Theoretical Computer Science 998 (2024) 114537

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Behavioral equivalences for AbU: Verifying security and safety in distributed IoT systems ☆

Michele Pasqua [a,*], Marino Miculan [b]

[a] University of Verona, Strada le Grazie 15, Verona - 37134, Verona, Italy
[b] University of Udine, Via delle Scienze 206, Udine - 33100, Udine, Italy

ARTICLE INFO

Keywords:
ECA rules
IoT programming
Distributed systems
Bisimulations
Formal methods
Autonomic computing
Verification

ABSTRACT

Attribute-based memory Updates (AbU in short) is an interaction mechanism recently introduced for adapting the *Event-Condition-Action* (ECA) programming paradigm to distributed reactive systems, such as autonomic and smart IoT device ensembles. In this model, an event (e.g., an input from a sensor, or a device state update) can trigger an ECA rule, whose execution can cause the state update of (possibly) many remote devices at once; the latter are selected "on the fly" by means of predicates over their state, without the need of a central coordinating entity.

However, the combination of different AbU systems may yield unexpected interactions, e.g., when a new device is added to an existing secure system, potentially hindering the security of the whole ensemble of devices. This can be critical in the IoT, where smart devices are more and more pervasive in our daily life.

In this paper, we consider the problem of ensuring *security* and *safety* requirements for AbU systems (and, in turn, for IoT devices). The first are a form of noninterference, as they correspond to avoid forbidden information flows (e.g., information flows violating confidentiality); while the second are a form of non-interaction, as they correspond to avoid unintended executions (e.g., leading to erroneous/unsafe states).

In order to formally model these requirements, we introduce suitable *behavioral equivalences* for AbU. These equivalences are generalizations of *hiding bisimilarity*, i.e., a kind of weak bisimilarity where we can compare systems up to actions at different levels of security. Leveraging these behavioral equivalences, we propose (syntactic) sufficient conditions guaranteeing the requirements and, then, effective algorithms for statically verifying such conditions.

1. Introduction

In the Event-Condition-Action (ECA) programming paradigm, the behavior of a system is defined by a set of rules of the form "**on** *Event* **if** *Condition* **do** *Action*" which means: when *Event* occurs, if *Condition* is verified then execute *Action*. Due to its reactive nature, this paradigm is well-suited for programming "smart" systems, such as in IoT scenarios [1,2]. ECA systems react to events (as inputs) from the environment by performing *internal* actions, that update the local state, and *external* actions, that influence the environment

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
E-mail addresses: michele.pasqua@univr.it (M. Pasqua), marino.miculan@uniud.it (M. Miculan).

https://doi.org/10.1016/j.tcs.2024.114537
Received 27 December 2022; Received in revised form 26 November 2023; Accepted 25 March 2024

Available online 28 March 2024

itself. Due to their inherently simple yet powerful programming paradigm, all main platforms in the field of Home/Automotive IoT (e.g., IFTTT, Samsung SmartThings, Microsoft PowerAutomate, Zapier Zaps, etc.) adopt ECA rules.

Nevertheless, in these platforms, devices cannot directly execute the ECA rules nor directly interact with each other; instead, ECA rules are stored and executed on a central coordinator node (often deployed in the cloud, and reachable via the Internet), which collects the inputs from the devices and delivers the actions to be performed. Despite its simplicity, this architecture suffers from many issues. First, it does not scale well, due to the strongly centralized underlying infrastructure. Secondly, the availability depends on the central node, which may be offline or just unreachable due to network problems. Third, the transmission of user's sensible information to remote, unknown, servers on possibly insecure channels raises privacy concerns. Finally, all this data transmission on the network introduces delays and increases energy consumption.

To mitigate these issues, the ECA paradigm has been recently extended with *Attribute-based memory Updates* (AbU) [3], a communication mechanism designed for reactive *and* distributed programming, that is derived from Attribute-based Communication (AbC) [4,5]. In this model, nodes (e.g., IoT devices) can directly communicate with each other and self-coordinate, in a truly decentralized setting, without the need of a central entity. Furthermore, ECA rules are deployed and executed directly on the nodes, thus computation moves from the *cloud* to the *edge*, akin *fog computing*. In particular, in AbU an event on a node can cause the update of the states of (possibly many) *remote* nodes, selected "on the fly" by means of ECA rule conditions. For instance, the following rule:

$$login \gg @(\overline{role} = \text{`logger'}) : \overline{log} \leftarrow \overline{log} \cdot time$$

means "when the (local) variable $login$ changes, on every node whose $role$ is 'logger' append my current (local) $time$ to the (remote) variable log". Therefore, AbU allows us to propagate effects to collections of nodes at once, abstracting from their identities (or even their existence). Hence, AbU seamlessly combines the flexibility of a decentralized, property-driven interaction mechanism (*à la* AbC) with the simplicity of ECA rules. Attribute-based Communication, as well as popular interaction mechanisms used in smart systems (e.g., channels, agents, pub/sub, broadcast/multicast, etc.) [4,5], can be encoded in AbU [3].

Nevertheless, the simplicity and expressiveness of the AbU programming model comes to a price: the combination of different ECA rules may yield unexpected or unsecure interactions. This may happen when a new node or component is added to an existing system, or when two systems independently designed and implemented are joined in the same environment. For instance, adding rules publishing content on social networks from a folder on a file server could inadvertently disclose sensitive pictures, e.g., taken from a security camera, if these pictures are saved on the same folder. As another example, adding a controller that opens the window when the temperature is too high inside a room whose heater is remotely controlled—leading to the possibility to open the window when no one is at home, clearing the way for burglary. Therefore, an important problem is how to prevent these unwanted interactions between ECA rules.

In this paper, we focus on two kinds of *security* and *safety* requirements. The first is a form of *noninterference* [6]: we aim at assessing if an AbU system will never exhibit any *information flow* violating a given *security policy*. The second is a form of *non-interaction*: we aim at assessing whether different nodes will not interact by acting on common resources in unexpected ways. This is a *safety* requirement, as we aim at avoiding unintended executions, possibly leading to erroneous or unsafe system states.

To formally model and reason about these requirements, we introduce suitable *behavioral equivalences* between AbU systems, following the approach of [2]. These equivalences, called *hiding bisimilarities*, are (weak) bisimulations hiding the observations that are not related to the requirements check (and that would trivially break the equivalence). However, we need to generalize the definition of [2] in order to deal with specific aspects of AbU. Indeed, an ECA rule in AbU may update at once resources at different levels of security; hence, we have to generalize hiding bisimilarity to compare observations involving different security levels at once. Leveraging these equivalences, we propose syntactic sufficient conditions and an algorithm to *statically* check noninterference and non-interaction of AbU systems.

Another aspect, peculiar of IoT scenarios, concerns the interaction with the physical environment. This can introduce *implicit* interactions between resources that appear unrelated from the programmer's point of view. For instance, we may have an interaction from the resource controlling a lamp to the resource reading the state of a light sensor; this interaction cannot be deducted from the analysis of the ECA rules alone. To deal with this issue, we extend our framework with a notion of *semantic dependency*, representing the implicit interactions given by the environment.

This paper is an extended version of [7,8]. With respect to this work, we generalized the bisimulations for security and safety presented in [7]. In particular, both bisimulations now consider also the inputs coming from the environment. Furthermore, the bisimulation for security and, hence, the noninterference definition, is given in terms of a generic security lattice, rather than the classic two-points (public/private) lattice as in [7]. The calculus of [8] has been extended with *invariants*, namely conditions that nodes have to fulfill during run-time. A new syntactic construct for specifying invariants has been added, together with the corresponding semantic rule. Security and safety requirements of [7] have been updated accordingly.

Furthermore, we added a notion of information declassification to the security requirement (i.e., to noninterference) presented in [7], adapting the proposed verification algorithm. The verification mechanisms for the security and safety requirements have been improved w.r.t. those presented in [7]. In particular, the algorithm for detecting information flows now computes also the system's *attack surface*, i.e., the set of resources which can be exploited to carry out an attack. Finally, we added a discussion about the compositionality of the verification for the security requirement (i.e., of noninterference) presented in [7] and new examples to showcase the generality of the calculus presented in [8].

Synopsis In Section 2 we provide a short introduction to AbU, an ECA-inspired calculus extended with Attribute-based memory Updates. Then, in Section 3 we define some behavioral equivalences for AbU systems, to model two requirements which are crucial

when designing secure and safe IoT systems; while in Section 4 we propose two verification mechanisms to statically check the previously defined requirements. In Section 5 we deal with the problem of implicit resources interactions and the controlled release of sensitive information (i.e., information flows declassification); together with a discussion about the compositionality of the security requirement verification. Related work is discussed Section 6, and finally, in Section 7 we draw some conclusions and give directions for future work. Full proofs can be found in Appendix A.

2. Attribute-based memory updates in short

In this section we recall AbU [3,9], a calculus merging the simplicity of ECA programming with a powerful distributed communication mechanism, i.e., *attribute-based memory updates*. The latter allows a node to update at once the states of many nodes, which are selected by means of their attributes. Moreover, in this paper we extend the calculus presented in [3] with the possibility of specifying *node invariants*, i.e., predicates on each node's state which must be always satisfied. This is useful to avoid erroneous or dangerous states, like forbidden trajectories in planning, deadlocks, inconsistent values, etc. These features are introduced without sacrificing coding simplicity: ECA rules are still used to program the devices. This programming model turns out to be well suited for IoT and smart devices, which can now interact and self-coordinate directly without any central controlling node. We will see some examples in Section 2.2.

2.1. AbU syntax and semantics

We present here an overview of the AbU calculus, the interested reader can find a more detailed description in [3,9]. An AbU system S is basically a list of *nodes* which execute in parallel:

$$\mathsf{S} ::= R, \iota\langle \Sigma, \Theta\rangle \mid \mathsf{S} \parallel \mathsf{S}$$

Each *node* $R, \iota\langle \Sigma, \Theta\rangle$ consists of: a set R of ECA rules; an invariant ι, namely a boolean expression that the node must satisfy at runtime; a *state* $\Sigma \in \mathbb{X} \to \mathbb{V}$, mapping resources $x \in \mathbb{X}$ to values $v \in \mathbb{V}$; an *execution pool* $\Theta \subseteq (\mathbb{X} \times \mathbb{V})^*$, that is a set $\Theta = \{\mathsf{upd}_1, \ldots, \mathsf{upd}_n\}$ of lists of pairs of the form $(x_1, v_1) \ldots (x_m, v_m)$. Each list, called an *update*, represents a simultaneous multiple update waiting to be applied to the state. In the following we will denote the set of updates as $\mathbb{U} = (\mathbb{X} \times \mathbb{V})^* = \bigcup_{i \in \mathbb{N}} (\mathbb{X} \times \mathbb{V})^i$.

