, location is usually considered a confidential resource, however, approximate locations can sometimes be disclosed. For instance, in the app `Area` the exact coordinates (latitude and longitude) are confidential, but the fact that the position belongs to a given geographic area can be made public. Similarly, the app `eSaver` uses timing information at the granularity of hour, which can be safely declassified as public.

Again, we have manually checked the information flows of the apps and then compared the results with the output of our prototype with declassification. The comparison is summarized in Table 7. As we can see, more apps (8 apps, gray-highlighted in Table 7) are now considered secure. As for the case without declassification, the prototype does not signal any false negative, reflecting the fact that the type system is sound (Theorem 4).

6 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, 17]. 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.

Comparison with the conference version. The present work is a revised extension of the conference paper [6]. Here, we generalized the definition of termination-insensitive hiding bisimulation and fixed the soundness proof accordingly. Basically, in the previous definition high-level actions can be matched by the same action or by τ-actions. In the new definition, instead, high-level actions can be matched by arbitrary high-level actions or τ-actions. This makes our definition of noninterference more permissive, as shown by the following example:

$$\texttt{if high = 0 then } \{\texttt{update}(\texttt{high}_1)\} \texttt{ else } \{\texttt{update}(\texttt{high}_2)\}; \texttt{update}(\texttt{low})$$

where $\texttt{high}, \texttt{high}_1, \texttt{high}_2$ are confidential services and `low` is a public service; this app is noninterferent, since there are no flows from confidential to public services, however, it does not fulfill the definition of noninterference given in [6]. Instead, it correctly fulfills the definition of noninterference of this paper (Definition 11), with the revised bisimulation of Definition 10.

We then extended our information-flow control mechanism in order to deal with declassification, namely intentional leaks of sensitive information. Indeed, in many real-world applications, noninterference is a too strong policy and some information can be safely disclosed even if it is not part of the flow relation, i.e., it can be declassified. We followed the approach of delimited release [46], introducing a declassification primitive in our language: the release of confidential information is only allowed via declassification points. We defined σ-level noninterference up to declassification, extended the security type system, and proved soundness up to declassification.

Finally, we added a new section concerning the implementation and validation of our enforcement mechanisms. First, we implemented a prototype and analyzed a benchmark dataset of 20,000 IFTTT apps, in order to capture cross-app interactions, using the syntactic conditions given in Definition 6. Then, we implemented and validated our security type system, including declassification, on a set of real-world IoT apps, translated in our language `CaITApp`.

Securing IoT apps. Recent research points out the security and safety risks arising in the context of IoT apps. Surbatovich et al. [49] study a dataset of 20K IFTTT apps and provide an empirical evaluation of potential secrecy and integrity violations, including violations due to cross-app interactions. Celik et al. [18, 19] propose static and dynamic enforcement mechanisms for unveiling cross-app interference vulnerabilities. Ding et al. [23] 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. [20] propose a systematic categorization of threats arising from unintentional or malicious interactions of apps in IoT platforms. To detect cross-app interference, they use symbolic execution techniques to analyze the apps' implementation. Nguyen et al. [41] design IoTSan, a system that uses model checking to reveal cross-app interaction flows. Similarly, SafeChain by Hsu et al. [30] leverage model checking techniques to identify cross-app vulnerabilities in IFTTT trigger-action rules. The above-mentioned works provide an excellent motivation for our foundational contributions: Our framework can be used to validate soundness and permissiveness of these verification techniques. Moreover, our empirical analysis of IFTTT applets shows that flow-sensitive security types can help tracking vulnerabilities with no false positives.

Another line of work focuses on enforcement mechanisms for checking security and safety of single IoT apps. Fernandes et al. [25] present FlowFence, an approach building secure IoT apps via information-flow tracking and controlled declassification. Celik et al. [16] leverage static taint tracking to identify sensitive data leaks in IoT apps. Bastys et al. [7–9] 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. [26] 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 [27] 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) [28]. Tuma et al. [50] propose a practical and principled approach to uncover insecure information the level of the design model, which help improving the precision of our safety analysis of cross-app interaction. Volpano and Smith [51] study a *flow-insensitive* type system for imperative languages. Because in our language the communication between services is handled via explicit *update* statements, a flow-insensitive type system would

be too restrictive and reject more secure programs. Hunt and Sands [31] 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 [21] 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 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 [52] propose Petri-net models to verify *nondeducibility security properties* of a natural gas pipeline system. More recently, Bohrer and Platzer [15] 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.

Formalizations of 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. [40] 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. [35] 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. Lanotte and Merro [36] extend and generalize the work of [35] in a timed setting by providing a bisimulation-based semantic theory that is suitable for compositional reasoning. Lanotte et al. [37] adapt a discrete-time generalization of Desharnais et al.'s *weak bisimulation metric* [22] to estimate the impact of attacks targeting sensor devices of IoT systems. Bodei et al. [13] 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. In [12], the same authors extend their work to infer *quantitative measures* to establish the cost of possibly security countermeasures, in terms of time and energy.

Process calculi have been used to model the semantics and express security conditions in different contexts including operating systems like Linux [34] and Android [33], and web browsers [10, 11, 14]. In the same vein, our calculus `CaITApp` is targeted to capture the execution model of IoT app platforms consisting of concurrent execution of simple apps that trigger the execution of code upon receiving events on source services and ultimately dispatch actions on action services. In contrast, `CaITApp` is simpler and it captures the execution semantics of IoT apps explicitly.

7 CONCLUSIONS

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. 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. We implemented our approach and demonstrated its feasibility on dataset of real-world apps.

Our analysis indicates that flow-sensitive security type systems with support for declassification policies are a good fit for enforcing IoT apps' security. While these results are encouraging, we remark that our conclusions are limited to the dataset under consideration. Unfortunately filter code in IFTTT is not publicly available, hence we considered a best-effort approach by analyzing code from public forums and previous works [8]. On the other hand, the filter code is mainly used for simple customizations of trigger-action rules, hence we expect the security type system to have very low false positives. While prior works have successfully applied techniques such as runtime monitoring [8, 19] and model checking [18, 20, 30, 41], static analysis via security type systems, at least from the perspective of IoT platform owners, can be preferable to validate IoT apps before publishing them to the store.

Declassification is also a challenging problem in information flow control, especially from a usability perspective. Because our declassification policies are global and refer only to the services, laundering attacks are excluded by design and IoT platform owners and end user can focus on defining and interpreting declassification polices in relation to their IoT services. For our dataset, the filter code requires at most one declassification, hence leaks via combination of multiple declassifications do not appear. An interesting direction for future work is to consider inference of declassification policies from filter code and their representation in a user-friendly manner. Ideally, such analysis should be implemented by the IoT platform owners and presented to end users prior to installing an IoT app.

Finally, we remark that our analysis relies on security classifications of sources and sinks from trigger-action services. As pointed out by existing works [5, 49], these classifications may depend on the context as well as user preferences, e.g., the security settings of a specific service like the audience of a social network feed. These works are complementary to our approach and advances in this direction can improve the accuracy of our results.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for pointing out issues with our security type system and the associated soundness proofs. We thank Ruggero Lanotte for suggesting us the issue on high-level actions of our termination-insensitive hiding bisimulation. 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, University and Research (MIUR).

REFERENCES

[1] Ravi Akella, Han Tang, and Bruce M. McMillin. 2010. Analysis of information flow security in cyber-physical systems. *International Journal of Critical Infrastructure Protection* 3, 3-4 (2010), 157–173. https://doi.org/10.1016/j.ijcip.2010.09.001

[2] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *Symposium on Security and Privacy, S&P 2019*. IEEE Computer Society, 1362–1380. https://doi.org/10.1109/SP.2019.00013

[3] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. 1998. On bisimulations for the asynchronous pi-calculus. *Theoretical Computer Science* 195, 2 (1998), 291 – 324. https://doi.org/10.1016/S0304-3975(97)00223-5

[4] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Computer Security - ESORICS (Lecture Notes in Computer Science, Vol. 5283)*. Springer Berlin Heidelberg, 333–348. https://doi.org/10.1007/978-3-540-88313-5_22

[5] Musard Balliu, Iulia Bastys, and Andrei Sabelfeld. 2019. Securing IoT Apps. *IEEE Security & Privacy Magazine* 17, 5 (2019), 22–29. https://doi.org/10.1109/MSEC.2019.2914190

[6] Musard Balliu, Massimo Merro, and Michele Pasqua. 2019. Securing Cross-App Interactions in IoT Platforms. In *Computer Security Foundations Symposium, CSF*. IEEE Computer Society, 319–334. https://doi.org/10.1109/CSF.2019.00029

[7] Iulia Bastys, Musard Balliu, Tamara Rezk, and Andrei Sabelfeld. 2020. Clockwork: Tracking Remote Timing Attacks. In *Computer Security Foundations Symposium, CSF*. IEEE Computer Society, 350–365. https://doi.org/10.1109/CSF49147.2020.00032

[8] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What?: Controlling Flows in IoT Apps. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 1102–1119. https://doi.org/10.1145/3243734.3243841

[9] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. 2018. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *Secure IT Systems - Nordic Conference, NordSec (Lecture Notes in Computer Science, Vol. 11252)*. 19–37. https://doi.org/10.1007/978-3-030-03638-6_2

[10] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*.

[11] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. 2011. Reactive non-interference for a browser model. In *International Conference on Network and System Security, NSS*. IEEE Computer Society, 97–104. https://doi.org/10.1109/ICNSS.2011.6059965

[12] Chiara Bodei, Stefano Chessa, and Letterio Galletta. 2019. Measuring security in IoT communications. *Theoretical Computer Science* 764 (2019), 100 – 124. https://doi.org/10.1016/j.tcs.2018.12.002

[13] Chiara Bodei, Pierpaolo Degano, Gian Luigi Ferrari, and Letterio Galletta. 2017. Tracing where IoT data are collected and aggregated. *Logical Methods in Computer Science* 13, 3 (2017). https://doi.org/10.23638/LMCS-13(3:5)2017

[14] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. 2009. Reactive noninterference. In *Conference on Computer and Communications Security, CCS*. ACM, 79–90. https://doi.org/10.1145/1653662.1653673

[15] Brandon Bohrer and André Platzer. 2018. A Hybrid, Dynamic Logic for Hybrid-Dynamic Information Flow. In *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*. IEEE Computer Society, 115–124. https://doi.org/10.1145/3209108.3209151

[16] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *USENIX Security Symposium, USENIX*. USENIX Association, 1687–1704.

[17] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. 2019. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Comput. Surv.* 52, 4 (2019), 74:1–74:30. https://doi.org/10.1145/3333501

[18] Z. Berkay Celik, Patrick D. McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *USENIX Annual Technical Conference, USENIX ATC*. USENIX Association, 147–158.

[19] Z. Berkay Celik, Gang Tan, and Patrick D. McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[20] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2020. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. IEEE Computer Society, 411–423. https://doi.org/10.1109/DSN48063.2020.00056

[21] Delphine Demange and David Sands. 2009. All Secrets Great and Small. In *European Symposium on Programming Languages and Systems, ESOP (Lecture Notes in Computer Science, Vol. 5502)*. Springer-Verlag, 207–221. https://doi.org/10.1007/978-3-642-00590-9_16

[22] Jose Desharnais, Radha Jagadeesan, Vineet Gupta, and Prakash Panangaden. 2002. The Metric Analogue of Weak Bisimulation for Probabilistic Processes. In *IEEE Symposium on Logic in Computer Science, LICS*. IEEE Computer Society, 413–422. https://doi.org/10.1145/1967701.1967710

[23] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 832–846. https://doi.org/10.1145/3243734.3243865

[24] dotnet 2020. .NET - Free. Cross-platform. Open source. https://dotnet.microsoft.com/.

[25] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security Symposium, USENIX Security*. IUSENIX Association, 531–548.