The syntax of the rules is defined by the following grammar.

$$\begin{array}{ll}
\mathsf{rule} ::= \mathsf{evt} \gg \mathsf{act}, \mathsf{task} & \mathsf{cnd} ::= \varphi \mid @\varphi \\
\mathsf{evt} ::= x \mid \mathsf{evt}\ \mathsf{evt} & \varphi, \iota ::= \mathtt{ff} \mid \mathtt{tt} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varepsilon \bowtie \varepsilon \\
\mathsf{act} ::= \epsilon \mid x \leftarrow \varepsilon\ \mathsf{act} \mid \overline{x} \leftarrow \varepsilon\ \mathsf{act} & \varepsilon ::= v \mid x \mid \overline{x} \mid \varepsilon \otimes \varepsilon \\
\mathsf{task} ::= \mathsf{cnd} : \mathsf{act} & x \in \mathbb{X} \quad v \in \mathbb{V}
\end{array}$$

An ECA rule $\mathsf{evt} \gg \mathsf{act}, \mathsf{task}$ has a listening *event* evt, which is a list of resources: when one of the resources in evt is modified, the rule is fired, namely the *default* action act and task are evaluated. Evaluation does not change the resource states immediately; instead, it yields update operations which are added to the execution pools, and applied later on. An action is a list of assignments of value expressions to *local* x or *remote* $\overline{x}$ resources. A task consists in a condition cnd and an action act. A *condition* is a boolean expression, optionally prefixed with the modifier @: when @ is not present, the task is *local*; otherwise the task is *remote*. In local tasks, the condition is checked in the local node and, if it holds, the action is evaluated. For remote tasks, on every node where the condition holds, the action is evaluated. The evaluation of an action yields an update, which is added to the current node pool in the case of default actions and local tasks; and added to remote nodes pools in the case of remote tasks. In the following, in order to simplify the notation, when a rule has an empty default action we write $\mathsf{evt} \gg \mathsf{task}$ in place of $\mathsf{evt} \gg \epsilon, \mathsf{task}$. In the syntax for boolean expressions φ (and invariants ι) and value expressions ε we left implicit comparison operators, e.g., $\bowtie \in \{<, \leq, >, \geq, =, \neq\}$, and binary operations, e.g., $\otimes \in \{+, -, *, /\}$.

The (small-step) semantics of AbU is modeled as a labeled transition system $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_2$ whose labels α are given by the grammar:

$$\alpha ::= T \mid \mathsf{upd} \triangleright T \mid \mathsf{upd} \blacktriangleright T$$

Here, T is a finite list of tasks and upd is an update. We have slightly modified the labels with respect to [3] since, in order to define the security and safety requirements, we need to *observe* which resources are updated. A transition can modify the state and the execution pool of the nodes but, at the same time, each node does not have a global knowledge about the system. The semantic rules are in Fig. 1. Rule (Exec) executes an update picked from the pool; while rule (Input) models an external modification of some resources. The execution of an update, or the external change of resources, may trigger some rules of the nodes. Hence, after updating a node state, the node launches a *discovery phase*, for finding new updates to add to the local pool (or some pools of remote nodes), given by the activation of some rules.

$$(\textsc{Exec})\ \frac{\begin{array}{c}\mathsf{upd} \in \Theta \quad \mathsf{upd} = (x_1, v_1) \dots (x_k, v_k) \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad \Sigma' \vDash \iota \\ \Theta'' = \Theta \setminus \{\mathsf{upd}\} \quad X = \{x_i \mid i \in [1..k] \wedge \Sigma(x_i) \neq \Sigma'(x_i)\} \\ \Theta' = \Theta'' \cup \mathsf{DefUpds}(R, X, \Sigma') \cup \mathsf{LocalUpds}(R, X, \Sigma') \quad T = \mathsf{ExtTasks}(R, X, \Sigma')\end{array}}{R, \iota\langle\Sigma, \Theta\rangle \xrightarrow{\mathsf{upd} \triangleright T} R, \iota\langle\Sigma', \Theta'\rangle}$$

$$(\textsc{Exec-Fail})\ \frac{\mathsf{upd} \in \Theta \quad \mathsf{upd} = (x_1, v_1) \dots (x_k, v_k) \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad \Sigma' \nvDash \iota \quad \Theta' = \Theta \setminus \{\mathsf{upd}\}}{R, \iota\langle\Sigma, \Theta\rangle \xrightarrow{\mathsf{upd} \triangleright T} R, \iota\langle\Sigma, \Theta'\rangle}$$

$$(\textsc{Input})\ \frac{\begin{array}{c}v_1, \dots, v_k \in \mathbb{V} \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad X = \{x_1, \dots, x_k\} \\ \Theta' = \Theta \cup \mathsf{DefUpds}(R, X, \Sigma') \cup \mathsf{LocalUpds}(R, X, \Sigma') \quad T = \mathsf{ExtTasks}(R, X, \Sigma')\end{array}}{R, \iota\langle\Sigma, \Theta\rangle \xrightarrow{(x_1, v_1) \dots (x_k, v_k) \blacktriangleright T} R, \iota\langle\Sigma', \Theta'\rangle}$$

$$(\textsc{Disc})\ \frac{\Theta'' = \{[\![\mathsf{act}]\!]\Sigma \mid \exists i \in [1..n]\,.\,\mathsf{task}_i = \varphi : \mathsf{act} \wedge \Sigma \vDash \varphi\} \quad \Theta' = \Theta \cup \Theta''}{R, \iota\langle\Sigma, \Theta\rangle \xrightarrow{\mathsf{task}_1 \dots \mathsf{task}_n} R, \iota\langle\Sigma, \Theta'\rangle}$$

$$(\textsc{StepL})\ \frac{\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{T} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ {\scriptstyle \alpha \in \{\mathsf{upd} \triangleright T, \mathsf{upd} \blacktriangleright T\}} \qquad (\textsc{StepR})\ \frac{\mathsf{S}_1 \xrightarrow{T} \mathsf{S}_1' \quad \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_2'}{\mathsf{S}_1 \parallel \mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_1' \parallel \mathsf{S}_2'}\ {\scriptstyle \alpha \in \{\mathsf{upd} \triangleright T, \mathsf{upd} \blacktriangleright T\}}$$

Fig. 1. Semantics of AbU calculus with invariants.

The discovery phase is composed by two parts, the local and the external one. A node $R, \iota\langle\Sigma, \Theta\rangle$ performs a local discovery by means of the functions DefUpds and LocalUpds, that add to the local pool Θ all updates originated by the activation of some rules in R. The *default updates* are the updates originated from the default actions of active rules in R, namely:

$$\mathsf{DefUpds}(R, X, \Sigma) \triangleq \{[\![\mathsf{act}]\!]\Sigma \mid \exists \mathsf{evt} \Rightarrow \mathsf{act}, \mathsf{task} \in \mathsf{Active}(R, X)\}$$

where $\mathsf{Active}(R, X)$ is the set of rules in R that listen on resources in X and $[\![\mathsf{act}]\!]\Sigma$ is the evaluation of the action act in the state Σ. The latter, it returns an update: $[\![x_1 \leftarrow \varepsilon_1 \dots x_n \leftarrow \varepsilon_n]\!]\Sigma \triangleq (x_1, [\![\varepsilon_1]\!]\Sigma) \dots (x_n, [\![\varepsilon_n]\!]\Sigma)$, where the evaluation semantics for value expressions ε is standard. The *local updates* are the updates originated from the tasks of the active rules in R that act only locally (@ is not present in the tasks' condition) and that satisfy the task's condition, namely:

$$\mathsf{LocalUpds}(R, X, \Sigma) \triangleq \{[\![\mathsf{act}_2]\!]\Sigma \mid \exists \mathsf{evt} \Rightarrow \mathsf{act}_1, \varphi : \mathsf{act}_2 \in \mathsf{Active}(R, X)\,.\,\Sigma \vDash \varphi\}$$

The satisfiability relation is $\Sigma \vDash \varphi \triangleq [\![\varphi]\!]\Sigma = \mathrm{tt}$, where the evaluation semantics for boolean expressions φ is standard as well.

The external discovery concerns tasks that contain the modifier @, hence an external node is needed to evaluate the task's condition. When a node needs to evaluate a task involving external nodes, it partially evaluates the task (with its own state) and then it sends the partially evaluated task to all other nodes. The latter, receive the task and complete the evaluation, potentially adding updates to their pool. In particular, the partial evaluation of tasks works as follows. With $\{\!|\mathsf{task}|\!\}\Sigma$ we denote the task obtained from task with each occurrence of x in the task's condition and the right-hand sides of the assignments in task's action replaced with the value $\Sigma(x)$. After that, each instance of $\overline{x}$ in the task's action is replaced with x and the modifier @ is dropped. For instance, $\{\!|@(x \leq \overline{x}) : \overline{y} \leftarrow x + \overline{y}|\!\}[x \mapsto 1\ y \mapsto 0] = (1 \leq x) : y \leftarrow 1 + y$.

Finally, the *external tasks* are the tasks of active rules in R whose condition contains @ (i.e., tasks that require an external node to be evaluated), namely:

$$\mathsf{ExtTasks}(R, X, \Sigma) \triangleq \{\!|\mathsf{task}_1|\!\}\Sigma \dots \{\!|\mathsf{task}_n|\!\}\Sigma$$

where for each $i \in [1..n]$ there exists a rule $\mathsf{evt} \Rightarrow \mathsf{act}, \mathsf{task}_i \in \mathsf{Active}(R, X)$ such that $\mathsf{task}_i = @\varphi : \mathsf{act}$.

Such discovery phase is launched by emitting the labels $\mathsf{upd} \triangleright T$, produced by the rule (Exec), and $\mathsf{upd} \blacktriangleright T$, produced by the rule (Input). On the other side, when a node receives a list of tasks (executing the rule (Disc) with a label T) it evaluates them and it adds to its pool the actions generated by the tasks whose condition is satisfied. The rules (StepL) and (StepR), the latter needed to enforce symmetry, complete and synchronize (on all nodes in the system) a discovery phase originated by a state change of a node.

The semantics also checks the fulfillment of invariants. Indeed, the rule (Exec) is applied only when the state modified by the update still satisfies the invariant (i.e., $\Sigma' \vDash \iota$); otherwise, rule (Exec-Fail) is applied. In this case, the update that would lead to a "bad" state is discarded and removed from the pool.

For each node, we define *legal execution states* the states satisfying the given invariant ι, that is the set $\{\Sigma \mid \Sigma \vDash \iota\}$. We assume that an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \dots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$ starts its execution on legal states only, namely for all $i \in [1..n]$ we have that $\Sigma_i \vDash \iota_i$ at the beginning of the computation.

2.2. AbU in action: IoT and security examples

Drone swarm Consider an IoT scenario where a swarm of drones is in charge of taking specific measurements, randomly picked in a large uninhabited area. Each drone is equipped with a battery that periodically needs to be recharged by returning to a docking station. It may happen that a drone runs out of energy before returning to the charging spot. In this case, the low-battery drone asks

for help from its neighbors. If a drone has some energy to share and it is close enough to the requester, it will enter the "rescue mode". We can model this scenario in AbU as follows (without the energy transfer phase, due to space reasons).

Suppose we have four drones. For each drone we have an AbU node with a resource *battery*, indicating the battery level of the drone; a resource *position*, indicating where is located the drone; a resource *mode*, indicating in which operative state is the drone; and a resource *helpPos*, indicating the position of a drone that needs help. Formally, the AbU system modeling the drone-swarm scenario is $\mathsf{S}_1 \parallel \mathsf{S}_2 \parallel \mathsf{S}_3 \parallel \mathsf{S}_4$, where

$$\mathsf{S}_1 = R\langle \Sigma_1, \varnothing \rangle,\ \mathsf{S}_2 = R\langle \Sigma_2, \varnothing \rangle,\ \mathsf{S}_3 = R\langle \Sigma_3, \varnothing \rangle,\ \mathsf{S}_4 = R\langle \Sigma_4, \varnothing \rangle$$

and R contains, among the others, the following two AbU rules:

$$battery \gg @(battery < 5 \wedge \overline{battery} > 80) : \overline{helpPos} \leftarrow position \tag{1}$$

$$helpPos \gg (|position - helpPos| < 7.0) : mode \leftarrow \text{'rescue'} \tag{2}$$

Now suppose that the execution states of the drones are the following:

$$\Sigma_1 = [battery \mapsto 5 \quad position \mapsto 2.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 0.0]$$

$$\Sigma_2 = [battery \mapsto 81 \quad position \mapsto 15.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 0.0]$$

$$\Sigma_3 = [battery \mapsto 97 \quad position \mapsto 6.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 0.0]$$

$$\Sigma_4 = [battery \mapsto 65 \quad position \mapsto 8.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 0.0]$$

The rule (1) says that when the current drone battery level is low ($battery < 5$), then the current drone has to send to all ($@$) neighbors with some energy to share ($\overline{battery} > 80$) its position, performing a remote update ($\overline{helpPos} \leftarrow position$). Suppose that the battery level of the first drone decreases by 1. Then, the first node can fire the rule (1), since its battery level is low. It pre-evaluates the task condition, yielding $(4 < 5 \wedge battery > 80)$, which is sent to the other nodes, together with the pre-evaluation of the task action, i.e., $helpPos \leftarrow 2.0$. Formally, the rule (Input) is applied on S_1, namely $R\langle \Sigma_1, \varnothing \rangle \xrightarrow{(battery,4) \blacktriangleright T} R\langle \Sigma_1', \varnothing \rangle$, where $\Sigma_1' = [battery \mapsto 4 \quad position \mapsto 2.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 0.0]$ and $T = (4 < 5 \wedge battery > 80) : helpPos \leftarrow 2.0$. Among all receivers, only the second and the third nodes are interested in the communication, since they are the only drones with battery level greater than 80. So they both add to their pool the update $(helpPos, 2.0)$. Formally, the rule (Disc) is applied on S_2, S_3 and S_4, but only on S_2 and S_4 it has some effect, namely $R\langle \Sigma_2, \varnothing \rangle \xrightarrow{T} R\langle \Sigma_2, \Theta \rangle$ and $R\langle \Sigma_3, \varnothing \rangle \xrightarrow{T} R\langle \Sigma_3, \Theta \rangle$, where $\Theta = \{(helpPos, 2.0)\}$. Indeed, since S_4 is not interested in the communication, we have $R\langle \Sigma_4, \varnothing \rangle \xrightarrow{T} R\langle \Sigma_3, \varnothing \rangle$. This ends the discovery phase originated by the first node. Now, the second and the third nodes can apply an execution step, since their pools are not empty, possibly triggering the rule (2). The rule (2), is fired when a drone receives a help request (i.e., when its resource *helpPos* changes) and basically checks if the current drone position is close to the requester position ($|position - helpPos| < 7.0$). If it is the case, the current drone enters the rescue mode performing a local update ($mode \leftarrow$ 'rescue'). In the example, when the second and the third nodes execute the update $(helpPos, 2.0)$, the task of the rule (2) may be executed. For the second node this does not happen, since $|15.0 - 2.0| < 7.0$ is not true (the node is too far from the first node). Instead, $|6.0 - 2.0| < 7.0$ and the third node can execute the rule task, adding to its pool the update $(mode, \text{'rescue'})$. Formally, suppose that S_2 is chosen for execution, namely we have $R\langle \Sigma_2, \Theta \rangle \xrightarrow{(helpPos,2.0) \rhd \epsilon} R\langle \Sigma_2', \varnothing \rangle$, by applying the rule (Exec), and $\Sigma_2' = [battery \mapsto 81 \quad position \mapsto 15.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 2.0]$. In this case, no further rule is triggered by the executed update, hence there are no external tasks (denote with ϵ in the transition label) and there is nothing to add to the node's local pool. Instead, when S_3 performs the execution step, we have that $R\langle \Sigma_3, \Theta \rangle \xrightarrow{(helpPos,2.0) \rhd \epsilon} R\langle \Sigma_3', \Theta' \rangle$, by applying the rule (Exec), with $\Sigma_3' = [battery \mapsto 97 \quad position \mapsto 6.0 \quad mode \mapsto \text{'measure'} \quad helpPos \mapsto 2.0]$ and $\Theta' = \{(mode, \text{'rescue'})\}$. In this case, the execution of the update triggers the rule (2), that it adds an update to the node's local pool (but no external tasks are generated).

Smart HVAC system In this example, we provide an AbU implementation of a Heating, Ventilation and Air Conditioning (HVAC) system, that makes use of device invariants. In this scenario we have three devices connected through a network: the HVAC control system, a temperature sensor, and a humidity sensor. To distinguish the devices, a logical resource *node* is used, which takes the values 'system', 'tempSens' and 'humSens' on the HVAC control system, the temperature sensor and the humidity sensor, respectively. We model such scenario in AbU as follows. The execution state for the HVAC control system is:

$$\Sigma_s = [heating \mapsto \texttt{ff} \quad conditioning \mapsto \texttt{ff} \quad temperature \mapsto 0 \quad humidity \mapsto 0 \quad airButton \mapsto \texttt{ff} \quad node \mapsto \text{'system'}]$$

while its ECA rules R_s are:

$$temperature \gg (temperature < 18) : heating \leftarrow \texttt{tt} \tag{3}$$

$$temperature \gg (temperature > 27) : heating \leftarrow \texttt{ff} \tag{4}$$

$$\begin{aligned} &humidity\ temperature \gg \\ &\quad (2 + 0.5 * temperature < humidity \wedge 38 - temperature < humidity) : conditioning \leftarrow \texttt{tt} \end{aligned} \tag{5}$$

$$airButton \gg (airButton = \texttt{tt}) : conditioning \leftarrow \texttt{ff} \tag{6}$$

The HVAC control system activates heating and air conditioning according to the values of temperature and humidity, received by the sensors. In particular, when the temperature is lower than $18\,°C$ ($temperature < 18$) the rule (3) activates the heating ($heating \leftarrow \texttt{tt}$). Instead, when the temperature is greater than $27\,°C$ ($temperature > 27$), then the rule (4) deactivates the heating ($heating \leftarrow \texttt{ff}$). The air conditioning is turned on ($conditioning \leftarrow \texttt{tt}$), by means of the rule (5), when the humidity exceeds the upper bound of the Givoni's *comfort zone* [10].

Instead, the execution state for the temperature and the humidity sensors are:

$$\Sigma_t = [temperature \mapsto 19 \;\; node \mapsto \text{'tempSens'}]$$

$$\Sigma_h = [humidity \mapsto 40 \;\; node \mapsto \text{'humSens'}]$$

while their ECA rules are:

$$R_t \triangleq temperature \gg @(node = \text{'system'}) : \overline{temperature} \leftarrow temperature \tag{7}$$

$$R_h \triangleq humidity \gg @(node = \text{'system'}) : \overline{humidity} \leftarrow humidity \tag{8}$$

The rule (7) on the temperature sensor device is simply responsible of signaling changes to the resource *temperature* to the HVAC control system, by selecting all devices that have *node* equals to 'system'; while the rule (8) does the same for the resource *humidity* on the humidity sensor device.

The HVAC control system is also bestowed with a physical button for manually stopping the air conditioning. Indeed, the rule (6) stops the air conditioning ($conditioning \leftarrow \texttt{ff}$) when the button is pressed (*airButton* is $\texttt{tt}$). Finally, by means of the invariant

$$\iota_s = \neg(conditioning \wedge heating)$$

on the HVAC control system device we specify that no update can result in the activation of both heating and air conditioning simultaneously. The complete AbU system is:

$$R_s, \iota_s\langle\Sigma_s, \varnothing\rangle \parallel R_t, \texttt{tt}\langle\Sigma_t, \varnothing\rangle \parallel R_h, \texttt{tt}\langle\Sigma_h, \varnothing\rangle$$

Note that, the same problem can be modeled by means of a single device, embedding the two sensors and the control system. We can model this scenario in AbU with a single device comprising all resources introduced so far and transforming remote rules into local ones. This highlights the flexibility of AbU, that is able to model both distributed and centralized ensembles of devices.