[26] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[27] Riccardo Focardi and Roberto Gorrieri. 2000. Classification of Security Properties (Part I: Information Flow). In *Foundations of Security Analysis and Design, FOSAD (Lecture Notes in Computer Science, Vol. 2171)*. 331–396. https://doi.org/10.1007/3-540-45608-2_6

[28] Riccardo Focardi and Fabio Martinelli. 1999. A Uniform Approach for the Definition of Security Properties. In *World Congress on Formal Methods, FM (Lecture Notes in Computer Science, Vol. 1708)*. Springer, 794–813. https://doi.org/10.1007/3-540-48119-2_44

[29] Kohei Honda and Mario Tokoro. 1991. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming, ECOOP (Lecture Notes in Computer Science, Vol. 512)*. Springer, 133–147. https://doi.org/10.1007/BFb0057019

[30] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. SafeChain: Securing Trigger-Action Programming From Attack Chains. *IEEE Trans. Inf. Forensics Secur.* 14, 10 (2019), 2607–2622. https://doi.org/10.1109/TIFS.2019.2899758

[31] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 79–90. https://doi.org/10.1145/1111037.1111045

[32] IFTTT 2020. IFTTT: If This Then That. https://ifttt.com.

[33] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android - (Extended Abstract). In *European Symposium on Research in Computer Security, ESORICS (Lecture Notes in Computer Science, Vol. 8134)*. Springer, 775–792. https://doi.org/10.1007/978-3-642-40203-6_43

[34] Maxwell N. Krohn and Eran Tromer. 2009. Noninterference for a Practical DIFC-Based Operating System. In *IEEE Symposium on Security and Privacy, IEEE S&P*. IEEE Computer Society, 61–76. https://doi.org/10.1109/SP.2009.23

[35] Ivan Lanese, Luca Bedogni, and Marco Di Felice. 2013. Internet of things: a process calculus approach. In *Annual ACM Symposium on Applied Computing, SAC*. ACM, 1339–1346. https://doi.org/10.1145/2480362.2480615

[36] Ruggero Lanotte and Massimo Merro. 2018. A semantic theory of the Internet of Things. *Information and Computation* 259, 1 (2018), 72–101. https://doi.org/10.1016/j.ic.2018.01.001

[37] Ruggero Lanotte, Massimo Merro, and Simone Tini. 2018. Towards a formal notion of impact metric for cyber-physical attacks. In *International Conference on Integrated Formal Methods, IFM (Lecture Notes in Computer Sciencce, Vol. 11023)*. Springer, 296–315.

[38] Massimo Merro and Davide Sangiorgi. 2004. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science* 14, 5 (2004), 715–767. https://doi.org/10.1017/S0960129504004323

[39] MSPA 2020. Microsoft Power Automate. https://flow.microsoft.com/en-us/.

[40] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. 2017. IOTA: A Calculus for Internet of Things Automation. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 119–133. https://doi.org/10.1145/3133850.3133860

[41] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of IoT Systems. In *International Conference on Emerging Networking EXperiments and Technologies, CoNEXT*. ACM, 191–203. https://doi.org/10.1145/3281411.3281440

[42] NST 2019. Nest Thermostat. https://ifttt.com/services/nest_thermostat.

[43] Federica Paci, Davide Bianchin, Elisa Quintarelli, and Nicola Zannone. 2020. IFTTT Privacy Checker. In *Emerging Technologies for Authorization and Authentication, ETAA (Lecture Notes in Computer Science, Vol. 12515)*. Springer, 90–107. https://doi.org/10.1007/978-3-030-64455-0_6

[44] T. Parr. 2013. *The Definitive ANTLR 4 Reference* (2 ed.). Raleigh, NC.

[45] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[46] Andrei Sabelfeld and Andrew C. Myers. 2003. A Model for Delimited Information Release. In *International Symposium - Software Security, ISSS (Lecture Notes in Computer Science, Vol. 3233)*. Springer, 174–191. https://doi.org/10.1007/978-3-540-37621-7_9

[47] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548. https://doi.org/10.3233/JCS-2009-0352

[48] smt 2020. SmartThings. https://ifttt.com/smartthings.

[49] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *International Conference on World Wide Web, WWW*. ACM, 1501–1510. https://doi.org/10.1145/3038912.3052709

[50] Katja Tuma, Musard Balliu, and Riccardo Scandariato. 2019. Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In *IEEE International Conference on Software Architecture, ICSA*. IEEE Computer Society, 191–200. https://doi.org/10.1109/ICSA.2019.00028

[51] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304

[52] Jingming Wang and Huiqun Yu. 2014. Analysis of the Composition of Non-Deducibility in Cyber-Physical Systems. *Applied Mathematics & Information* 8 (2014), 3137–3143. Issue 6. https://doi.org/10.12785/amis/080655
[53] Zapier 2020. Zapier. https://zapier.com.

A PROOFS

Proof of Theorem 1. We prove the first part of the theorem (for the second part, just swap S with R). Assume $\mathsf{S} \nrightarrow \mathsf{R}$, i.e., $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{trg}(\mathsf{R}) = \varnothing$, then we have to prove that for any global store $\mathfrak{G} \in \mathbb{S}$ 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 $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \not\approx_{H_\mathsf{S}} \mathsf{R}$, for some global store $\mathfrak{G} \in \mathbb{S}$. This means that it does not exist a hiding bisimulation $\mathcal{R}$ parametric on H_S that contains the pair of configurations $(\mathfrak{C}_a, \mathfrak{C}_b)$, with $\mathfrak{C}_a \stackrel{\text{def}}{=} \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R}$ and $\mathfrak{C}_b \stackrel{\text{def}}{=} \langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{R}$. More precisely, by definition of bisimulation relation, whenever we try to build up a hiding bisimulation $\mathcal{R}$ parametric on H_S and containing the pair $(\mathfrak{C}_a, \mathfrak{C}_b)$, the bisimulation game stops in a pair of configurations $(\mathfrak{C}'_a, \mathfrak{C}'_b)$, with $\mathfrak{C}'_a$ (*resp.*, $\mathfrak{C}'_b$) derivative of $\mathfrak{C}_a$ (*resp.*, $\mathfrak{C}_b$), because: (i) either $\mathfrak{C}'_a$ can perform an action α that cannot be (weakly) mimicked by $\mathfrak{C}'_b$ (the vice versa is similar), or (ii) action mimicking is always possible but it leads us to configuration pairs of the form $(\mathfrak{C}''_a, \mathfrak{C}''_b)$ which do not belong to $\mathcal{R}$. Actually, since a bisimulation proof is a constructive procedure, we can always assume that the sought relation $\mathcal{R}$ is large enough so that case (ii) never applies.

Let $\mathfrak{C}'_a \stackrel{\text{def}}{=} \langle \mathfrak{G}'_a, \mathfrak{L}'_a \rangle \triangleright \mathsf{S}'_a \parallel \mathsf{R}'_a$ and $\mathfrak{C}'_b \stackrel{\text{def}}{=} \langle \mathfrak{G}'_b, \mathfrak{L}'_b \rangle \triangleright \mathsf{R}'_b$, with S'_a derivative of S, R'_a derivative of R (we recall that our apps cannot directly interact), and R'_b derivative of R. We proceed by case analysis on the action α that would distinguish the two configurations $\mathfrak{C}'_a$ and $\mathfrak{C}'_b$.

– $\alpha = \tau$. 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 2. 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.

- 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.
- 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 id reaches different states in the two configurations $\mathfrak{C}'_a$ and $\mathfrak{C}'_b$ leading to two possible situations: (i) the writing on x is possible in $\mathfrak{C}'_a$ but not in $\mathfrak{C}'_b$ (or vice versa); (ii) the writing on x is possible in both configurations but with different values. Since both systems S ∥ 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 in the two configurations if and only if it would be affected by S. In particular, this means that in the execution trace leading from $\mathfrak{C}_a$ to $\mathfrak{C}'_a$, (a derivative of) the system S should have modified (i.e., written): (a) either a service that R'_a listens on, or (b) a service that R'_a reads from the global store. Recall that there is no direct information passing between apps, so the only way for apps to interact is via the global store on the cloud. However, the syntactic condition $\mathsf{S} \nrightarrow \mathsf{R}$, i.e., $\mathfrak{act}(\mathsf{S}) \cap \mathfrak{trg}(\mathsf{R}) = \emptyset$, is trivially preserved by all derivatives of S and R. This ensures that neither case applies.

As it does not exist a distinguishing action α, it follows that the original configurations $\mathfrak{C}_a$ and $\mathfrak{C}_b$ must be hiding bisimilar, i.e., $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \parallel \mathsf{R} \approx_{H_\mathsf{S}} \mathsf{R}$, with $H_\mathsf{S} = \mathfrak{upd}(\mathsf{S})$.

□

Proof of Theorem 2. Let us focus on the first part of the theorem (for the second part, just swap S with R). Assume $\mathsf{S} \stackrel{\Lambda}{\nrightarrow} \mathsf{R}$ for some dependency policy Λ, i.e., $\mathfrak{clo}(\Lambda, \mathfrak{act}(\mathsf{S})) \cap \mathfrak{trg}(\mathsf{R}) = \varnothing$, then we

have to prove that for any global store $\mathfrak{G} \in \mathbb{S}$ we have: $\langle \mathfrak{G}, \mathfrak{L}_\perp \rangle \triangleright S \parallel R \overset{\Delta}{\approx}_{H_S} R$, with $H_S = \mathfrak{upd}(S)$. The proof proceeds by contradiction and goes exactly along the same lines of that of Theorem 1. Indeed, the setting of the bisimulation is exactly the same but the syntactic condition $S \overset{\Delta}{\nrightarrow} R$ (Definition 8) entails the condition $S \nrightarrow R$ (Definition 6) used to prove Theorem 1. □

Lemma 1 (Substitution lemma)**.** *Let* Q *be an arbitrary process and* $\mathbb{X}$ *be a process variable that may occur free in* Q*. If* $\sigma \vdash \Gamma_1 \{ Q \} \Gamma_2$*, for some security clearance* σ *and some type environments* Γ_1 *and* Γ_2 *of the form* $\Gamma_1 = \Gamma_1'[\mathbb{X} \mapsto \sigma]$ *and* $\Gamma_2 = \Gamma_2'[\mathbb{X} \mapsto \sigma]$*, then* $\sigma \vdash \Gamma_1' \{ Q \{^{\mathsf{fix}\,\mathbb{X} \bullet Q}/_{\mathbb{X}}\} \} \Gamma_2'$.

Proof. Let $P = \mathsf{fix}\,\mathbb{X} \bullet Q$. The proof is by rule induction on the derivation of $\sigma \vdash \Gamma_1 \{ Q \} \Gamma_2$, for some security clearance σ, and some type environments Γ_1 and Γ_2 of the form $\Gamma_1 = \Gamma_1'[\mathbb{X} \mapsto \sigma]$ and $\Gamma_2 = \Gamma_2'[\mathbb{X} \mapsto \sigma]$, respectively, for some Γ_1' and Γ_2'.

- Suppose that $\sigma \vdash \Gamma_1 \{ Q \} \Gamma_2$ was derived by an application of the typing rule (Pvar). There are two cases.
  - $Q = \mathbb{X}$. By definition of rule (Pvar), we have $\Gamma_1 = \Gamma_2$. As $Q = \mathbb{X}$, the process substitution returns: $Q \{^P/_{\mathbb{X}}\} = \mathbb{X} \{^P/_{\mathbb{X}}\} = P$. We recall that $P = \mathsf{fix}\,\mathbb{X} \bullet Q$. As $\Gamma_1 = \Gamma_2$, by an application of the typing rule (Fix) it follows that $\sigma \vdash \Gamma_1' \{ P \} \Gamma_2'$.
  - $Q = \mathbb{Y} \neq \mathbb{X}$. This case is straightforward as the process substitution does not apply to Q.
- Suppose that $\sigma \vdash \Gamma_1 \{ Q \} \Gamma_2$ was derived by an application of the typing rule (Fix). As a consequence, we have: (i) $Q = \mathsf{fix}\,\mathbb{Y} \bullet Q_1$, for some Q_1 and $\mathbb{Y}$ (up to α-conversion we can always assume $\mathbb{Y} \neq \mathbb{X}$), (ii) $\Gamma_1 = \Gamma_2$, and (iii) $\sigma \vdash \hat{\Gamma}_1 \{ Q_1 \} \hat{\Gamma}_2$, for $\hat{\Gamma}_1 = \Gamma_1[\mathbb{Y} \mapsto \sigma]$ and $\hat{\Gamma}_2 = \Gamma_2[\mathbb{Y} \mapsto \sigma]$ (obviously, $\hat{\Gamma}_1 = \hat{\Gamma}_2$). Now, the process substitution on Q returns: $Q \{^P/_{\mathbb{X}}\} = (\mathsf{fix}\,\mathbb{Y} \bullet Q_1) \{^P/_{\mathbb{X}}\} = \mathsf{fix}\,\mathbb{Y} \bullet (Q_1 \{^P/_{\mathbb{X}}\})$. By inductive hypothesis we derive: $\sigma \vdash \Gamma_1'' \{ Q_1 \{^P/_{\mathbb{X}}\} \} \Gamma_2''$, for $\hat{\Gamma}_1 = \Gamma_1''[\mathbb{X} \mapsto \sigma]$ and $\hat{\Gamma}_2 = \Gamma_2''[\mathbb{X} \mapsto \sigma]$ (obviously, $\Gamma_1'' = \Gamma_2''$). As $\Gamma_1'' = \Gamma_2''$, by an application of the typing rule (Fix) it follows that $\sigma \vdash \Gamma_1''' \{ \mathsf{fix}\,\mathbb{Y} \bullet (Q_1 \{^P/_{\mathbb{X}}\}) \} \Gamma_2'''$, for $\Gamma_1'' = \Gamma_1'''[\mathbb{Y} \mapsto \sigma]$ and $\Gamma_2'' = \Gamma_2'''[\mathbb{Y} \mapsto \sigma]$ (obviously, $\Gamma_1''' = \Gamma_2'''$). Thus, for $i \in \{1, 2\}$ we have: $\hat{\Gamma}_i = \Gamma_i[\mathbb{Y} \mapsto \sigma] = \Gamma_i'[\mathbb{X} \mapsto \sigma, \mathbb{Y} \mapsto \sigma]$ and $\hat{\Gamma}_i = \Gamma_i''[\mathbb{X} \mapsto \sigma] = \Gamma_i'''[\mathbb{Y} \mapsto \sigma, \mathbb{X} \mapsto \sigma]$. This entails $\Gamma_1' = \Gamma_1'''$ and $\Gamma_2' = \Gamma_2'''$. As a consequence, from (i) $Q = \mathsf{fix}\,\mathbb{Y} \bullet Q_1$, (ii) $Q \{^P/_{\mathbb{X}}\} = \mathsf{fix}\,\mathbb{Y} \bullet (Q_1 \{^P/_{\mathbb{X}}\})$, and (iii) $\sigma \vdash \Gamma_1''' \{ \mathsf{fix}\,\mathbb{Y} \bullet (Q_1 \{^P/_{\mathbb{X}}\}) \} \Gamma_2'''$, it follows that $\sigma \vdash \Gamma_1' \{ Q \{^P/_{\mathbb{X}}\} \} \Gamma_2'$.
- Suppose that $\sigma \vdash \Gamma_1 \{ Q \} \Gamma_2$ was derived by an application of the typing rule (Seq). This derivation is possible only under the hypotheses that: (i) $Q = Q_1; Q_2$, (ii) $\sigma \vdash \Gamma_1 \{ Q_1 \} \Gamma_{12}$, for some Γ_{12}, of the form $\Gamma_{12} = \Gamma_{12}'[\mathbb{X} \mapsto \sigma]$ for some Γ_{12}', and (iii) $\sigma \vdash \Gamma_{12} \{ Q_2 \} \Gamma_2$. Now, the process substitution returns: $Q \{^P/_{\mathbb{X}}\} = (Q_1; Q_2) \{^P/_{\mathbb{X}}\} = Q_1 \{^P/_{\mathbb{X}}\}; Q_2 \{^P/_{\mathbb{X}}\}$. By inductive hypothesis, $\sigma \vdash \Gamma_1' \{ Q_1 \{^P/_{\mathbb{X}}\} \} \Gamma_{12}'$ and $\sigma \vdash \Gamma_{12}' \{ Q_2 \{^P/_{\mathbb{X}}\} \} \Gamma_2'$. Thus, by an application of the typing rule (Seq) it follows that $\sigma \vdash \Gamma_1' \{ Q_1 \{^P/_{\mathbb{X}}\}; Q_2 \{^P/_{\mathbb{X}}\} \} \Gamma_2'$. Hence, $\sigma \vdash \Gamma_1' \{ Q \{^P/_{\mathbb{X}}\} \} \Gamma_2'$.
- Suppose that $\sigma \vdash \Gamma_1 \{ Q \} \Gamma_2$ was derived by an application of the typing rule (IfElse). This derivation is possible only under the hypotheses that: (i) $Q = \mathsf{if}\ b\ \mathsf{then}\ \{Q_1\}\ \mathsf{else}\ \{Q_2\}$, for some b, Q_1 and Q_2, (ii) $\Gamma_1 \vdash b : \sigma'$, for some σ', (iii) $\sigma \sqcup \sigma' \vdash \Gamma_1 \{ Q_1 \} \Gamma_2$, and (iv) $\sigma \sqcup \sigma' \vdash \Gamma_1 \{ Q_2 \} \Gamma_2$. The process substitution on Q returns: $Q \{^P/_{\mathbb{X}}\} = (\mathsf{if}\ b\ \mathsf{then}\ \{Q_1\}\ \mathsf{else}\ \{Q_2\}) \{^P/_{\mathbb{X}}\} = \mathsf{if}\ b\ \mathsf{then}\ \{Q_1 \{^P/_{\mathbb{X}}\}\}\ \mathsf{else}\ \{Q_2 \{^P/_{\mathbb{X}}\}\}$. By inductive hypothesis we derive: $\sigma \sqcup \sigma' \vdash \Gamma_1' \{ Q_1 \{^P/_{\mathbb{X}}\} \} \Gamma_2'$ and $\sigma \sqcup \sigma' \vdash \Gamma_1' \{ Q_2 \{^P/_{\mathbb{X}}\} \} \Gamma_2'$. Furthermore, from $\Gamma_1 \vdash b : \sigma'$ it follows $\Gamma_1' \vdash b : \sigma'$. Thus, we derive $\sigma \vdash \Gamma_1' \{ \mathsf{if}\ b\ \mathsf{then}\ \{Q_1 \{^P/_{\mathbb{X}}\}\}\ \mathsf{else}\ \{Q_2 \{^P/_{\mathbb{X}}\}\} \} \Gamma_2'$ by an application of the typing rule (IfElse). Hence, $\sigma \vdash \Gamma_1' \{ Q \{^P/_{\mathbb{X}}\} \} \Gamma_2'$.

- Suppose that $\sigma \vdash \Gamma_1\{\mathsf{Q}\}\Gamma_2$ was derived by an application of one typing rule among (Skip), (Update), and (Listen). These cases are easy as the substitution does not affect the process Q.
- Suppose that $\sigma \vdash \Gamma_1\{\mathsf{Q}\}\Gamma_2$ was derived by an application of the subtyping rule (Sub.Proc). This derivation is possible only under the hypotheses that there are $\hat{\sigma}$, $\hat{\Gamma}_1$ and $\hat{\Gamma}_2$ such that: (i) $\hat{\sigma} \vdash \hat{\Gamma}_1\{\mathsf{Q}\}\hat{\Gamma}_2$, (ii) $\sigma \preccurlyeq \hat{\sigma}$, (iii) $\Gamma_1 \preccurlyeq \hat{\Gamma}_1$, and (iv) $\hat{\Gamma}_2 \preccurlyeq \Gamma_2$. By inductive hypothesis, we derive $\hat{\sigma} \vdash \Gamma_1''\{\mathsf{Q}\{^{\mathsf{P}}/_{\mathbb{X}}\}\}\Gamma_2''$, for $\hat{\Gamma}_1 = \Gamma_1''[\mathbb{X} \mapsto \hat{\sigma}]$ and $\hat{\Gamma}_2 = \Gamma_2''[\mathbb{X} \mapsto \hat{\sigma}]$. As $\Gamma_1 = \Gamma_1'[\mathbb{X} \mapsto \sigma]$ and $\Gamma_2 = \Gamma_2'[\mathbb{X} \mapsto \sigma]$, it follows that $\Gamma_1' \preccurlyeq \Gamma_1''$ and $\Gamma_2'' \preccurlyeq \Gamma_2'$. Thus, by an application of the subtyping rule (Sub.Proc) it follows that $\sigma \vdash \Gamma_1'\{\mathsf{Q}\{^{\mathsf{P}}/_{\mathbb{X}}\}\}\Gamma_2'$, as required.

□

Lemma 2 (Subject reduction for process configurations). *Let $\Sigma \in \mathsf{Service} \rightharpoonup SL$ be a security policy, P be a process, and $\sigma \in SL$ be a security level. If $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$ and $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P} \xrightarrow{\lambda} \langle \mathfrak{G}', \phi'\rangle \triangleright \mathsf{P}'$ 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 1). In the table, all transition rules are axioms (base cases of the induction) except for the rule (Seq) which represents the only inductive case. Let us proceed by case analysis on which semantic rules has been used to derive $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P} \xrightarrow{\lambda} \langle \mathfrak{G}', \phi'\rangle \triangleright \mathsf{P}'$.

- Rule (SetLocal). In this case, we have: $\mathsf{P} = \mathsf{x} \leftarrow \mathsf{e}$ and $\mathsf{P}' = \mathsf{skip}$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (Assign), it follows that $\Gamma_1 \vdash \mathsf{e} : \delta$, for some δ, and $\Gamma_2 = \Gamma_1[\mathsf{x} \mapsto \delta \sqcup \sigma]$. Thus, we can set $\Gamma_1' = \Gamma_2$ to derive $\sigma \vdash \Gamma_1'\{\mathsf{skip}\}\Gamma_2$ by an application of the typing rule (Skip).
- Rule (StopListening). In this case, we have $\mathsf{P} = \mathsf{listen}(\mathsf{L})$ and $\mathsf{P}' = \mathsf{skip}$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (Listen), it follows that $\Gamma_1 = \Gamma_2$. Thus, setting $\Gamma_1' = \Gamma_1$, we can derive $\sigma \vdash \Gamma_1'\{\mathsf{skip}\}\Gamma_2$ by an application of the typing rule (Skip).
- Rule (Update). In this case, we have $\mathsf{P} = \mathsf{update}(\mathsf{x})$, for some service x, and $\mathsf{P}' = \mathsf{skip}$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (Update), it follows that $\Gamma_1 = \Gamma_2$. Thus, setting $\Gamma_1' = \Gamma_1 = \Gamma_2$, we can derive $\sigma \vdash \Gamma_1'\{\mathsf{skip}\}\Gamma_2$ by an application of the typing rule (Skip).
- Rule (SkipUpdate). In this case, we have $\mathsf{P} = \mathsf{update}(\mathsf{x})$ and $\mathsf{P}' = \mathsf{skip}$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (Update), it follows that $\Gamma_1 = \Gamma_2$. Thus, setting $\Gamma_1' = \Gamma_2$, we can derive $\sigma \vdash \Gamma_1'\{\mathsf{skip}\}\Gamma_2$ by an application of the typing rule (Skip).
- Rule (IfTrue). In this case, we have $\mathsf{P} = \mathsf{if\ b\ then\ }\{\mathsf{P}_1\}\mathsf{\ else\ }\{\mathsf{P}_2\}$ and $\mathsf{P}' = \mathsf{P}_1$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (IfElse) we have that $\Gamma_1 \vdash b : \delta$, for some δ, and $\sigma \sqcup \delta \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_2$. Thus, setting $\Gamma_1' = \Gamma_1$, since $\sigma \preccurlyeq \sigma \sqcup \delta$, by an application of the sub-typing rule (Sub.Proc), we derive $\sigma \vdash \Gamma_1'\{\mathsf{P}_1\}\Gamma_2$.
- Rule (IfFalse). Similar to the previous case.
- Rule (Seq). In this case, $\mathsf{P} = \mathsf{P}_1;\mathsf{P}_2$, for some P_1 and P_2, $\mathsf{P}' = \mathsf{P}_1';\mathsf{P}_2$, for some P_1' such that $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}', \phi'\rangle \triangleright \mathsf{P}_1'$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (Seq) there is a type environment Γ_3 such that $\sigma \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_3$ and $\sigma \vdash \Gamma_3\{\mathsf{P}_2\}\Gamma_2$. Since $\sigma \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_3$ and the depth of 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 that of the derivation tree for $\langle \mathfrak{G}, \phi\rangle \triangleright \mathsf{P} \xrightarrow{\lambda} \langle \mathfrak{G}', \phi'\rangle \triangleright \mathsf{P}'$, by inductive hypothesis, we derive that $\sigma \vdash \Gamma_1'\{\mathsf{P}_1'\}\Gamma_3$, for some Γ_1'. Finally, since $\sigma \vdash \Gamma_3\{\mathsf{P}_2\}\Gamma_2$, by an application of the typing rule (Seq) it follows that $\sigma \vdash \Gamma_1'\{\mathsf{P}_1';\mathsf{P}_2\}\Gamma_2$.
- Rule (SeqSkip). In this case, we have $\mathsf{P} = \mathsf{skip};\mathsf{Q}$, for some process Q, and $\mathsf{P}' = \mathsf{Q}$. As $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, by definition of the typing rule (Seq) (and (Skip)) we have that $\sigma \vdash \Gamma_1\{\mathsf{skip}\}\Gamma_1$ and $\sigma \vdash \Gamma_1\{\mathsf{Q}\}\Gamma_2$. Thus, setting $\Gamma_1' = \Gamma_1$, we derive $\sigma \vdash \Gamma_1'\{\mathsf{Q}\}\Gamma_2$.