Diffie-Hellman key exchange In this last example, we model a well-known security protocol in AbU, that is, the Diffie-Hellman key exchange [11]. In such protocol, two parties, Alice and Bob, aim at securely exchange on an untrusted channel a (symmetric) cryptographic key, to be used in subsequent (secured) communications. Very briefly, the two parties agree on a prime number P and a primitive root G modulo P (that may be public). At the beginning of the protocol, Alice and Bob secretly chose an integer number each, say p_A and p_B, and compute a partial key each, say $X_A = G^{p_A} \bmod P$ and $X_B = G^{p_B} \bmod P$. Then, they exchange the partial keys and secretly compute the final shared key $key = {X_B}^{p_A} \bmod P = {X_A}^{p_B} \bmod P$. In AbU, we define a system with two nodes $\mathsf{S} = R\langle\Sigma_A, \varnothing\rangle \parallel R\langle\Sigma_B, \varnothing\rangle$ with the following initial states:

$$\Sigma_A = [G \mapsto 3 \;\; P \mapsto 7 \;\; node \mapsto \text{'Alice'} \;\; party \mapsto \text{'Bob'} \;\; exchange \mapsto \texttt{tt} \;\; phase1 \mapsto \texttt{ff} \;\; phase2 \mapsto \texttt{ff}]$$

$$\Sigma_B = [G \mapsto 3 \;\; P \mapsto 7 \;\; node \mapsto \text{'Bob'} \;\; party \mapsto \text{'Alice'} \;\; exchange \mapsto \texttt{ff} \;\; phase1 \mapsto \texttt{ff} \;\; phase2 \mapsto \texttt{ff}]$$

Each state contains a resource *node*, indicating the name of the party it represents (Alice or Bob) and a resource *party*, indicating the name of the other party (Bob or Alice). The other resources relate to the Diffie-Hellman protocol, in particular, G is the primitive root modulo a prime number P, on which the two parties agree on. Furthermore, both states comprise the following uninitialized[1] resources: p, X, Y, key. The rules R for the key exchange protocols are the following:

$$p \gg (exchange \neq \texttt{tt}) : X \leftarrow (G^p \bmod P) \qquad \textit{newPrime}$$

$$p \gg (exchange = \texttt{tt}) : X \leftarrow (G^p \bmod P) \;\; phase1 \leftarrow \texttt{tt} \;\; exchange \leftarrow \texttt{ff} \qquad \textit{newPrimeWithExchange}$$

$$phase1 \gg @(phase1 = \texttt{tt} \wedge \overline{node} = party) : \overline{Y} \leftarrow X \;\; phase1 \leftarrow \texttt{ff} \;\; \overline{phase2} \leftarrow \texttt{tt} \qquad \textit{exchangePhase1}$$

$$phase2 \gg @(phase2 = \texttt{tt} \wedge \overline{node} = party) : \overline{Y} \leftarrow X \;\; phase2 \leftarrow \texttt{ff} \qquad \textit{exchangePhase2}$$

$$Y \gg (\texttt{tt}) : key \leftarrow (X^Y \bmod P) \qquad \textit{computeKey}$$

The first two rules are used when a new integer number is generated by the parties. We have two different rules here since one of the party should start the protocol, but not both (as happens also in the Diffie-Hellman protocol, a form of synchronization between the parties is needed). Hence, one party (the one having *exchange* equals to $\texttt{tt}$) generates its partial key and starts the protocol setting *phase*1 to $\texttt{tt}$, while the other one just generates its partial key without performing any further update (phases basically

[1] Uninitialized resources here means that we do not care about the initial value of such resources.

simulate session information). The third rule executes the first exchange of the partial key, that is saved in the resource Y of the other device, and sets *phase*2 to tt on the other device. The fourth rule executes the second exchange of the partial key, completing the protocol. When the resource Y in one of the party is changed, meaning that a partial key is received, the device can readily compute (in parallel) the shared key by means of the last ECA rule.

3. Behavioral equivalences for AbU systems

In this section, we provide a semantic characterization of security and safety requirements for AbU systems, based on the notion of bisimulation. The security requirement we aim to assess is a form of *noninterference* [6], adapted to AbU systems. In particular, given a security policy defining the allowed information flow between resources, we aim at assessing whether an AbU system is *secure*, namely if it does not exhibit forbidden information flows (for instance, a flow from a confidential resource to a public one). Concerning the safety requirement, we consider the following scenario, quite common in the IoT world. We have some nodes, equipped with some ECA rules, whose behavior is known and safe for the user, and we have another node, also safe for the user. Is the ensemble of all such nodes still *safe*? This is a sort of *non-interaction* check, namely, we check whether different nodes interact with each other by acting on common resources in a way not intended by the user (leading to possibly inconsistent states).

We define formally these requirements by means of suitable *behavioral equivalences* between AbU systems, following (and generalizing) the approach of [2]. Intuitively, we aim at defining two bisimulations that capture, semantically, the security and safety requirements. To do so, we need a particular (weak) bisimulation hiding the system labels that are not related to the requirements check, and that would trivially break the bisimulation.

In particular, a *hiding bisimulation* makes non-observable all labels from a given set of labels, hence called *hidden*. Differently from [2], where labels can be either hidden or *fully* observable, in our approach we can also specify labels that are *partially* observable. Here partially means that we can fix an abstraction on what we can observe about not hidden labels. In other words, partially observable labels can be mimicked in the bisimulation game by other labels which are observationally equivalent, fixed a given labels abstraction.

Formally, let $\mathcal{L}$ be the set of all AbU system labels and $h \in \mathcal{L} \to \mathcal{L} \cup \{\diamond\}$ a function. We denote with $\to_h$ the relation involving any possible hidden label, i.e., $\to_h \triangleq \bigcup\{ \xrightarrow{\alpha} \mid h(\alpha) = \diamond \}$, and with $\Rightarrow_h$ its transitive closure, i.e., $\Rightarrow_h \triangleq \to_h^*$. Then, $\overset{\alpha}{\Rightarrow}_h \triangleq \Rightarrow_h \xrightarrow{\alpha} \Rightarrow_h$ means that we can perform an arbitrary, possibly empty, sequence of hidden labels, but at least one observable α label must be present.

Definition 1 *(Hiding bisimulation).* Let $h \in \mathcal{L} \to \mathcal{L} \cup \{\diamond\}$ be a function. A symmetric relation $\mathcal{R}$ between AbU systems is a *hiding bisimulation* w.r.t. h, if and only if for all $\mathsf{S}_1\ \mathcal{R}\ \mathsf{S}_2$ and $\mathsf{S}_1 \xrightarrow{\alpha} \mathsf{S}_1'$ we have:

- if $h(\alpha) = \diamond$ then $\mathsf{S}_2 \Rightarrow_h \mathsf{S}_2'$, for some S_2, with $\mathsf{S}_1'\ \mathcal{R}\ \mathsf{S}_2'$
- if $h(\alpha) \neq \diamond$ then $\mathsf{S}_2 \overset{\beta}{\Rightarrow}_h \mathsf{S}_2'$, for some β and S_2, with $h(\alpha) = h(\beta)$ and $\mathsf{S}_1'\ \mathcal{R}\ \mathsf{S}_2'$

and dually for $\mathsf{S}_2 \xrightarrow{\alpha} \mathsf{S}_2'$. We say that two AbU systems S_1 and S_2 are *hiding bisimilar* with respect to h, written $\mathsf{S}_1 \approx_h \mathsf{S}_2$, if $\mathsf{S}_1\ \mathcal{R}\ \mathsf{S}_2$ for some hiding bisimulation $\mathcal{R}$, w.r.t. h.

3.1. A bisimulation for security

In real situations (e.g., IoT systems), resources may have different security clearance: e.g., a security camera should definitely not leak any information to a resource that publicly hosts pictures on Internet. In the following, we assume a *security policy* $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$, which associates each resource used by an AbU system with a security level $\ell \in \mathrm{SL}$, taken from a complete lattice $\langle \mathrm{SL}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$. The lattice consists of a set SL of security levels, an ordering relation $\sqsubseteq$, the join $\sqcup$ and meet $\sqcap$ operators, as well as a top security level $\top$ and a bottom security level $\bot$. For the sake of simplicity, in the following examples we will consider the standard two-points security lattice $\{\mathsf{L}, \mathsf{H}\}$, where the bottom is L, representing *public* data, and the top is H, representing *confidential* data. The goal is to achieve classic *noninterference* [6] results stating that an AbU system is interference-free w.r.t. a given security level ℓ if its resources with clearance ℓ or lower are not affected by changes occurring at its resources with clearance ℓ' or greater, for a security level ℓ' (strictly) greater than ℓ. So, information can securely flow from a resource x to a resource y if[2] $\mathcal{P}(x) \sqsubseteq \mathcal{P}(y)$.

A security policy $\mathcal{P}$ induces an equivalence relation between states. Given two states Σ_1 and Σ_2, we say that they are ℓ-equivalent if they agree on the values associated to all resources with security at most ℓ.

Definition 2 *(ℓ-equivalence).* Let $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ be a security policy and $\ell \in \mathrm{SL}$. We say that the AbU nodes states Σ_1 and Σ_2 are *ℓ-equivalent*, written $\Sigma_1 \equiv_\ell \Sigma_2$, if for each resource $x \in \mathbb{X}$ such that $\mathcal{P}(x) \sqsubseteq \ell$, it is $\Sigma_1(x) = \Sigma_2(x)$.

We can easily extend this notion to arbitrary sets of states yielding from an AbU system. Given an AbU system $\mathsf{S} = R_1, \imath_1\langle \Sigma_1, \Theta_1 \rangle \ldots R_n, \imath_n\langle \Sigma_n, \Theta_n \rangle$, its *state set* is $\overline{\Sigma} = \{\Sigma_1, \ldots, \Sigma_n\}$, comprising the states of all nodes in S (we implicitly assume a correspondence between the i^{th} node in S and its corresponding state Σ_i in $\overline{\Sigma}$). In this setting, the state set $\overline{\Sigma}$ is ℓ-equivalent to another

[2] The ordering $\sqsubseteq$ for the two-points lattice is trivially defined as: $\{(\mathsf{L},\mathsf{L}), (\mathsf{L},\mathsf{H}), (\mathsf{H},\mathsf{H})\}$.

state set $\overline{\Sigma}' = \{\Sigma'_1, \ldots, \Sigma'_n\}$ when for all $i \in [1..n]$ we have that $\Sigma_i \equiv_\ell \Sigma'_i$. In other words, two state sets are ℓ-equivalent when they are element-wise ℓ-equivalent. We abuse notation by using the symbol $\equiv_\ell$ for both ℓ-equivalence of states and state sets.

As discussed at the beginning of the section, the goal is to formalize a bisimulation-based notion of noninterference. Intuitively, the runtime behavior at the security level ℓ or below of an interference-free AbU system does not change when we vary *only* resources with security clearance greater than ℓ. Similarly to what has been done in [2], a notion of hiding bisimilarity can be used to hide (but not to suppress) labels involving changes affecting resources at security level ℓ or below. In particular, only the updates involving resources with clearance greater than ℓ must be *hidden*, while the updates involving resources with clearance ℓ or below must be *fully* observable. Finally, updates with mixed resources must be *partially* observable (we need to make observable assignments to resources at security level ℓ or below only). We use the hiding bisimulation of Definition 1, with a specific function h, to define noninterference for AbU sets of rule lists and, in turn, for AbU systems.

Given a security level ℓ, we can define a *projection function* $(\cdot)\downarrow_\ell : (\mathbb{X} \times \mathbb{V})^* \to (\mathbb{X} \times \mathbb{V})^*$ that given an update upd returns its projection $\mathsf{upd}\downarrow_\ell$ on assignments to resources at security level ℓ or below only:

$$(\epsilon)\downarrow_\ell = \epsilon \qquad ((x,v)\,\mathsf{upd})\downarrow_\ell = \begin{cases} (x,v)\,(\mathsf{upd})\downarrow_\ell & \text{if } \mathcal{P}(x) \sqsubseteq \ell \\ (\mathsf{upd})\downarrow_\ell & \text{otherwise.} \end{cases}$$

Now let h_ℓ be a function hiding discovery labels and execution or input labels involving *only* resources with clearance greater than ℓ, and projecting execution or input labels involving mixed resources on their assignments to resources at security level ℓ or below only. Formally:

$$h_\ell(\alpha) \triangleq \begin{cases} \diamond & \text{if } \alpha = T \\ \diamond & \text{if } \alpha = \mathsf{upd} \rhd T \text{ or } \alpha = \mathsf{upd} \blacktriangleright T \text{ and } (\mathsf{upd})\downarrow_\ell = \epsilon \\ (\mathsf{upd})\downarrow_\ell & \text{if } \alpha = \mathsf{upd} \rhd T \text{ or } \alpha = \mathsf{upd} \blacktriangleright T \text{ and } (\mathsf{upd})\downarrow_\ell \neq \epsilon \end{cases}$$

Indeed, when $(\mathsf{upd})\downarrow_\ell = \epsilon$ we have that all resources in upd have security level greater than ℓ, namely $\mathsf{upd} = (x_1, v_1) \ldots (x_k, v_k)$ and $\ell \sqsubset \bigsqcap_{i \in [1..k]} \mathcal{P}(x_i)$. Dually, when $(\mathsf{upd})\downarrow_\ell \neq \epsilon$ we have that at least one resource in upd has security level less or equal than ℓ, namely $\mathsf{upd} = (x_1, v_1) \ldots (x_k, v_k)$ and $\bigsqcap_{i \in [1..k]} \mathcal{P}(x_i) \sqsubseteq \ell$. Note that, when an update upd involves resources with clearance at most ℓ only, we have that $(\mathsf{upd})\downarrow_\ell = \mathsf{upd}$, hence the label is fully observable (no abstraction). We call h_ℓ the *hiding function for ℓ-noninterference*.

In the following definition, and in the rest of the paper, we make use of some auxiliary notions. Given an AbU system $\mathsf{S} = R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \ldots R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$, its *rule list set* is $\overline{R} = \{R_1, \ldots, R_n\}$, comprising the rule lists of all nodes in S, while its *invariants set* is $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$, comprising the invariants of all nodes in S (again, we implicitly assume a correspondence between the i-th node in S and its corresponding rule list R_i in $\overline{R}$ and its invariant ι_i in $\overline{\iota}$).

Given a rule list set $\overline{R} = \{R_1, \ldots, R_n\}$ and an invariants set $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$, we define $\mathsf{comp}(\overline{R}, \overline{\iota})$ as the set comprising all possible state sets compatible with $\overline{R}$ and $\overline{\iota}$. Compatibility here means that states are defined for all and only the resources present in the rules and such states are legal for rule invariants. Formally, a state set $\overline{\Sigma}$ is *compatible* with $\overline{R}$ and $\overline{\iota}$, i.e., $\overline{\Sigma} \in \mathsf{comp}(\overline{R}, \overline{\iota})$ when for all $i \in [1..n]$ we have:

$$\mathrm{dom}(\Sigma_i) = \mathrm{vars}(R_i) \text{ and } \Sigma_i \vDash \iota_i$$

We also need a "system initialization" function $\mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma})$ that takes a rule list set, invariants set and a (compatible) state set and returns an AbU system with all pools empty. Formally, given $\overline{R} = \{R_1, \ldots, R_n\}$, $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$ and $\overline{\Sigma} = \{\Sigma_1, \ldots, \Sigma_n\} \in \mathsf{comp}(\overline{R}, \overline{\iota})$, we define $\mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}) = R_1, \iota_1\langle\Sigma_1, \varnothing\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \varnothing\rangle$.

Definition 3 *(AbU ℓ-noninterference).* Let $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ be a security policy and $\ell \in \mathrm{SL}$. We say that the AbU rule list set $\overline{R} = \{R_1, \ldots, R_n\}$ and invariants set $\overline{\iota}$ are *ℓ-level interference-free*, written ℓ-$\mathsf{NI}(\overline{R}, \overline{\iota})$, whenever:

$$\forall \overline{\Sigma}, \overline{\Sigma}' \in \mathsf{comp}(\overline{R}, \overline{\iota}) \,.\, \overline{\Sigma} \equiv_\ell \overline{\Sigma}' \implies \mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}) \approx_{h_\ell} \mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}')$$

3.2. An IoT-centric version of noninterference

The mere initialization of H-level (e.g., confidential) resources might activate a rule, thus leaking information about the *occurrence/presence* of a confidential event. The noninterference of Definition 3 ignores such presence leaks, as it is commonly done in language-based security. This design choice is usually justified by the fact that it increases the permissiveness of the enforcement mechanisms, but it is not a realistic assumption in the IoT context.

Example 1. Consider the following AbU rule:

$$\mathit{motion} \gg (00:00 < \mathit{time} \wedge \mathit{time} < 06:00) : \mathit{light} \leftarrow \text{'on'}$$

where *motion* is confidential while *time* and *light* public (i.e., $\mathcal{P}(\mathit{motion}) = \mathsf{H}$ and $\mathcal{P}(\mathit{time}) = \mathcal{P}(\mathit{light}) = \mathsf{L}$). Basically, the rule turns on the lights when, during the night, some movements in a room are detected. According to Definition 3 (with $\ell = \mathsf{L}$) there is no harmful information flows. Nevertheless, observing the (public) resource *light* we can infer that the (confidential) resource *motion*

has been changed (i.e., a robber may infer that someone is in the room).

End Example

Note that Definition 3 does not trivially ignore rule triggers, when checking noninterference. Indeed, it is able to capture harmful flows generated by rules acting on confidential triggers, as we can see in the following example.

Example 2. Consider the following AbU rules:

$$GPS \gg (GPS\ -\ center > 5.0) : area \leftarrow \text{`exit'} \tag{9}$$

$$area \gg (\mathtt{tt}) : log \leftarrow log \cdot \text{`border crossed at:'} \cdot time \tag{10}$$

where $area$, GPS and $center$ are confidential while log and $time$ public (i.e., $\mathcal{P}(area) = \mathcal{P}(GPS) = \mathcal{P}(center) = \mathsf{H}$ and $\mathcal{P}(log) = \mathcal{P}(time) = \mathsf{L}$). Rule (9) checks when the node exits a specific area, while rule (10) logs when the area borders are crossed (exiting or entering the area). Here, we have an information flow from the (confidential) resource GPS to the (public) resource log, which is not allowed by the security requirement and, indeed, is captured by Definition 3 (with $\ell = \mathsf{L}$).

End Example

What we want to remark with Example 2 is that Definition 3 is not able to capture presence leaks originated by external changes (i.e., inputs), but it is still able to capture presence leaks due to internal resources modifications (i.e., updates execution). In order to capture information flows due to generic resource presence leaks, we need a stronger (i.e., more restrictive) requirement.

Given an AbU rule list set $\overline{R} = \{R_1, \ldots, R_n\}$, an AbU invariants set $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$ and a security level $\ell \in \mathrm{SL}$, we define the *ℓ-higher events set* $\mathsf{evset}^{\ell}(\overline{R}, \overline{\iota})$ of $(\overline{R}, \overline{\iota})$ as all the resources with clearance greater than ℓ in the events of all rules in $\overline{R}$. Then, the *ℓ-level twin* of $(\overline{R}, \overline{\iota})$ is the pair $(\overline{R_\ell}, \overline{\iota_\ell})$ of rule list and invariants set where all resources in $\mathsf{evset}^{\ell}(\overline{R}, \overline{\iota})$ are substituted in $\overline{R}$ and $\overline{\iota}$ with their *primed version*. In particular, each resource in the rule list and in the invariants set is syntactically substituted with a renamed version (we assume a denumerable set of resource identifiers, hence we can always assign to the resources to rename a 'fresh' identifier, not already present in the initial set of names). As an example, the L-level twin of $(\overline{R}, \overline{\iota})$, where $\overline{R}$ is given by rules (9) and (10) of Example 2 and $\overline{\iota} \triangleq \{area > 0\}$ is $(\overline{R_{\mathsf{L}}}, \overline{\iota_{\mathsf{L}}}) = (\{area' \gg (\mathtt{tt}) : log \leftarrow log \cdot \text{`border crossed at:'} \cdot time\ \ GPS' \gg (GPS' - center > 5.0) : area' \leftarrow \text{`exit'}\}, \{area' > 0\})$. Note that, the resource $center$ is not modified since it does not belong to the set $\mathsf{evset}^{\mathsf{L}}(\overline{R}, \overline{\iota}) = \{area, GPS\}$.

The ℓ-level twin will be used in the following definition of noninterference. We have taken inspiration from self-composition verification mechanisms [12], where a k-hypersafety [13] verification problem for a program is reduced to a safety verification problem on its k-product program [12]. Indeed, noninterference is a 2-bounded subset-closed hyperproperty [14], so we can, in principle, verify it on two copies of the program, where confidential variables are renamed. Consider the case where we rename the resources with clearance greater than ℓ that rules are listening on (i.e., rules triggers at security levels greater than ℓ), we take ℓ-equivalent execution states (as for standard noninterference), and we run the two copies of the AbU system (which differ syntactically only on rule triggers at security levels greater than ℓ). If we assume no information flows w.r.t. Definition 3, it is easy to see that a change in the behavior at security level ℓ or below of the two systems can only be due to presence leaks originated from rule triggers at security levels greater than ℓ. Consider the rule $motion \gg (00:00 < time \wedge time < 06:00) : light \leftarrow \text{`on'}$ of Example 1, that is secure w.r.t. Definition 3, and its $\mathsf{L}$-level twin $motion' \gg (00:00 < time \wedge time < 06:00) : light \leftarrow \text{`on'}$. If we execute the two rules in isolation, when $motion$ changes we have that in the first case $light$ will take the value 'on', while in the second case nothing happens. Hence, we can note a difference in the behavior of the two AbU system, that is due to a presence leak originated by the confidential resource $motion$.