- Rule (Fix). In this case, we have $\mathsf{P} = \mathsf{fix}\,\mathbb{X} \bullet \mathsf{Q}$, for some process Q, and $\mathsf{P}' = \mathsf{Q}\left\{{}^{\mathsf{fix}\,\mathbb{X}\bullet\mathsf{Q}}/_{\mathbb{X}}\right\}$. Thus, the typing $\sigma \vdash \Gamma_1\left\{\mathsf{P}\right\}\Gamma_2$ was derived by an application of the typing rule (Fix) of Table 3, under the hypotheses that: (i) $\Gamma_1 = \Gamma_2$, and (ii) $\sigma \vdash \hat{\Gamma}_1\left\{\mathsf{Q}\right\}\hat{\Gamma}_1$, for $\hat{\Gamma}_1 = \Gamma_1[\mathbb{X} \mapsto \sigma]$ and $\hat{\Gamma}_2 = \Gamma_2[\mathbb{X} \mapsto \sigma]$. By an application of Lemma 1, we derive that $\sigma \vdash \Gamma_1'\left\{\mathsf{Q}\left\{{}^{\mathsf{fix}\,\mathbb{X}\bullet\mathsf{Q}}/_{\mathbb{X}}\right\}\right\}\Gamma_2$, for $\Gamma_1' = \Gamma_1$.

□

In order to prove Theorem 3, we first prove the soundness of the security type system for single-app systems (Proposition 1) and then, relying on the fact that our bisimilarity $\approx_H^{\mathrm{ti}}$ is preserved by parallel composition (Lemma 8), we generalize the result to systems of apps.

First of all, we need to define a σ-equivalence relation on process configurations which, as we will show, is preserved by well-typed processes. In Definition 9 we already defined σ-equivalence among global stores, $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$. Similarly, we define σ-equivalence $\phi_1 \equiv_{\Gamma,\sigma} \phi_2$ on local stores ϕ_1 and ϕ_2, with respect to a local typing environment Γ and attacker level σ, such that for all local variables x, it holds that $\phi_1(\mathsf{x}) = \phi_2(\mathsf{x})$ whenever $\Gamma(\mathsf{x}) \preccurlyeq \sigma$.

In the following, for simplicity, we write σ^{H} to denote the level of a program counter *pc* whenever $pc \not\preccurlyeq \sigma$. When convenient, we avoid specifying the typing environments and simply assume that such environments exist.

Definition 14 (Process-level σ-equivalence)**.** Let $\Sigma \in \mathsf{Service} \to SL$ be a security policy and $\sigma \in SL$ a security clearance. The σ-equivalence relation $\equiv_\sigma$ between well-typed processes is defined as:

(1) $P \equiv_\sigma P$ for all well-formed processes P;
(2) if $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$ and $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_2\right\}\Gamma_2$, then $P_1 \equiv_\sigma P_2$;
(3) if $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$ and $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_2\right\}\Gamma_2$, then $P_1; P \equiv_\sigma P_2; P$ for all well-formed processes P;
(4) if $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$ then $P_1; P \equiv_\sigma P$ and $P \equiv_\sigma P_1; P$, for all well-formed processes P.

We say that two process configurations $\mathfrak{C}_1 = \langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_1$ and $\mathfrak{C}_2 = \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2$ are σ-equivalent, written $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$, only if: (i) $\mathsf{P}_1 \equiv_\sigma \mathsf{P}_2$, P_1 and P_2 well-typed, i.e., $\mu \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$, $\mu \vdash \Gamma_1'\left\{\mathsf{P}_2\right\}\Gamma_2$, for $\mu \in \{\sigma, \sigma^{\mathsf{H}}\}$ and some $\Gamma_1, \Gamma_2, \Gamma_1'$, and (ii) $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$, and (iii) $\phi_1 \equiv_{\Gamma_1 \sqcup \Gamma_1', \sigma} \phi_2$, where $\Gamma_1 \sqcup \Gamma_1'$ denote the pointwise join of the two security environments Γ_1 and Γ_1' used to type P_1 and P_2, respectively.

We use the invariant on process configurations to prove soundness of the security type system for single apps. We first state and prove a few helper lemmas.

Lemma 3 (Simple security)**.** *Let $\Sigma \in \mathsf{Service} \to SL$ be a security policy and $\sigma \in SL$ a security clearance. If $\Gamma \vdash e : \rho$, $\rho \preccurlyeq \sigma$, $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$, and $\phi_1 \equiv_{\Gamma,\sigma} \phi_2$, then $[\![e]\!](\mathfrak{G}_1, \phi_1) = [\![e]\!](\mathfrak{G}_2, \phi_2)$.*

Proof. By structural induction on e, typing rules for expressions, and σ-equivalence for stores. □

Now, in Lemma 4 we show that configurations containing processes that are typed in a high context σ^{H} preserve σ-equivalence with respect to the initial configuration and do not produce any update events at security levels below σ. Intuitively, the lemma holds because the type system prevents implicit flows, i.e., updates to attacker-visible services in a high context.

As shown in Table 1, (the code of) an app can only produce actions $\lambda \in \mathcal{L}$. In the following lemmas (precisely, Lemmas 4, 5 and 6) we assume a set $H \stackrel{\text{def}}{=} \{\lambda \in \mathcal{L} \mid \Sigma(\lambda) \not\preccurlyeq \sigma\} \cup \{\tau\}$ of non-observable actions at the attacker's security level σ. We also lift the definition of transition for process configurations (Table 1) from actions to traces of actions $t = \lambda_1 \cdots \lambda_n$, writing $\stackrel{t}{\twoheadrightarrow}$ as an abbreviation for $\stackrel{\lambda_1}{\twoheadrightarrow} \ldots \stackrel{\lambda_n}{\twoheadrightarrow}$; we also write $t \in H$ whenever, for all λ_i, for $1 \le i \le n$, we have $\lambda_i \in H$.

Lemma 4 (High-steps invariant). *If* $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$ *then for all* $\mathfrak{G}_1, \phi_1 \in \mathbb{S}$, *if* $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_2$, *then there is* Γ_1' *such that* $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_2\right\}\Gamma_2$, $\lambda \in H$, *and* $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \equiv_\sigma \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_2$.

Proof. By rule induction on how the transition $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_2$ was derived. Subject reduction (Lemma 2) ensures that there exists Γ_1' such that $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_2\right\}\Gamma_2$. Therefore, we have to show that an execution step in a high context σ^{H} preserves the invariant $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \equiv_\sigma \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_2$, namely $\mathsf{P}_1 \equiv_\sigma \mathsf{P}_2$, $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$, $\phi_1 \equiv_{\Gamma_1 \sqcup \Gamma_1', \sigma} \phi_2$, and $\lambda \in H$. By Definition 14, $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$ and $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_2\right\}\Gamma_2$, we have that $\mathsf{P}_1 \equiv_\sigma \mathsf{P}_2$ since the program counter is σ^{H}. Hence, we only need to show that σ-equivalence on global and local stores is preserved and that $\lambda \in H$. We show the interesting cases for the rules in Table 1 that may change either the global store or the local store.

- Rule (SetLocal), $\mathsf{P}_1 = \mathsf{x} \leftarrow \mathsf{e}$. The rule sets $\phi_2 = \phi_1[\mathsf{x} \mapsto [\![\mathsf{e}]\!](\mathfrak{G}, \phi)]$ and $\lambda = \tau \in H$, while $\mathfrak{G}_1 = \mathfrak{G}_2$. We show that $\phi_2 \equiv_{\Gamma_1 \sqcup \Gamma_1', \sigma} \phi_1[\mathsf{x} \mapsto [\![\mathsf{e}]\!](\mathfrak{G}, \phi)]$. As $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$, by definition of the typing rule (Assign), it follows that $\Gamma_1 \vdash \mathsf{e} : \delta$, for some δ, and $\Gamma_1' = \Gamma_1[\mathsf{x} \mapsto \delta \sqcup \sigma^{\mathsf{H}}] = \Gamma_1[\mathsf{x} \mapsto \sigma^{\mathsf{H}}]$. Hence, since $\Gamma_1'(\mathsf{x}) = \sigma^{\mathsf{H}}$ and $\Gamma_1 \sqcup \Gamma_1' = \Gamma_1'$, we have $\phi_1 \equiv_{\Gamma_1', \sigma} \phi_2$.
- Rule (Update), $\mathsf{P}_1 = \mathsf{update}(\mathsf{x})$. The rule sets $\mathfrak{G}_2 = \mathfrak{G}_1[\mathsf{x} \mapsto \phi_1(x)]$ and produces an action $\lambda = \mathsf{x}!v$, while $\phi_1 = \phi_2$. As $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}\right\}\Gamma_2$, by definition of the typing rule (Update), it follows that $\sigma^{\mathsf{H}} \sqcup \Gamma_1(\mathsf{x}) \preccurlyeq \Sigma(\mathsf{x})$, implying that $\sigma^{\mathsf{H}} \preccurlyeq \Sigma(\mathsf{x})$, hence $\lambda \in H$. Moreover, Lemma 2 (subject reduction), in the case of an update construct, ensures that $\Gamma_1' = \Gamma_1$, hence $\phi_1 \equiv_{\Gamma_1', \sigma} \phi_2$. Finally, $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$ since $H \preccurlyeq \Sigma(\mathsf{x})$ and $\sigma \preccurlyeq H$, hence updates to high-level services may differ.
- Rule (Seq), with $\mathsf{P}_1 = \mathsf{P}_a; \mathsf{P}_b$ and $\mathsf{P}_2 = \mathsf{P}_a'; \mathsf{P}_b$, for some P_a' such that $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_a \xrightarrow{\lambda} \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_a'$. As $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$, by definition of the typing rule (Seq) there exists Γ_3 such that $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_a\right\}\Gamma_3$ and $\sigma^{\mathsf{H}} \vdash \Gamma_3\left\{\mathsf{P}_b\right\}\Gamma_2$. Since $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_a\right\}\Gamma_3$ and $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_a \xrightarrow{\lambda} \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_a'$, by induction hypothesis, there exists Γ_1' such that $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_a'\right\}\Gamma_3$, $\lambda \in H$, $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$, and $\phi_1 \equiv_{\Gamma_1 \sqcup \Gamma_1', \sigma} \phi_2$. By Definition 14, it follows that $\mathsf{P}_1 \equiv_\sigma \mathsf{P}_2$, Finally, by two different applications of the typing rule (Seq) we derive: (i) $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_a; \mathsf{P}_b\right\}\Gamma_2$, and (ii) $\sigma^{\mathsf{H}} \vdash \Gamma_1'\left\{\mathsf{P}_a'; \mathsf{P}_b\right\}\Gamma_2$. As a consequence, by Definition 14 it follows that $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \equiv_\sigma \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_2$, as required.

□

Corollary 5 (High-traces invariant). *If* $\sigma^{\mathsf{H}} \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$ *and* $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \xrightarrow{t} \langle \mathfrak{G}_2, \phi_2\rangle \triangleright$ *skip, for some trace* t, *then* $t \in H$ *and* $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \equiv_\sigma \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{skip}$.

Proof. By induction on the length of the trace t, and then by Lemma 4. □

We now show that σ-equivalent process configurations that may differ only on global and local stores but not on the process code, mimic each others' actions, while preserving σ-equivalence.

Lemma 5 (Low-steps invariant). *For any* $\mathsf{P}_1, \mathfrak{G}_1, \mathfrak{G}_2, \phi_1, \phi_2, \mathfrak{C}_1'$, *if* $\sigma \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$, $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \equiv_\sigma \langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_1$ *and* $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda_1} \langle \mathfrak{G}_1', \phi_1'\rangle \triangleright \mathsf{P}_1' = \mathfrak{C}_1'$, *then there exists* $\mathfrak{C}_2' = \langle \mathfrak{G}_2', \phi_2'\rangle \triangleright \mathsf{P}_2'$ *such that* $\langle \mathfrak{G}_2, \phi_2\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda_2} \mathfrak{C}_2'$, $\mathfrak{C}_1' \equiv_\sigma \mathfrak{C}_2'$, $\mathsf{P}_1' = \mathsf{P}_2'$, *and, if* $\lambda_1 \notin H$ *then* $\lambda_1 = \lambda_2$.

Proof. By rule induction on how the transition $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}_1', \phi_1'\rangle \triangleright \mathsf{P}_1'$ was derived. Since $\sigma \vdash \Gamma_1\left\{\mathsf{P}_1\right\}\Gamma_2$, by Lemma 2 (subject reduction) we have $\sigma \vdash \Gamma_1'\left\{\mathsf{P}_1'\right\}\Gamma_2$. By assumption $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$ and $\phi_1 \equiv_{\Gamma_1,\sigma} \phi_2$. We show the most interesting cases, i.e., the rules that may change either the global store or the local store.

- Rule (SetLocal), $\mathsf{P}_1 = \mathsf{x} \leftarrow \mathsf{e}$: By definition $\langle \mathfrak{G}_1, \phi_1\rangle \triangleright (\mathsf{x} \leftarrow \mathsf{e}) \xrightarrow{\tau} \langle \mathfrak{G}_1, \phi_1[\mathsf{x} \mapsto [\![\mathsf{e}]\!](\mathfrak{G}_1, \phi_1)]\rangle \triangleright \mathsf{skip}$ and $\langle \mathfrak{G}_2, \phi_2\rangle \triangleright (\mathsf{x} \leftarrow \mathsf{e}) \xrightarrow{\tau} \langle \mathfrak{G}_2, \phi_2[\mathsf{x} \mapsto [\![\mathsf{e}]\!](\mathfrak{G}_2, \phi_2)]\rangle \triangleright \mathsf{skip}$. Hence, $\mathsf{P}_1' = \mathsf{P}_2' = \mathsf{skip}$ and $\sigma \vdash \Gamma_1'\left\{\mathsf{skip}\right\}\Gamma_2 \equiv_\sigma \sigma \vdash \Gamma_1'\left\{\mathsf{skip}\right\}\Gamma_2$ and $\lambda_1 = \lambda_2 = \tau \in H$. Moreover, $\mathfrak{G}_1' \equiv_{\Sigma,\sigma} \mathfrak{G}_2'$ because the

rule does not change the global stores ($\mathfrak{G}_1 = \mathfrak{G}_1'$ and $\mathfrak{G}_2 = \mathfrak{G}_2'$) and the security environment Σ is fixed. It remains to show that $\phi_1' \equiv_{\Gamma_1',\sigma} \phi_2'$. By definition of the typing rule (Assign), it follows that $\Gamma_1 \vdash \mathsf{e} : \rho$, for some ρ, and $\Gamma_1' = \Gamma_1[\mathsf{x} \mapsto \rho \sqcup \sigma]$. We distinguish two cases: (1) $\rho \preccurlyeq \sigma$ and (2) $\rho \not\preccurlyeq \sigma$.

(1) $\rho \preccurlyeq \sigma$: From Lemma 3 it follows that $[\![\mathsf{e}]\!](\mathfrak{G}_1, \phi_1) = [\![\mathsf{e}]\!](\mathfrak{G}_2, \phi_2)$, which implies that $[\![\mathsf{x}]\!](\mathfrak{G}_1', \phi_1') = [\![\mathsf{x}]\!](\mathfrak{G}_2', \phi_2')$, hence $\phi_1' \equiv_{\Gamma_1',\sigma} \phi_2'$.

(2) Case (2): If $\rho \not\preccurlyeq \sigma$, then $\Gamma_1'(\mathsf{x}) \not\preccurlyeq \sigma$, hence $\phi_1' \equiv_{\Gamma_1',\sigma} \phi_2'$.

- Rule (Update), $\mathsf{P}_1 = \mathsf{update}(\mathsf{x})$: By definition $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{update}(\mathsf{x}) \xrightarrow{\mathsf{x}!v_1} \langle \mathfrak{G}[\mathsf{x} \mapsto v_1], \phi_1 \rangle \triangleright \mathsf{skip}$ and $\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{update}(\mathsf{x}) \xrightarrow{\mathsf{x}!v_2} \langle \mathfrak{G}[\mathsf{x} \mapsto v_2], \phi_2 \rangle \triangleright \mathsf{skip}$. By Lemma 2 (subject reduction) and the typing rule (Update), we have $\Gamma_1 = \Gamma_2 = \Gamma_1'$ and $\sigma \vdash \Gamma_1' \{\mathsf{skip}\} \Gamma_2 \equiv_\sigma \sigma \vdash \Gamma_1' \{\mathsf{skip}\} \Gamma_2$. Moreover, $\phi_1' \equiv_{\Gamma_1',\sigma} \phi_2'$ because the rule does not change the local stores ($\phi_1 = \phi_1'$ and $\phi_2 = \phi_2'$) and $\Gamma_1 = \Gamma_1'$. It remains to show that $\mathfrak{G}_1' \equiv_{\Sigma,\sigma} \mathfrak{C}_2'$ and $\mathsf{x}!v_1 = \mathsf{x}!v_2$ whenever $\mathsf{x}!v_1 \notin H$. By definition of the typing rule (Update), it holds that $\sigma \sqcup \Gamma_1(\mathsf{x}) \preccurlyeq \Sigma(\mathsf{x})$. We distinguish two cases: (1) $\Gamma_1(\mathsf{x}) \preccurlyeq \sigma = \Sigma(\mathsf{x})$; (2) $\Gamma_1(\mathsf{x}) \not\preccurlyeq \sigma$ and $\Gamma_1(\mathsf{x}) \preccurlyeq \Sigma(\mathsf{x})$.

(1) $\Gamma_1(\mathsf{x}) \preccurlyeq \sigma = \Sigma(\mathsf{x})$: In this case $\mathsf{x}!v_1, \mathsf{x}!v_2 \notin H$. It follows that $[\![\mathsf{x}]\!](\mathfrak{G}_1, \lambda\mathsf{x}\,.\,\bot) = [\![\mathsf{x}]\!](\mathfrak{G}_2, \lambda\mathsf{x}\,.\,\bot)$ since $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$, which implies that $v_1 = v_2$. Moreover, $\mathfrak{G}_1' \equiv_{\Sigma,\sigma} \mathfrak{G}_2'$ since $\mathfrak{G}_1'(\mathsf{x}) = \mathfrak{G}_2'(\mathsf{x}) = v_1$ and $\mathfrak{G}_1 \equiv_{\Sigma,\sigma} \mathfrak{G}_2$ by assumption.

(2) $\Gamma_1(\mathsf{x}) \not\preccurlyeq \sigma$ and $\Gamma_1(\mathsf{x}) \preccurlyeq \Sigma(\mathsf{x})$: The former implies that $\mathsf{x}!v_1, \mathsf{x}!v_2 \in H$ and the latter ensures that $\mathfrak{G}_1' \equiv_{\Sigma,\sigma} \mathfrak{G}_2'$.

- Rule (Seq), with $\mathsf{P}_1 = \mathsf{P}_a; \mathsf{P}_b$ and $\mathsf{P}_1' = \mathsf{P}_a'; \mathsf{P}_b$, for some P_a' s.t. $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_a \xrightarrow{\lambda_1} \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_a' = \mathfrak{C}_a'$. As $\sigma \vdash \Gamma_1 \{\mathsf{P}_a; \mathsf{P}_b\} \Gamma_2$, by definition of the typing rule (Seq) there exists Γ_3 such that $\sigma \vdash \Gamma_1 \{\mathsf{P}_a\} \Gamma_3$ and $\sigma \vdash \Gamma_3 \{\mathsf{P}_b\} \Gamma_2$. Since $\sigma \vdash \Gamma_1 \{\mathsf{P}_a\} \Gamma_3$ and $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_a \xrightarrow{\lambda_1} \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_a'$, by Lemma 2 (subject reduction) there exists Γ_1' such that $\sigma \vdash \Gamma_1' \{\mathsf{P}_a'\} \Gamma_3$. Then, by induction hypothesis we have that $\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_a \xrightarrow{\lambda_2} \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_a'' = \mathfrak{C}_a''$ such that $\mathfrak{C}_a' \equiv_\sigma \mathfrak{C}_a''$, $\mathsf{P}_a' = \mathsf{P}_a''$, and $\lambda_1 = \lambda_2$, whenever $\lambda_1 \notin H$. In particular, since $P_a' = P_a''$ and by subject reduction we can derive $\sigma \vdash \Gamma_1' \{\mathsf{P}_a''\} \Gamma_3$, it follows that: (i) $\mathsf{P}_a' \equiv_\sigma \mathsf{P}_a''$, (ii) $\mathfrak{G}_1' \equiv_{\Sigma,\sigma} \mathfrak{G}_2'$, and (iii) $\phi_1' \equiv_{\Gamma_1',\sigma} \phi_2'$. By two different applications of the typing rule (Seq) we derive: (i) $\sigma \vdash \Gamma_1' \{\mathsf{P}_a''; \mathsf{P}_b\} \Gamma_2$, and (ii) $\sigma \vdash \Gamma_1' \{\mathsf{P}_a''; \mathsf{P}_b\} \Gamma_2$. Finally, since $\mathsf{P}_a' = \mathsf{P}_a''$ then by Definition 14, it follows that $\mathsf{P}_a'; \mathsf{P}_b \equiv_\sigma \mathsf{P}_a''; \mathsf{P}_b$.

□

We now prove our crucial lemma saying that two σ-equivalent process configurations can mimic each others' actions, according to our notion of termination-insensitive hiding bisimulation.

Lemma 6 (Bisimulation step). *Let $H \stackrel{\text{def}}{=} \{\lambda \in \mathcal{L} \mid \Sigma(\lambda) \not\preccurlyeq \sigma\} \cup \{\tau\}$ for some security clearance $\sigma \in SL$. Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be process configurations such that $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$. If there is $\mathfrak{C}_1'$ such that $\mathfrak{C}_1 \xrightarrow{\lambda} \mathfrak{C}_1'$, then there exists a process configuration $\mathfrak{C}_2'$ such that*

(1) *$\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$ and $\mathfrak{C}_1' \equiv_\sigma \mathfrak{C}_2'$, whenever $\lambda \in H$, or*

(2) *either (a) $\mathfrak{C}_2 \stackrel{\lambda}{\Rightarrow}_H \mathfrak{C}_2'$ and $\mathfrak{C}_1' \equiv_\sigma \mathfrak{C}_2'$ or (b) $\mathfrak{C}_2 \Uparrow_H$, whenever $\lambda \notin H$.*

Proof. Let $\mathfrak{C}_1 = \langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_1$, $\mathfrak{C}_2 = \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2$, and $\mathfrak{C}_1' = \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1'$. Since $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$ it follows that $\mathsf{P}_1 \equiv_\sigma \mathsf{P}_2$ (see Definition 14). We proceed by case analysis on why $\mathsf{P}_1 \equiv_\sigma \mathsf{P}_2$. In each of the possible four cases, we will show that there is a process configuration $\mathfrak{C}_2' = \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_2'$, satisfying conditions (1) and (2) of the lemma.

- Case $\mathsf{P}_1 = \mathsf{P}_2$. By Lemma 5, there exists $\mathfrak{C}_2'$ such that $\mathfrak{C}_2 \xrightarrow{\lambda} \mathfrak{C}_2'$ and $\mathfrak{C}_1' \equiv_\sigma \mathfrak{C}_2'$. If $\lambda \in H$ then $\mathfrak{C}_2 \rightarrow_H \mathfrak{C}_2'$, and condition (1) holds, otherwise if $\lambda \notin H$ then condition (2a) holds.