Definition 4 *(AbU presence-sensitive ℓ-noninterference).* Let $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ be a security policy and $\ell \in \mathrm{SL}$. We say that the AbU rule list set $\overline{R} = \{R_1, \ldots, R_n\}$ and invariants set $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$ are *ℓ-level presence-sensitive interference-free*, written ℓ-PNI$(\overline{R}, \overline{\iota})$, whenever:

$$\forall \overline{\Sigma} \in \mathsf{comp}(\overline{R}, \overline{\iota})\, \forall \overline{\Sigma}' \in \mathsf{comp}(\overline{R_\ell}, \overline{\iota_\ell}) \,.\, \overline{\Sigma} \equiv_\ell \overline{\Sigma}' \implies \mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}) \approx_{h_\ell} \mathsf{sys}(\overline{R_\ell}, \overline{\iota_\ell}, \overline{\Sigma}')$$

Using the noninterference notion of Definition 4, the AbU rules of Example 1 are now considered not secure. Presence-sensitive noninterference is a stronger requirement than classic (presence-insensitive) noninterference. Indeed, Definition 4 implies Definition 3, meaning that all AbU systems that satisfy Definition 4 also satisfy Definition 3, but not vice versa (a counter-example is the rule in Example 1 that satisfies Definition 3 but not Definition 4). Intuitively, a system satisfies Definition 4 if it does not have neither information flows nor presence leaks, while a system satisfies Definition 3 if it does not have information flows, but it may have presence leaks. In this sense, the set of systems satisfying Definition 3 is (strictly) larger than the set of systems satisfying Definition 4, that is ℓ-PNI$(\overline{R}, \overline{\iota})$ implies ℓ-NI$(\overline{R}, \overline{\iota})$.

3.3. A bisimulation for safety

We provide now a semantic characterization of *safe* interaction between AbU systems, by which we mean that two systems do not exhibit unintended behaviors when deployed together. For instance, consider a node that opens the window when the room temperature exceeds a given threshold, and another node equipped with a rule that turns on the thermostat at home when the user leaves his work location. Both node can be considered safe, in isolation, but when deployed together they may interact with each

other, causing an (unexpected) opening of the window when the user is not at home (clearing a way for burglary). Another unsafe scenario is when two nodes interact by updating some common resource (of remote nodes) in a inconsistent manner, e.g., a valve that is opened by a node and closed by the other at the same time.

Following [2], we would like to say that an AbU system S does not interact with, or is *transparent* for, another system R if the behavior of R when running in parallel with S does not differ from its behavior when running in isolation. In particular, we would like to say that S is transparent for R if $\mathsf{S} \parallel \mathsf{R} \approx_h \mathsf{R}$ for some bisimilarity $\approx_h$ that hides the updates originated from S.

Let $\overline{R^{\mathsf{S}}}$ and $\overline{R^{\mathsf{R}}}$ be the rule list sets of S and R, respectively. We can use the hiding bisimulation of Definition 1 to formalize a semantic-based notion of rule list sets transparency (and, in turn, of the corresponding systems). Our intention is to hide only those updates originated from rules in $\overline{R^{\mathsf{S}}}$. Formally:

$$h_{\mathsf{S}}(\alpha) \triangleq \begin{cases} \diamond & \text{if } \alpha = T \\ \diamond & \text{if } \alpha = \mathsf{upd} \rhd T \wedge \mathsf{source}(\mathsf{upd}) = \overline{R^{\mathsf{S}}} \\ \alpha & \text{if } \alpha = \mathsf{upd} \blacktriangleright T \\ \alpha & \text{if } \alpha = \mathsf{upd} \rhd T \wedge \mathsf{source}(\mathsf{upd}) \neq \overline{R^{\mathsf{S}}} \end{cases}$$

Here, we assume to have a function source returning the rule list set that has generated a given update. A mechanism for retrieving such information can be easily obtained augmenting AbU nodes with unique identifiers and recording in the AbU system labels $\mathsf{upd} \rhd T$ the identifier of the node performing the update. Alternatively, we can augment each AbU node with a "group identifier", indicating that the node belongs to S or R, in place of the node identifier (this is quite useful in the IoT, where nodes are often anonymous). For the sake of readability, we do not modify the syntax and the semantics of the calculus. We call h_{S} the *hiding function for transparency*.

Definition 5 *(AbU transparency).* Let $\overline{R^{\mathsf{S}}}$ and $\overline{\imath^{\mathsf{S}}}$ be the rule list and invariant sets for the system S, and $\overline{R^{\mathsf{R}}}$ and $\overline{\imath^{\mathsf{R}}}$ be the rule list and invariant sets of the system R. We say that $\overline{R^{\mathsf{S}}}$ and $\overline{\imath^{\mathsf{S}}}$ are *transparent* for $\overline{R^{\mathsf{R}}}$ and $\overline{\imath^{\mathsf{R}}}$, written $(\overline{R^{\mathsf{S}}}, \overline{\imath^{\mathsf{S}}}) \dashv\circ (\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}})$, if for each $\overline{\Sigma} \in \mathsf{comp}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{S}}} \cup \overline{\imath^{\mathsf{R}}})$ we have that:

$$\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{S}}} \cup \overline{\imath^{\mathsf{R}}}, \overline{\Sigma}) \approx_{h_{\mathsf{S}}} \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}}, \overline{\Sigma})$$

When $(\overline{R^{\mathsf{S}}}, \overline{\imath^{\mathsf{S}}}) \dashv\circ (\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}})$ and $(\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}}) \dashv\circ (\overline{R^{\mathsf{S}}}, \overline{\imath^{\mathsf{S}}})$, the two rule list and invariant sets are said *independent*, written $(\overline{R^{\mathsf{S}}}, \overline{\imath^{\mathsf{S}}}) \circ\!\dashv\circ (\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}})$.

In other words, if $\overline{R^{\mathsf{S}}}$ and $\overline{\imath^{\mathsf{S}}}$ are transparent for $\overline{R^{\mathsf{R}}}$ and $\overline{\imath^{\mathsf{R}}}$, then a system with $\overline{R^{\mathsf{S}}}$ as rule list set and $\overline{\imath^{\mathsf{S}}}$ as invariant set does not affect in any way the behavior of a system with $\overline{R^{\mathsf{R}}}$ as rule list set and $\overline{\imath^{\mathsf{R}}}$ as invariant set.

Example 3. Consider an AbU node managing a security camera. It is equipped with an AbU rule $camera \gg (\mathtt{tt}) : \overline{cloud.private} \leftarrow camera$ that basically uploads an image to the "private" folder of a given cloud service, when the camera detects some movements. Then, we can have another node managing the cloud service: when a new picture in the folder "public" is uploaded, the node will post it on Instagram. This can be modeled with the rule $cloud.public \gg (\mathtt{tt}) : instagram.post \leftarrow cloud.public$, which is self-explanatory. Until now, everything is ok, the two nodes are safe, even if executed together. Indeed, Definition 5 is fulfilled: taking S as the system comprising the camera-node and R as the system comprising the cloud-node, we have that S and R are independent.

Things change if we consider a buggy version of the camera node $camera \gg (\mathtt{tt}) : \overline{cloud.public} \leftarrow camera$. In this case, the node uploads the sensitive image to the "public" folder, instead to the "private" folder. Now, we have an unintended interaction chain: when the camera collects a sensitive image, the latter is automatically posted on Instagram. This interaction is captured by Definition 5, indeed the system S is now not transparent for the system R.

End Example

3.4. On the compositionality of requirements

Independence (Definition 5) is crucial when we aim at verifying *dynamically* a given requirement. In fact, suppose to have an AbU system R, that we know to satisfy a given requirement (e.g., termination [3], noninterference, etc.). If we combine (at runtime) R with another AbU system S satisfying the same requirement, and we know that the added system is independent from R, then we automatically have that $\mathsf{S} \parallel \mathsf{R}$ is compliant with the requirement. In other words, with independent systems we can reason about the satisfaction of a given requirement in a compositional way.

Note that, for some kind of properties (e.g., termination [3]) independence is not strictly necessary: transparency is a sufficient condition for guaranteeing compositionality. Indeed, if we have that the systems S and R are both loop-free (which is a sufficient condition for termination), and S is transparent for R, then we can conclude that $\mathsf{S} \parallel \mathsf{R}$ is loop-free as well.

4. Checking security and safety of AbU systems

In this section, we provide verification mechanisms for effectively checking the safety and security requirements introduced in Section 3. They are static, in the sense that they do not require the execution of the AbU systems under test: the check is purely based on the inspection of systems rules and invariants.

Algorithm $\texttt{IFRules}(\ell, \mathsf{rule}_1 \dots \mathsf{rule}_n)$
1 **return** $\bigcup_{i\in[1..n]} \texttt{IFSingleRule}(\ell, \mathsf{rule}_i)$

Procedure $\texttt{IFSingleRule}(\ell, x_1 \dots x_n \gg \mathsf{act}_1, \varphi : \mathsf{act}_2)$
2 $evtLevel := \bigsqcup_{i\in[1..n]} \mathcal{P}(x_i)$
3 $assignLevel := \texttt{Assign}(\mathsf{act}_1) \sqcap \texttt{Assign}(\mathsf{act}_2)$
4 $presLeak := evtLevel \not\sqsubseteq \ell \wedge assignLevel \sqsubseteq \ell$
5 $ctx := \texttt{Const}(\varphi)$
6 **if** $\texttt{IFAct}(\ell, \mathsf{act}_1, \kappa) \vee \texttt{IFAct}(\ell, \mathsf{act}_2, ctx) \vee presLeak$ **then**
7 **return** $\{x_1, \dots, x_n\}$
else
8 **return** $\emptyset$
end

Procedure $\texttt{IFAct}(\ell, x_1 \leftarrow \varepsilon_1 \dots x_n \leftarrow \varepsilon_n, ctx)$
9 $flow := \textbf{false}$
10 **for** $i = 1$ **to** n **do**
11 $isConst := \texttt{Const}(\varepsilon_i) \neq \not\kappa \wedge ctx \neq \not\kappa$
12 **if** $\mathcal{P}(x_i) \sqsubseteq \ell \wedge isConst = \textbf{false}$ **then**
13 $flow := \textbf{true}$
end
end
14 **return** $flow$

Procedure $\texttt{Assign}(x_1 \leftarrow \varepsilon_1 \dots x_n \leftarrow \varepsilon_n)$
15 **return** $\bigsqcap_{i\in[1..n]} \mathcal{P}(x_i)$

Algorithm 1. Information flows detection algorithm for ℓ-noninterference.

4.1. Verifying security

In order to provide a syntactic sufficient condition for noninterference we define a verification method detecting potential harmful information flows, parametric in the security policy $\mathcal{P}$. The detection process for a list of AbU rules is depicted in Algorithm 1. Given a security level $\ell \in \mathrm{SL}$, the algorithm computes the set of event resources that trigger rules yielding information flows. Hence, if the algorithm returns a non-empty set then the system contains harmful information flows from resources with clearance greater than ℓ to resources with clearance ℓ or below in, at least, one rule (line **1**). In this case the whole list of rules does not satisfy ℓ-noninterference. The procedure at lines **2..7** of Algorithm 1 looks for information flows inside single rules, and it works as follows.