- Case $\sigma^{\mathsf{H}} \vdash \Gamma_1\{\mathsf{P}_1\}\Gamma_2$ and $\sigma^{\mathsf{H}} \vdash \Gamma_1'\{\mathsf{P}_2\}\Gamma_2$. By Lemma 4, we have that $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_1'$ and $\lambda \in H$. Then, from the assumption $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$, and transitivity of $\equiv_\sigma$, it follows that $\mathfrak{C}_1' \equiv_\sigma \mathfrak{C}_2$. Thus, there is $\mathfrak{C}_2' = \mathfrak{C}_2$ such that $\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$ and condition (1) holds.
- Case $\mathsf{P}_1 = \mathsf{P}_3; \mathsf{P}$, $\mathsf{P}_2 = \mathsf{P}_4; \mathsf{P}$, $\sigma^{\mathsf{H}} \vdash \Gamma_1\{\mathsf{P}_3\}\Gamma_2$, and $\sigma^{\mathsf{H}} \vdash \Gamma_1''\{\mathsf{P}_4\}\Gamma_2$. We consider the shape of P_3. If $\mathsf{P}_3 = \mathsf{skip}$, then $\mathfrak{C}_1 \xrightarrow{\tau} \mathfrak{C}_1' = \langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}$ by an application of rule (SkipSeq), with $\tau \in H$. But then $\mathfrak{C}_1' = \langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}$ is σ-equivalent to $\mathfrak{C}_2 = \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright (\mathsf{P}_4; \mathsf{P})$ because $\mathsf{P} \equiv_\sigma \mathsf{P}_4; \mathsf{P}$ (Definition 14), and the global and local stores are equivalent by assumption. Hence, condition (1) holds for $\mathfrak{C}_2' = \mathfrak{C}_2$ and $\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$.

  Otherwise, if $\mathsf{P}_3 \neq \mathsf{skip}$, then $\mathfrak{C}_1 \xrightarrow{\lambda} \mathfrak{C}_1' = \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright (\mathsf{P}_3'; \mathsf{P})$ follows because $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_3 \xrightarrow{\lambda} \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_3'$, by an application of the rule (Seq). By Lemma 4, $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_3 \equiv_\sigma \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_3'$ and there exists Γ_1' such that $\sigma^{\mathsf{H}} \vdash \Gamma_1'\{\mathsf{P}_3'\}\Gamma_2$. The latter property and Definition 14 imply that $\mathsf{P}_3'; \mathsf{P} \equiv_\sigma \mathsf{P}_4; \mathsf{P}$. Moreover, the assumption $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$, i.e., $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright (\mathsf{P}_3; \mathsf{P}) \equiv_\sigma \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright (\mathsf{P}_4; \mathsf{P})$, and the transitivity of σ-equivalence over the global and local stores imply that $\mathfrak{C}_1' = \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright (\mathsf{P}_3'; \mathsf{P}) \equiv_\sigma \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright (\mathsf{P}_4; \mathsf{P}) = \mathfrak{C}_2$. As a result, condition (1) holds for $\mathfrak{C}_2' = \mathfrak{C}_2$ and $\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$.
- Case $\mathsf{P}_1 = \mathsf{P}_3; \mathsf{P}_2$ and $\sigma^{\mathsf{H}} \vdash \Gamma_1\{\mathsf{P}_3\}\Gamma_2$. As in the previous case, we consider the shape of P_3. If $\mathsf{P}_3 = \mathsf{skip}$, then $\mathfrak{C}_1 = \langle \mathfrak{G}_1, \phi_1 \rangle \triangleright (\mathsf{skip}; \mathsf{P}_2) \xrightarrow{\tau} \langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_2 = \mathfrak{C}_1'$ by an application of the rule (SkipSeq), with $\tau \in H$. But then $\mathfrak{C}_1'$ is σ-equivalent to $\mathfrak{C}_2$ by Definition 14 and the assumption that the global and local stores are equivalent. Hence, condition (1) holds for $\mathfrak{C}_2' = \mathfrak{C}_2$ and $\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$.

  Otherwise, if $\mathsf{P}_3 \neq \mathsf{skip}$, then $\mathfrak{C}_1 \xrightarrow{\lambda} \mathfrak{C}_1' = \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright (\mathsf{P}_3'; \mathsf{P}_2)$ follows because $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_3 \xrightarrow{\lambda} \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_3'$, by an application of rule (Seq). By Lemma 4, $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_3 \equiv_\sigma \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_3'$, $\lambda \in H$, and there exists Γ' such that $\mathsf{H} \vdash \Gamma_1'\{\mathsf{P}_3'\}\Gamma_2$. The latter property and Definition 14 imply that $\mathsf{P}_3'; \mathsf{P}_2 \equiv_\sigma \mathsf{P}_4; \mathsf{P}_2$. Moreover, the assumption $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$, i.e., $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright (\mathsf{P}_3; \mathsf{P}_2) \equiv_\sigma \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2$ and the transitivity of σ-equivalence over the global and local stores imply that $\langle \mathfrak{G}_1', \phi_1' \rangle \triangleright (\mathsf{P}_3'; \mathsf{P}_2) \equiv_\sigma \langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2$, i.e., $\mathfrak{C}_1' \equiv_\sigma \mathfrak{C}_2$. As a result, condition (1) holds for $\mathfrak{C}_2' = \mathfrak{C}_2$ and $\mathfrak{C}_2 \Rightarrow_H \mathfrak{C}_2'$.
- Case $\mathsf{P}_2 = \mathsf{P}_3; \mathsf{P}_1$ and $\sigma^{\mathsf{H}} \vdash \Gamma_1\{\mathsf{P}_3\}\Gamma_2$. We distinguish two cases. First, if there exists a configuration $\mathfrak{C}_2'' = \langle \mathfrak{G}_2'', \phi_2'' \rangle \triangleright \mathsf{P}_1$ such that $\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_3; \mathsf{P}_1 \xrightarrow{t} \langle \mathfrak{G}_2'', \phi_2'' \rangle \triangleright \mathsf{P}_1$ for some trace t, then by Corollary 5 and definition of rule (SeqSkip), it holds that $t \in H$, and hence $\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_3; \mathsf{P}_1 \Rightarrow_H \langle \mathfrak{G}_2'', \phi_2'' \rangle \triangleright \mathsf{P}_1$. Moreover, by Corollary 5 we also have that $\mathfrak{G}_2 \equiv_{\Sigma,\sigma} \mathfrak{G}_2''$ and $\phi_2 \equiv_{\Gamma_2,\sigma} \phi_2''$. By the assumption $\mathfrak{C}_1 \equiv_\sigma \mathfrak{C}_2$, Definition 14, and transitivity of σ-equivalence on global and local stores, it follows that $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_1 \equiv_\sigma \langle \mathfrak{G}_2'', \phi_2'' \rangle \triangleright \mathsf{P}_1$. By Lemma 5, there exists $\langle \mathfrak{G}_2'', \phi_2'' \rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda} \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_1'$ such that $\langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1' \equiv_\sigma \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_1'$. If $\lambda \in H$ then condition (1) holds, otherwise if $\lambda \notin H$ then condition (2a) holds.
  Otherwise, if there exists no such configuration $\mathfrak{C}_2''$ then $\mathfrak{C}_2 \Uparrow_H$, then condition (2b) is satisfied.

□

Now we are ready to prove a special case of Theorem 3 for a system that contains only one app.

Proposition 1 (Soundness of security types for single-app systems). *Let Σ be a security policy, $\mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$ a well-formed app, and $H \stackrel{\text{def}}{=} \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\} \cup \{\tau\}$ the set of all possible non-observable system actions with respect to a security clearance $\sigma \in SL$. Let $\mathfrak{G}_a, \mathfrak{G}_b \in \mathbb{S}_\perp$ be two arbitrary global stores such that $\mathfrak{G}_a \equiv_{\Sigma,\sigma} \mathfrak{G}_b$ and $\mathfrak{L}_a, \mathfrak{L}_b \in \mathcal{I} \to \mathbb{S}$ be two arbitrary local stores such that $\mathfrak{L}_a(\mathtt{id}) \equiv_{\Gamma_1,\sigma} \mathfrak{L}_b(\mathtt{id})$ for some Γ_1. If $\sigma \vdash \Gamma_1\{\mathsf{P}\}\Gamma_2$, for some Γ_2, then $\langle \mathfrak{G}_a, \mathfrak{L}_a \rangle \triangleright \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,] \approx_H^{\text{ti}} \langle \mathfrak{G}_b, \mathfrak{L}_b \rangle \triangleright \mathtt{id}[\,\mathsf{D} \bowtie \mathsf{P}\,]$.*

Proof. Let $\mathbb{C}_a \stackrel{\text{def}}{=} \langle \mathfrak{G}_a, \mathfrak{L}_a \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P} \,]$ and $\mathbb{C}_b \stackrel{\text{def}}{=} \langle \mathfrak{G}_b, \mathfrak{L}_b \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P} \,]$. Let $\mathcal{R}$ be a binary relation over configurations, defined as follows:

$$\mathcal{R} \stackrel{\text{def}}{=} \left\{ \left(\mathbb{C}_1, \mathbb{C}_2 \right) : \mathbb{C}_i \stackrel{\text{def}}{=} \langle \mathfrak{G}_i, \mathfrak{L}_i \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_i \,] \ \wedge \ \sigma \vdash \Gamma_1^i \left\{ \mathsf{P}_i \right\} \Gamma_2^i \, , i \in \{1,2\} \ \wedge \ \mathbb{C}_1 \equiv_\sigma \mathbb{C}_2 \right\}.$$

By definition, $(\mathbb{C}_a, \mathbb{C}_b) \in \mathcal{R}$. We will prove that $\mathcal{R}$ is a termination-insensitive hiding bisimulation parametric on H. Let $(\mathbb{C}_1, \mathbb{C}_2) \in \mathcal{R}$ and $\mathbb{C}_1 \xrightarrow{\alpha} \mathbb{C}_1'$, for some action α, and some configuration $\mathbb{C}_1' = \langle \mathfrak{G}_1', \mathfrak{L}_1' \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_1' \,]$. By Lemma 2 (subject reduction for processes), from $\sigma \vdash \Gamma_1^1 \left\{ \mathsf{P}_1 \right\} \Gamma_2^1$ it follows that $\sigma \vdash \Gamma_1' \left\{ \mathsf{P}_1' \right\} \Gamma_2^1$, for some Γ_1'. The proof proceeds by case analysis on the action α to show that there is a configuration $\mathbb{C}_2'$ such that $\mathbb{C}_2 \stackrel{\hat{\alpha}}{\Rightarrow}_H \mathbb{C}_2'$, with $(\mathbb{C}_1', \mathbb{C}_2') \in \mathcal{R}$.

- Let $\alpha \in H$, $\alpha \neq \mathtt{x}?v$. Then, $\mathbb{C}_1 \xrightarrow{\alpha} \mathbb{C}_1' = \langle \mathfrak{G}_1', \mathfrak{L}_1' \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_1' \,]$ follows because $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda}_\triangleright \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1'$, with $\mathfrak{L}_1(\mathtt{id}) = \phi_1$ and $\mathfrak{L}_1'(\mathtt{id}) = \phi_1'$, by an application of either rule (App) or rule (AppUpdate) of Table 2. In particular, in the first case, $\lambda = \tau$, whereas, in the second case $\lambda = \mathtt{x}!v$, for $\alpha = \mathtt{id}{:}\mathtt{x}!v$. Let $\mathfrak{L}_2(\mathtt{id}) = \phi_2$. Since $\mathbb{C}_1 \equiv_\sigma \mathbb{C}_2$, by an application of Lemma 6 there is a weak transition $\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2 \Rightarrow_H \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_2'$ such that $\langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1' \equiv_\sigma \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_2'$. As $\sigma \vdash \Gamma_1^2 \left\{ \mathsf{P}_2 \right\} \Gamma_2^2$, for some Γ_1^2 and Γ_2^2, by several applications of Lemma 2 we know that $\sigma \vdash \Gamma \left\{ \mathsf{P}_2' \right\} \Gamma'$, for some Γ and Γ'. Thus, by several applications of the transition rules (App) and (AppUpdate) mentioned before there is $\mathbb{C}_2' = \langle \mathfrak{G}_2', \mathfrak{L}_2' \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_2' \,]$, with $\mathfrak{L}_2'(\mathtt{id}) = \phi_2'$, such that $\mathbb{C}_2 \Rightarrow_H \mathbb{C}_2'$. As $\langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1' \equiv_\sigma \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_2'$, it follows that $\mathbb{C}_1' \equiv_\sigma \mathbb{C}_2'$. As a consequence, $(\mathbb{C}_1', \mathbb{C}_2') \in \mathcal{R}$, as required.
- Let $\alpha \notin H$, $\alpha \neq \mathtt{x}?v$. Then, $\mathbb{C}_1 \xrightarrow{\alpha} \mathbb{C}_1' = \langle \mathfrak{G}_1', \mathfrak{L}_1' \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_1' \,]$ follows because $\langle \mathfrak{G}_1, \phi_1 \rangle \triangleright \mathsf{P}_1 \xrightarrow{\lambda}_\triangleright \langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1'$, with $\mathfrak{L}_1(\mathtt{id}) = \phi_1$ and $\mathfrak{L}_1'(\mathtt{id}) = \phi_1'$, by an application of either rule (App) or rule (AppUpdate) of Table 2. In particular, in the first case, $\lambda = \tau$, whereas, in the second case $\lambda = \mathtt{x}!v$, for $\alpha = \mathtt{id}{:}\mathtt{x}!v$. Let $\mathfrak{L}_2(\mathtt{id}) = \phi_2$. Since $\mathbb{C}_1 \equiv_\sigma \mathbb{C}_2$, by Lemma 6 there are two possibilities:
  - either there is a weak transition $\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2 \Rightarrow_H \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_2'$ such that $\langle \mathfrak{G}_1', \phi_1' \rangle \triangleright \mathsf{P}_1' \equiv_\sigma \langle \mathfrak{G}_2', \phi_2' \rangle \triangleright \mathsf{P}_2'$, and we proceed as in the previous case;
  - or $(\langle \mathfrak{G}_2, \phi_2 \rangle \triangleright \mathsf{P}_2) \Uparrow_H$, which entails $\mathbb{C}_2 \Uparrow_H$, and we are done.
- Let $\alpha = \mathtt{x}?v$. The transition $\mathbb{C}_1 \xrightarrow{\alpha} \mathbb{C}_1'$ can be only derived by an application of the rule (EnvChange) in Table 2, where $\mathbb{C}_1' = \langle \mathfrak{G}_1[\mathtt{x} \mapsto v], \mathfrak{L}_1 \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_1 \,]$. As already pointed out, this action denotes a modification of the cloud made by the external observer with no requirements. Thus, $\mathbb{C}_2 \xrightarrow{\mathtt{x}?v} \mathbb{C}_2'$, with $\mathbb{C}_2' = \langle \mathfrak{G}_2[\mathtt{x} \mapsto v], \mathfrak{L}_2 \rangle \triangleright \mathtt{id}[\, \mathsf{D} \bowtie \mathsf{P}_2 \,]$, where $\mathbb{C}_1' \equiv_\sigma \mathbb{C}_2'$ easily follows from $\mathbb{C}_1 \equiv_\sigma \mathbb{C}_2$. As a consequence, if $\alpha \in H$ then $\mathbb{C}_2 \rightarrow_H \mathbb{C}_2'$, otherwise, if $\alpha \notin H$ then $\mathbb{C}_2 \stackrel{\mathtt{x}?v}{\Longrightarrow}_H \mathbb{C}_2'$. As the action $\mathtt{x}?v$ only affects the global stores of $\mathbb{C}_1$ and $\mathbb{C}_2$ in a consistent manner, leaving both local stores and systems unchanged, it follows that $(\mathbb{C}_1', \mathbb{C}_2') \in \mathcal{R}$.

□

Once we have proved the soundness of our security type system restricted to a single app we can rely on the compositionality of our termination-insensitive hiding bisimilarity, $\approx_H^{\text{ti}}$, to lift the result to systems of apps. Before that we need an easy technical result on weakening of local stores.

Lemma 7 (Store weakening). *Let Σ be a security policy, $\mathtt{id}_1[\, \mathsf{D}_1 \bowtie \mathsf{P}_1 \,]$ and $\mathtt{id}_2[\, \mathsf{D}_2 \bowtie \mathsf{P}_2 \,]$ be two well-formed apps, and $H = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\preccurlyeq \sigma\} \cup \{\tau\}$ the set of all possible non-observable actions with respect to a security clearance $\sigma \in SL$. Let $\mathfrak{G}_1, \mathfrak{G}_2 \in \mathbb{S}_\perp$ be two arbitrary global stores. Let $\mathfrak{L}_i \in \{\mathtt{id}_i\} \rightarrow \mathbb{S}$, for $i \in \{1,2\}$, be two arbitrary local stores and $\mathfrak{L} \in \mathcal{I}_\mathsf{S} \rightarrow \mathbb{S}$ be an arbitrary local store defined for a set of app identifiers different from $\mathtt{id}_1$ and $\mathtt{id}_2$. Then, $\langle \mathfrak{G}_1, \mathfrak{L}_1 \rangle \triangleright \mathtt{id}_1[\, \mathsf{D}_1 \bowtie \mathsf{P}_1 \,] \approx_H^{\text{ti}} \langle \mathfrak{G}_2, \mathfrak{L}_2 \rangle \triangleright \mathtt{id}_2[\, \mathsf{D}_2 \bowtie \mathsf{P}_2 \,]$ if and only if $\langle \mathfrak{G}_1, \mathfrak{L}_1 \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_1[\, \mathsf{D}_1 \bowtie \mathsf{P}_1 \,] \approx_H^{\text{ti}} \langle \mathfrak{G}_2, \mathfrak{L}_2 \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_2[\, \mathsf{D}_2 \bowtie \mathsf{P}_2 \,]$.*

Proof. The proof is straightforward as, by definition, the two apps do not have interaction of any kind with the local store $\mathfrak{L}$. □

Lemma 8 ($\approx_H^{\text{ti}}$ under parallel composition). *Let Σ be a security policy, $\mathtt{id}_1[\,\mathsf{D}_1 \bowtie \mathsf{P}_1\,]$ and $\mathtt{id}_2[\,\mathsf{D}_2 \bowtie \mathsf{P}_2\,]$ be two well-formed apps, and $H = \{\alpha \in \mathcal{A} \mid \Sigma(\alpha) \not\leq \sigma\} \cup \{\tau\}$. Let S be a system of apps that contains neither $\mathtt{id}_1$ nor $\mathtt{id}_2$ and $\mathfrak{G}_1, \mathfrak{G}_2 \in \mathbb{S}_\perp$ be two arbitrary global stores. Let $\mathfrak{L}_i \in \{\mathtt{id}_i\} \to \mathbb{S}$, for $i \in \{1,2\}$, be two arbitrary local stores. Let $\mathfrak{L} \in \mathcal{I}_\mathsf{S} \to \mathbb{S}$ be an arbitrary local store defined for all app identifiers in S. If $\langle \mathfrak{G}_1, \mathfrak{L}_1 \rangle \triangleright \mathtt{id}_1[\,\mathsf{D}_1 \bowtie \mathsf{P}_1\,] \approx_H^{\text{ti}} \langle \mathfrak{G}_2, \mathfrak{L}_2 \rangle \triangleright \mathtt{id}_2[\,\mathsf{D}_2 \bowtie \mathsf{P}_2\,]$ then $\langle \mathfrak{G}_1, \mathfrak{L}_1 \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_1[\,\mathsf{D}_1 \bowtie \mathsf{P}_1\,] \parallel \mathsf{S}) \approx_H^{\text{ti}} \langle \mathfrak{G}_2, \mathfrak{L}_2 \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_2[\,\mathsf{D}_2 \bowtie \mathsf{P}_2\,] \parallel \mathsf{S})$.*

Proof. By relying on Lemma 7, we prove that the relation

$$\mathcal{R} \stackrel{\text{def}}{=} \left\{ \left(\langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,] \parallel \mathsf{R}),\ \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}) \right) : \mathfrak{C}_{ap} \approx_H^{\text{ti}} \mathfrak{C}_{bp},\ \mathsf{R} \text{ arbitrary} \right\}$$

is a termination-insensitive hiding bisimulation for $\mathfrak{C}_{ap} \stackrel{\text{def}}{=} \langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,]$, $\mathfrak{C}_{bp} \stackrel{\text{def}}{=} \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,]$, for apps $\mathtt{id}_a$ and $\mathtt{id}_b$ not in R, for a local store $\mathfrak{L}$ defined for any app identifier in R. The proof relies on the fact that apps within the same system interact only via the global store.

Let $(\mathfrak{C}_a, \mathfrak{C}_b) \in \mathcal{R}$ such that $\mathfrak{C}_a \xrightarrow{\alpha}_\triangleright \mathfrak{C}'_a$, for some configuration $\mathfrak{C}'_a$. There are three possibilities.

- The action α has been triggered by the app $\mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,]$, i.e., $\alpha \neq \mathtt{x}?v$, and propagated to the whole system via the semantic rule (ParRight). In that case we rely on $\mathfrak{C}_{ap} \approx_H^{\text{ti}} \mathfrak{C}_{bp}$ and the semantic rule (ParRight) to close up the bisimulation game without any involvement of the system R and the local store $\mathfrak{L}$, independently of whether $\alpha \in H$ or $\alpha \notin H$.
- The action α has been triggered by the system R, i.e., $\alpha \neq \mathtt{x}?v$, and propagated to the whole system via the semantic rule (ParLeft). This means $\mathfrak{C}_a = \langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,] \parallel \mathsf{R}) \xrightarrow{\alpha}_\triangleright \langle \mathfrak{G}'_a, \mathfrak{L}_a \uplus \mathfrak{L}' \rangle \triangleright (\mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,] \parallel \mathsf{R}') = \mathfrak{C}'_a$. We recall that $\mathfrak{C}_{ap} \approx_H^{\text{ti}} \mathfrak{C}_{bp}$. Now, there are three possibilities.
  - $\alpha = \mathtt{id} : \mathtt{x}!v$ and $\alpha \in H$. In this case, the transition $\mathfrak{C}_a \xrightarrow{\alpha}_\triangleright \mathfrak{C}'_a$ was triggered by the application of the rule (Update) in Table 1, with $\mathfrak{G}'_a = \mathfrak{G}_a[\mathtt{x} \mapsto v]$ and $\mathfrak{L}' = \mathfrak{L}$. By an application of rule (EnvChange) we have $\mathfrak{C}_{ap} \xrightarrow{\mathtt{x}?v} \mathfrak{C}'_{ap} = \langle \mathfrak{G}'_a, \mathfrak{L}_a \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,]$. Notice that since $\alpha \in H$ it follows that $\mathtt{x} \in H$ and $\mathtt{x}?v \in H$. As $\mathfrak{C}_{ap} \approx_H^{\text{ti}} \mathfrak{C}_{bp}$ and $\mathtt{x}?v \in H$, there is $\mathfrak{C}'_{bp} = \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,]$ such that $\mathfrak{C}_{bp} \Rightarrow_H \mathfrak{C}'_{bp}$ and $\mathfrak{C}'_{ap} \approx_H^{\text{ti}} \mathfrak{C}'_{bp}$. For $\mathfrak{C}_b = \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R})$, let $\mathfrak{G}_b(\mathtt{x}) = w$, for some value w. Let $\mathfrak{C}_b \to_H \langle \mathfrak{G}_b[\mathtt{x} \mapsto v], \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}')$ by an application of the rules (Update) and (ParRight) (we recall that $\alpha \in H$). Then, $\langle \mathfrak{G}_b[\mathtt{x} \mapsto v], \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}') \to_H \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}')$ by an application of the rule (EnvChange) with action $\mathtt{x}?w$ (note that $\mathtt{x} \in H$ entails $\mathtt{x}?w \in H$). Finally, from $\mathfrak{C}_{bp} \Rightarrow_H \mathfrak{C}'_{bp}$ and several applications of the rule (ParLeft) it follows that $\langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}') \Rightarrow_H \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,] \parallel \mathsf{R}') = \mathfrak{C}'_b$. Summarizing, we have $\mathfrak{C}_b \Rightarrow_H \mathfrak{C}'_b$, with $(\mathfrak{C}'_a, \mathfrak{C}'_b) \in \mathcal{R}$ because $\mathfrak{C}'_{ap} \approx_H^{\text{ti}} \mathfrak{C}'_{bp}$.
  - $\alpha = \mathtt{id} : \mathtt{x}!v$ and $\alpha \notin H$. Also in this case, the transition $\mathfrak{C}_a \xrightarrow{\alpha}_\triangleright \mathfrak{C}'_a$ was triggered by the application of the rule (Update) in Table 1, with $\mathfrak{G}'_a = \mathfrak{G}_a[\mathtt{x} \mapsto v]$ and $\mathfrak{L}' = \mathfrak{L}$. By an application of rule (EnvChange) we have $\mathfrak{C}_{ap} \xrightarrow{\mathtt{x}?v} \mathfrak{C}'_{ap} = \langle \mathfrak{G}'_a, \mathfrak{L}_a \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_a[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,]$. Notice that since $\alpha \notin H$ it follows that $\mathtt{x} \notin H$ and $\mathtt{x}?v \notin H$. As $\mathfrak{C}_{ap} \approx_H^{\text{ti}} \mathfrak{C}_{bp}$ and $\mathtt{x}?v \notin H$, there is $\mathfrak{C}'_{bp} = \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,]$ such that $\mathfrak{C}_{bp} \stackrel{\mathtt{x}?v}{\Longrightarrow}_H \mathfrak{C}'_{bp}$ and $\mathfrak{C}'_{ap} \approx_H^{\text{ti}} \mathfrak{C}'_{bp}$. More precisely, $\mathfrak{C}_{bp} \Rightarrow_H \langle \mathfrak{G}^1_b, \mathfrak{L}^1_b \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \xrightarrow{\mathtt{x}?v} \langle \mathfrak{G}^1_b[\mathtt{x} \mapsto v], \mathfrak{L}^1_b \uplus \mathfrak{L} \rangle \triangleright \mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \Rightarrow_H \mathfrak{C}'_{bp}$. As a consequence, $\mathfrak{C}_b \Rightarrow_H \langle \mathfrak{G}^1_b, \mathfrak{L}^1_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R})$, by several applications of the rule (ParLeft). Then, $\langle \mathfrak{G}^1_b, \mathfrak{L}^1_b \uplus \mathfrak{L} \rangle \triangleright (\mathtt{id}_b[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R}) \xrightarrow{\alpha} \langle \mathfrak{G}^1_b[\mathtt{x} \mapsto v], \mathfrak{L}^1_b \uplus$