First, it checks potential presence leaks. Line **2** computes the security level of the rule event: if at least one resource in the event is greater than ℓ then $evtLevel$ is greater than ℓ, otherwise is ℓ or below. Line **3** checks if the default and the task actions contain assignments to resources with clearance ℓ or below, by means of the procedure at line **14**. The latter computes the minimal clearance of the resources in the left-hand side of the assignments. Then, in line **4**, we check if there is a potential presence leak: $evtLevel \not\sqsubseteq \ell \wedge assignLevel \sqsubseteq \ell$ means that the event contains a resource with clearance greater than ℓ (or not comparable with ℓ) and we have assignments to resources with clearance ℓ or below in the actions.

Second, it checks potential harmful information flows in the default and the task actions. Line **5** computes a *constancy analysis* on the task condition,[3] in order to capture *implicit* information flows. The function `Const` returns κ when all resources with clearance ℓ or below are constants; and $\not\kappa$ otherwise. Here, constancy means that no variety is conveyed from resources with clearance greater than ℓ (the only ones that may change in Definition 4) to resources with clearance ℓ or below (assumed to be initially constant in Definition 4). Technically, our constancy analysis detects the presence of resources with clearance greater than ℓ inside boolean φ and value ε expressions. Line **6** computes the information flows in the default and in the task actions. Implicit flows can only happen in the task action, so for the default action we compute *explicit* information flows only, calling the `IFAct` function with κ as context. Instead, for the task action, `IFAct` is called with the context computed by the constancy analysis on the task condition, in order to track implicit flows. The procedure returns a non-empty set when presence leaks are detected or when information flows are present in the rules actions.

Finally, the procedure at lines **9..14** computes the potential information flows of an action,[4] parametric on a given context. It is a loop inspecting all assignments of the action. The condition at line **11** performs the check. Only two cases lead to harmful information flows: a resource with clearance ℓ or below is assigned with a not constant expression (explicit flow); a resource with clearance ℓ or below is assigned inside a not constant context (implicit flow). Recall that, not constancy means that variety is conveyed from resources with clearance greater than ℓ to resources with clearance ℓ or below.

Algorithm 1 detects potentially harmful information flows when considering a single list of AbU rules, namely a single node. Nevertheless, it is easy to note that the algorithm does not take into account inter-nodes communication, hence the verification on a rule list set, i.e., a set of nodes, boils down to the verification on a single rules list comprising all rules in the set. Intuitively, if we have a forbidden information flow between two nodes, such information flow must be present internally in one of the nodes as well.

Proposition 1. *Consider a rule list set* $\{R_1, \dots, R_n\}$. *Let* R *be the list comprising all rules of all elements in* $\{R_1, \dots, R_n\}$. *Then we have:*

$$\bigcup_{i\in[1..n]} \texttt{IFRules}(R_i) = \emptyset \iff \texttt{IFRules}(R) = \emptyset$$

Theorem 2 *(Soundness for security). Let* $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ *be a security policy and* $\ell \in \mathrm{SL}$. *Consider an AbU rule list set* $\overline{R} = \{R_1, \dots, R_n\}$ *and invariants set* $\overline{\imath} = \{\imath_1, \dots, \imath_n\}$. *Let* R *be the list comprising all rules of all elements in* $\overline{R}$. *If* $\texttt{IFRules}(R) = \emptyset$ *then* ℓ-$\mathsf{PNI}(\overline{R}, \overline{\imath})$ *holds.*

Proof. The proof is quite complex and it requires some preliminary results; see Appendix A.1. □

[3] The modifier @ does not influence the analysis, hence we omit it in the algorithm.
[4] Remote updates $\overline{x} \leftarrow \varepsilon$ do not influence the analysis, we omit them in the algorithm.

Recall that presence-sensitive noninterference implies the classic presence-insensitive version of noninterference. Hence, we can extend the soundness result as follows.

Corollary 3 *(Soundness for security). Let* $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ *be a security policy and* $\ell \in \mathrm{SL}$. *Consider an AbU rule list set* $\overline{R} = \{R_1, \ldots, R_n\}$ *and invariants set* $\overline{\imath} = \{\imath_1, \ldots, \imath_n\}$. *Let* R *be the list comprising all rules of all elements in* $\overline{R}$. *If* $\mathtt{IFRules}(R) = \varnothing$ *then* ℓ-$\mathsf{NI}(\overline{R}, \overline{\imath})$ *holds.*

Example 4. Take the AbU rule of Example 1. We have that Algorithm 1 will correctly mark it as not secure, capturing a presence leak. Intuitively, the L-level resource *light* is assigned when an H-level event is present, due to the H-level resource *motion*. Algorithm 1 computes at line **2** the security level of the rule event $evtLevel = \mathsf{H}$, since $\mathcal{P}(motion) = \mathsf{H}$. The rule action assigns the public variable *light*, hence $assignLevel = \mathsf{L}$ at line **3**. Then, Algorithm 1 checks at line **4** whether the rule contains presence leaks or not. In this case, $presLeak = \mathbf{true}$, since $\mathsf{H} = evtLevel \not\sqsubseteq \ell = \mathsf{L}$ and $\mathsf{L} = assignLevel \sqsubseteq \ell = \mathsf{L}$ are both satisfied (indicating a presence leak). Line **5** computes the constancy analysis on the rule condition. In this case, $ctx = \kappa$ since no confidential resources are present in the rule condition. The procedure `IFAct` on the rule action returns **false**, since the action context is constant (computed at line **5**) and the action expression is constant (hence, at line **11**, $isConst = \mathbf{true}$). This means that no information flows are present in the rule action. Nevertheless, since $presLeak = \mathbf{true}$ we have a presence leak and, consequently, the condition at line **6** is satisfied. Hence, Algorithm 1 returns the set $\{motion\}$, indicating a violation of Definition 4. An analogous reasoning can be done for the AbU rules of Example 2.

Now consider the AbU rule $access \gg (user.role = \text{'guest'}) : log \leftarrow user.name \cdot time$ that logs the access time of users that have role 'guest' only. Suppose that the user role is confidential, while all other resources are public (i.e., $\mathcal{P}(access) = \mathcal{P}(user.name) = \mathcal{P}(time) = \mathcal{P}(log) = \mathsf{L}$ while $\mathcal{P}(user.role) = \mathsf{H}$). We have an implicit information flow here (H to L), from *user.role* to *log*. Indeed, Algorithm 1 will correctly mark it as not secure: we assign a L-level resource (*log*) inside an action with a not constant context, given by $\mathtt{Const}(user.role = \text{'guest'}) = \not\kappa$. Indeed, Algorithm 1 computes $evtLevel = assignLevel = presLeak = \mathsf{L}$ and $ctx = \not\kappa$, since the rule condition contains the confidential resource *user.role*. In this case, the procedure `IFAct` on the rule action returns **true**, since the action expression is constant but the action context is not constant (hence, at line **11**, $isConst = \mathbf{false}$). Due to the fact that the rule assigns a public resource in a not constant context, at line **13** we have $flow = \mathbf{true}$. Hence, Algorithm 1 returns the set $\{access\}$, indicating a violation of Definition 4.

End Example

4.2. Verifying safety

In order to provide a syntactic sufficient condition for transparency we have to individuate the resources that a system may potentially update (*sinks*) and the resources that may influence a rule behavior (*sources*). The first are the left-hand sides of assignments in rules actions, while the latter are the rules events. In addition to the events, also resources involved in tasks condition and resources used in the actions should be considered sources. Indeed, take the AbU rules $x \gg (x < 3) : z \leftarrow 4$ and $x \gg (\mathtt{tt}) : z \leftarrow w$. The resources y and w should be considered sources, since their modification by an external node influences the behavior of the rules (even if they are not triggers). More formally, let us define

$$\mathsf{LHS}(x_1 \leftarrow \varepsilon_1 \ldots x_n \leftarrow \varepsilon_n) \triangleq \{x_1, \ldots, x_n\} \qquad \mathsf{RHS}(x_1 \leftarrow \varepsilon_1 \ldots x_n \leftarrow \varepsilon_n) \triangleq \bigcup_{i \in [1..n]} \mathsf{Vars}(\varepsilon_i)$$

(which are defined analogously also when the action contains remote assignments $\overline{x} \leftarrow \varepsilon$). The sinks and sources of a rule are:

$$\mathsf{snk}(x_1 \ldots x_n \gg \mathsf{act}_1, \mathsf{cnd} : \mathsf{act}_2) \triangleq \mathsf{LHS}(\mathsf{act}_1) \cup \mathsf{LHS}(\mathsf{act}_2) \qquad \textit{sinks}$$

$$\mathsf{src}(x_1 \ldots x_n \gg \mathsf{act}_1, \mathsf{cnd} : \mathsf{act}_2) \triangleq \{x_1, \ldots, x_n\} \cup \mathsf{RHS}(\mathsf{act}_1) \cup \mathsf{RHS}(\mathsf{act}_2) \cup \mathsf{Vars}(\mathsf{cnd}) \qquad \textit{sources}$$

Given an AbU system with rule list set $\overline{R} = \{R_1, \ldots, R_n\}$, its *sinks* consist in all sinks of all rules in $\overline{R}$ while its *sources* consist in all sources of all rules in $\overline{R}$. Formally:

$$\mathsf{snk}(\overline{R}) \triangleq \bigcup_{1 \le i \le n} \mathsf{snk}(R_i), \quad \text{with } \mathsf{snk}(\mathsf{rule}_1 \ldots \mathsf{rule}_k) \triangleq \bigcup_{1 \le i \le k} \mathsf{snk}(\mathsf{rule}_i)$$

$$\mathsf{src}(\overline{R}) \triangleq \bigcup_{1 \le i \le n} \mathsf{src}(R_i), \quad \text{with } \mathsf{src}(\mathsf{rule}_1 \ldots \mathsf{rule}_k) \triangleq \bigcup_{1 \le i \le k} \mathsf{src}(\mathsf{rule}_i)$$

It is easy to note that when no sinks of S are sources of R, i.e., when $\mathsf{snk}(\overline{R^{\mathsf{R}}}) \cap \mathsf{src}(\overline{R^{\mathsf{S}}}) = \varnothing$, then S is transparent for R. This provides us with a sufficient syntactic condition for transparency, yielding the verification procedure described in Algorithm 2. In the nested loops at lines **3..6** we compute the sinks of the first system (S), by collecting all resources in the left-hand sides of all rules in the AbU system. Similarly, in the nested loops at lines **7..12** we compute the sources of the second system (R), by collecting all resources in events, right-hand sides and conditions of all rules in the AbU system. Finally, the value the algorithm returns depends on whether the sinks and the sources share some resources (line **13**). If this is the case, the algorithm returns **false**, meaning that the first system (S) is not transparent for the second one (R). Conversely, the algorithm returns **true**, meaning that the first system is transparent for the second one.