$\mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R}')$ by an application of rules (Update) and (ParRight). Finally, $\langle \mathfrak{G}^1_b[\mathtt{x} \mapsto v], \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R}') \Rightarrow_H \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,] \parallel \mathsf{R}') = \mathfrak{C}'_b$ by several applications of rule (ParLeft). Summarizing, we have $\mathfrak{C}_b \overset{\alpha}{\Rightarrow}_H \mathfrak{C}'_b$, with $(\mathfrak{C}'_a, \mathfrak{C}'_b) \in \mathcal{R}$ because $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$.

- $\alpha = \tau$. In this case, the transition $\mathfrak{C}_a \xrightarrow{\tau} \mathfrak{C}'_a = \langle \mathfrak{G}'_a, \mathfrak{L}_a \uplus \mathfrak{L}'\rangle \triangleright (\mathtt{id_a}[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,] \parallel \mathsf{R}')$ was derived by an application of the rule (ParRight) of Table 2, with $\mathfrak{G}'_a = \mathfrak{G}_a$, because $\langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L}\rangle \triangleright \mathsf{R} \xrightarrow{\tau} \langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L}'\rangle \triangleright \mathsf{R}'$ was fired by an application of one of the rules of Table 1. For convenience, we call this rule $\mathtt{r}$. Now, we want to show that there is $\mathfrak{G}'_b$ such that $\langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L}\rangle \triangleright \mathsf{R} \Rightarrow_H \langle \mathfrak{G}'_b, \mathfrak{L}_b \uplus \mathfrak{L}'\rangle \triangleright \mathsf{R}'$ where the rule $\mathtt{r}$ was used to move to R'. By inspection on Table 1, the premises of any possible rule $\mathtt{r}$ used in the transition $\langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L}\rangle \triangleright \mathsf{R} \xrightarrow{\tau} \langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L}'\rangle \triangleright \mathsf{R}'$ require only comparisons/evaluations involving both the global store $\mathfrak{G}_a$ and the local store $\mathfrak{L}$ (as the action is triggered by R). As $\mathfrak{G}_a \equiv_{\Sigma,\sigma} \mathfrak{G}_b$, the two stores may only differ on high-level services. Thus, we can always fire high-level (and hence non-observable) actions of the form $\mathtt{x}_i?v_i$ to align the global store $\mathfrak{G}_b$ to $\mathfrak{G}_a$ on *all high-level services* $\mathtt{x}_1 \ldots \mathtt{x}_k$. In practice, $\langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L}\rangle \triangleright \mathsf{R} \xrightarrow{\mathtt{x}_1?v_1} \ldots \xrightarrow{\mathtt{x}_k?v_k} \langle \mathfrak{G}_a, \mathfrak{L}_b \uplus \mathfrak{L}\rangle \triangleright \mathsf{R} \xrightarrow{\tau} \langle \mathfrak{G}_a, \mathfrak{L}_b \uplus \mathfrak{L}'\rangle \triangleright \mathsf{R}' \xrightarrow{\mathtt{x}_1?w_1} \ldots \xrightarrow{\mathtt{x}_k?w_k} \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L}'\rangle \triangleright \mathsf{R}'$, where the τ-action is due to an application of exactly the same rule $\mathtt{r}$, mentioned before; the remaining k actions $\mathtt{x}_i?w_i$ serve to recover the initial global store $\mathfrak{G}_b$. As a consequence, by k applications of the rule (EnvChange), one application of the rule (ParRight), and again k applications of the rule (EnvChange), we have $\mathfrak{C}_b \Rightarrow_H \mathfrak{C}'_b = \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L}'\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}')$. Let $\mathfrak{C}'_{ap} \overset{\text{def}}{=} \langle \mathfrak{G}_a, \mathfrak{L}_a \uplus \mathfrak{L}'\rangle \triangleright \mathtt{id_a}[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,]$ and $\mathfrak{C}'_{bp} \overset{\text{def}}{=} \langle \mathfrak{G}_b, \mathfrak{L}_b \uplus \mathfrak{L}'\rangle \triangleright \mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,]$. As $\mathfrak{C}_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}_{bp}$, by an application of Lemma 7 it follows that $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$. Summarizing, we have $\mathfrak{C}_b \Rightarrow_H \mathfrak{C}'_b$, with $(\mathfrak{C}'_a, \mathfrak{C}'_b) \in \mathcal{R}$ because $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$.

- The action α is of the form $\mathtt{x}?v$. In this case, the transition $\mathfrak{C}_a \xrightarrow{\alpha} \mathfrak{C}'_a = \langle \mathfrak{G}'_a, \mathfrak{L}'_a \uplus \mathfrak{L}'\rangle \triangleright (\mathtt{id_a}[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,] \parallel \mathsf{R})$ was triggered by an application of the rule (EnvChange), with $\mathfrak{G}'_a = \mathfrak{G}_a[\mathtt{x} \mapsto v]$ and $\mathfrak{L}' = \mathfrak{L}$. By a different application of rule (EnvChange) we have $\mathfrak{C}_{ap} \xrightarrow{\mathtt{x}?v} \mathfrak{C}'_{ap} = \langle \mathfrak{G}'_a, \mathfrak{L}_a \uplus \mathfrak{L}\rangle \triangleright \mathtt{id_a}[\,\mathsf{D}_a \bowtie \mathsf{P}_a\,]$. Now, there are two possibilities:
  - $\alpha \in H$. As $\mathfrak{C}_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}_{bp}$, there is $\mathfrak{C}'_{bp} = \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L}\rangle \triangleright \mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,]$ such that $\mathfrak{C}_{bp} \Rightarrow_H \mathfrak{C}'_{bp}$ and $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$. As a consequence, by several applications of the rule (ParLeft) it follows that $\mathfrak{C}_b \Rightarrow_H \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}_b\,] \parallel \mathsf{R}) = \mathfrak{C}'_b$, with $(\mathfrak{C}'_a, \mathfrak{C}'_b) \in \mathcal{R}$ because $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$.
  - $\alpha \notin H$. As $\mathfrak{C}_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}_{bp}$, there is $\mathfrak{C}'_{bp} = \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L}\rangle \triangleright \mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,]$ such that $\mathfrak{C}_{bp} \overset{\mathtt{x}?v}{\Longrightarrow}_H \mathfrak{C}'_{bp}$ and $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$. More precisely, $\mathfrak{C}_{bp} \Rightarrow_H \langle \mathfrak{G}^1_b, \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright \mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \xrightarrow{\mathtt{x}?v} \langle \mathfrak{G}^1_b[\mathtt{x} \mapsto v], \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright \mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \Rightarrow_H \mathfrak{C}'_{bp}$. As a consequence, $\mathfrak{C}_b \Rightarrow_H \langle \mathfrak{G}^1_b, \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R})$, by several applications of the rule (ParLeft). Then, $\langle \mathfrak{G}^1_b, \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R}) \xrightarrow{\mathtt{x}?v} \langle \mathfrak{G}^1_b[\mathtt{x} \mapsto v], \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R})$ by an application of rule (EnvChange). Finally, $\langle \mathfrak{G}^1_b[\mathtt{x} \mapsto v], \mathfrak{L}^1_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}^1_b\,] \parallel \mathsf{R}) \Rightarrow_H \langle \mathfrak{G}'_b, \mathfrak{L}'_b \uplus \mathfrak{L}\rangle \triangleright (\mathtt{id_b}[\,\mathsf{D}_b \bowtie \mathsf{P}'_b\,] \parallel \mathsf{R}) = \mathfrak{C}'_b$ by several applications of rule (ParLeft). Summarizing, we have $\mathfrak{C}_b \overset{\alpha}{\Rightarrow}_H \mathfrak{C}'_b$, with $(\mathfrak{C}'_a, \mathfrak{C}'_b) \in \mathcal{R}$ because $\mathfrak{C}'_{ap} \approx^{\mathrm{ti}}_H \mathfrak{C}'_{bp}$.

□

Proof of Theorem 3. Without loss of generality we can assume $\mathsf{S} \stackrel{\text{def}}{=} \prod_{j=1}^{n} \mathtt{id_j}[\,\mathsf{D}_j \bowtie \mathsf{P}_j\,]$. By n different applications of Proposition 1, we derive $\mathfrak{C}_a^j \approx_H^{\text{ti}} \mathfrak{C}_b^j$, for any $j \in \{1, \ldots, n\}$, for $\mathfrak{C}_a^j = \langle \mathfrak{G}_a, \mathfrak{L}_\perp \rangle \triangleright \mathtt{id_j}[\,\mathsf{D}_j \bowtie \mathsf{P}_j\,]$ and $\mathfrak{C}_b^j = \langle \mathfrak{G}_b, \mathfrak{L}_\perp \rangle \triangleright \mathtt{id_j}[\,\mathsf{D}_j \bowtie \mathsf{P}_j\,]$. Then, by several applications of Lemma 8 and the transitivity of $\approx_H^{\text{ti}}$ it follows that $\langle \mathfrak{G}_a, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S} \approx_H^{\text{ti}} \langle \mathfrak{G}_b, \mathfrak{L}_\perp \rangle \triangleright \mathsf{S}$, as required. □

Proof of Theorem 4. The proof follows the same line of thought as the proof of Theorem 3. Specifically, we need to replace σ-equivalence over global stores with the relation $\equiv^{\mathsf{d}}_{\Sigma,\sigma}$ to accommodate declassification policies. Notice that the typing rule for declassification (Declassify) ensures that the declassification operator applies only to global services and downgrades the security level of an expression e (defined over global services) to a lower security level. As a result, we can extend Lemma 3 to show that if $\Gamma \vdash_D \mathsf{declassify}(\mathsf{e}, \rho) : \rho$ with $\rho \preccurlyeq \sigma$ and $\mathfrak{G}_1 \equiv^{\mathsf{d}}_{\Sigma,\sigma} \mathfrak{G}_2$, then $[\![e]\!](\mathfrak{G}_1, \lambda \mathsf{x} \,.\, \perp) = [\![e]\!](\mathfrak{G}_2, \lambda \mathsf{x} \,.\, \perp)$. This result allows to lift Lemma 3 to expressions that may contain declassification operators. We can apply this result in Lemma 5 for the case of rule (SetLocal) to prove that $\equiv^{\mathsf{d}}_{\Sigma,\sigma}$ is preserved by single-step transitions. This allows us to lift the proofs of the other lemmas and ultimately prove Theorem 4 along the same lines of the proof of Theorem 3.

□