Thus, Algorithm 2 implements an easy-to-verify syntactic condition to check our semantic-based notion of safe interaction, formalized in Definition 5.

Algorithm `TransparencyCheck`$(\{R_1, \ldots, R_n\}, \{R'_1, \ldots, R'_m\})$

```
1  | sinks = ∅
2  | sources = ∅
3  | for i = 1 to n do
4  |   | for j = 1 to |R_i| do
5  |   |   | rule := the j^th rule of R_i
6  |   |   | sinks := sinks ∪ LeftHandSideVars(rule)
   |   | end
   | end
7  | for i = 1 to m do
8  |   | for j = 1 to |R'_i| do
9  |   |   | rule := the j^th rule of R'_i
10 |   |   | sources := sources ∪ EventVars(rule)
11 |   |   | sources := sources ∪ RightHandSideVars(rule)
12 |   |   | sources := sources ∪ ConditionVars(rule)
   |   | end
   | end
13 | return (sinks ∩ sources = ∅)
```

Algorithm 2. Transparency check algorithm for AbU systems.

Theorem 4 *(Soundness for safety). Let* $\overline{R^{\mathsf{S}}}$ *and* $\overline{\imath^{\mathsf{S}}}$ *be the rule list and invariant sets for the AbU system* S*, and* $\overline{R^{\mathsf{R}}}$ *and* $\overline{\imath^{\mathsf{R}}}$ *be the rule list and invariant sets of the AbU system* R*. If* $\texttt{TransparencyCheck}(\overline{R^{\mathsf{S}}}, \overline{R^{\mathsf{R}}}) = \textbf{true}$*, then* $(\overline{R^{\mathsf{S}}}, \overline{\imath^{\mathsf{S}}}) \dashv\circ (\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}})$.

Proof. The proof is quite complex and it requires some preliminary results. In order to simplify the reading, we moved the full proof to Appendix A.1. □

Example 5. Continuing Example 3, we have independence between (the first version of) the systems, since $\mathsf{snk}(\{camera \gg (\texttt{tt}) : \overline{cloud.private \leftarrow camera}\}) \cap \mathsf{src}(\{cloud.public \gg (\texttt{tt}) : instagram.post \leftarrow cloud.public\}) = \emptyset$ (and vice versa). Instead, in the case of the buggy version of the rules, we have that $\texttt{TransparencyCheck}(\{camera \gg (\texttt{tt}) : \overline{cloud.public \leftarrow camera}\}, \{cloud.public \gg (\texttt{tt}) : instagram.post \leftarrow cloud.public\}) = \textbf{false}$. Indeed, the set $\mathsf{snk}(\{camera \gg (\texttt{tt}) : \overline{cloud.public \leftarrow camera}\})$ and the set $\mathsf{src}(\{cloud.public \gg (\texttt{tt}) : instagram.post \leftarrow cloud.public\})$ have $\{cloud.public\}$ as intersection, capturing the unintended nodes interaction.

End Example

4.3. On the completeness of the verification mechanisms

The proposed verification mechanisms are sound, i.e., they do not expose false negatives, but they are *not complete*, i.e., they may expose false positives. Indeed, consider the two AbU rules:

$$l_1 \gg (h_1) : l_2 \leftarrow 3 \tag{11}$$

$$l_1 \gg (\neg h_1) : l_2 \leftarrow 3 \tag{12}$$

with $\mathcal{P}(l_1) = \mathcal{P}(l_2) = \mathsf{L}$ and $\mathcal{P}(h_1) = \mathsf{H}$. Algorithm 1 will flag as not secure an AbU system equipped with these rules, even if there is no interference (for both presence-sensitive and presence-insensitive versions). Indeed, independently from the value of h_1, we have that l_2 always takes the value 3. Another incompleteness witness consists in the following single AbU rule:

$$h_1 \gg (\texttt{tt}) : l_2 \leftarrow l_2 \tag{13}$$

which is rejected by our verification mechanism, even if it is actually secure. Indeed, the action $l_2 \leftarrow l_2$ does not change the value of any L-level resource (the update is idempotent).

Similarly, also Algorithm 2 rules out safe systems. For instance, consider a system with the rule list set $\{x \gg (\texttt{ff}) : y \leftarrow 3\}$, which is transparent for a system with rule list set $\{y \gg (\texttt{tt}) : z \leftarrow 2\}$, i.e., $\{x \gg (\texttt{ff}) : y \leftarrow 3\} \dashv\circ \{y \gg (\texttt{tt}) : z \leftarrow 2\}$. We have that $\mathsf{snk}(\{x \gg (\texttt{ff}) : y \leftarrow 3\}) \cap \mathsf{src}(\{y \gg (\texttt{tt}) : z \leftarrow 2\}) = \{y\}$, hence $\texttt{TransparencyCheck}(\{x \gg (\texttt{ff}) : y \leftarrow 3\}, \{y \gg (\texttt{tt}) : z \leftarrow 2\}) = \textbf{false}$.

Since to check the requirements defined in Section 3 is undecidable, every sound verification mechanism necessarily suffers from completeness issues, but we can in some cases improve precision to mitigate the problem. For instance, refining the procedure `IFAct` of Algorithm 1 by checking whether the left-hand side of an update is syntactically equal to right-hand side, we can spot the false positive resulting from rule (13). Indeed, in some cases we can remove false positives by applying simple heuristics that individuate syntactic patterns, as the one in rule (13). Instead, more complex cases, as the one represented by rules (11) and (12), require a more sophisticated analysis. For instance, in the case of rules (11) and (12) we need an inter-procedural version of the constancy analysis to rule out such false positive.

$$\text{(Exec)}\ \frac{\begin{array}{c}\mathsf{upd} \in \Theta \quad \mathsf{upd} = (x_1, v_1) \dots (x_k, v_k) \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad \Sigma' \vDash \iota \\ \Theta'' = \Theta \setminus \{\mathsf{upd}\} \quad X = \mathsf{clo}(\mathcal{K}, \{x_i \mid i \in [1..k] \wedge \Sigma(x_i) \neq \Sigma'(x_i)\}) \\ \Theta' = \Theta'' \cup \mathsf{DefUpds}(R, X, \Sigma') \cup \mathsf{LocalUpds}(R, X, \Sigma') \quad T = \mathsf{ExtTasks}(R, X, \Sigma')\end{array}}{R, \iota\langle\Sigma, \Theta\rangle \xrightarrow{\mathsf{upd} \triangleright T}_{\mathcal{K}} R, \iota\langle\Sigma', \Theta'\rangle}$$

$$\text{(Input)}\ \frac{\begin{array}{c}v_1, \dots, v_k \in \mathbb{V} \quad \Sigma' = \Sigma[v_1/x_1 \dots v_k/x_k] \quad X = \mathsf{clo}(\mathcal{K}, \{x_1, \dots, x_k\}) \\ \Theta' = \Theta \cup \mathsf{DefUpds}(R, X, \Sigma') \cup \mathsf{LocalUpds}(R, X, \Sigma') \quad T = \mathsf{ExtTasks}(R, X, \Sigma')\end{array}}{R, \iota\langle\Sigma, \Theta\rangle \xrightarrow{(x_1, v_1) \dots (x_k, v_k) \blacktriangleright T}_{\mathcal{K}} R, \iota\langle\Sigma', \Theta'\rangle}$$

Fig. 2. Modified AbU semantics rules to account for implicit interactions (remaining rules are as in Fig. 1).

5. Dealing with implicit interactions and declassification

In this section, we consider weakened forms of the requirements presented in Section 3, in order to better model application scenarios typical of the IoT.

5.1. Implicit interactions

We now study the challenge posed by *implicit interactions* that arises whenever two (physical) resources are semantically related, though this relation cannot be derived from the syntactic description of the system.

Example 6. Consider the rules $button \gg @(button = \text{'pressed'}) : \overline{robotCleaner} \leftarrow \text{'on'}$ and $motion \gg @(motion = \mathtt{tt} \wedge time < 12:00) : \overline{alarm} \leftarrow \text{'on'}$, deployed on different nodes. The first activates a robot cleaner in the house when a button on the phone is pressed. The second rings an alarm when some movement in the house is detected, during the morning. Though there are no (syntactic) interactions between the two rules, we clearly know that when the robot cleaner starts moving, then the motion sensor is activated and consequently the alarm will ring. We cannot catch this interaction with the LTS of Fig. 1, namely we would mark the nodes as independent.

End Example

We model these kind of *semantic dependencies* by means of a binary relation $\mathcal{K} \subseteq \mathbb{X} \times \mathbb{X}$ on resources such that $(x, y) \in \mathcal{K}$ when the resource y may be affected by changes occurring at the resource x (which is analogous to the *dependency policy* of [2]). Note that, this information is not syntactically modeled in the calculus; instead, it is an *exogenous*, "out of band" information, that the system developer should provide to rule out "semantic false negatives". Semantic dependencies can be composed, hence we will consider the reflexive and transitive closure of $\mathcal{K}$, in order to capture all possible dependencies associated to a resource. We write $\mathsf{clo}(\mathcal{K}, x)$ to denote the reflexive and transitive closure of the semantic dependencies relation $\mathcal{K}$ with respect to the resource x only. More generally, given a set of resources $X \subseteq \mathbb{X}$ we define $\mathsf{clo}(\mathcal{K}, X) \triangleq \bigcup_{x \in X} \mathsf{clo}(\mathcal{K}, x)$. In Example 6 we would have that $\mathcal{K} \triangleq \{(robotCleaner, motion)\}$, allowing us to capture the semantic dependence between the robot cleaner and the motion sensor.

As mentioned above, if $(x, y) \in \mathcal{K}$ it means that each time the resource x changes then the resource y can be somehow affected. We can include this abstract information in the discovery phase in the AbU semantics to all the resources affected by x. In other words, when we perform an execution or an input step in the semantics, we discover the actually modified resources *and* all the related resources, given by $\mathcal{K}$. Therefore, we can easily define a labeled transitions semantics $\rightarrow_{\mathcal{K}}$, parametric on $\mathcal{K}$, for which we just have to modify the rules (Exec) and (Input) of the original AbU semantics as depicted in Fig. 2.

Considering again Example 6, when the (Exec) rule performs the update $(robotCleaner, \text{'on'})$ then $\mathsf{clo}(\mathcal{K}, \{robotCleaner\}) = \{robotCleaner, motion\}$ and, hence, the rule concerning the motion sensor is selected by the discovery. Indeed, the nodes equipped with the rules in Example 6 now fail transparency, since in the bisimulation game the system without the cleaner cannot perform the update firing the alarm.

5.2. Information declassification

Noninterference is usually considered a too restrictive policy, to be effectively used in real-world applications. Indeed, sometimes a controlled release of sensitive information should deliberately allowed. The classic example is a password checking program which compares the password provided in input with the actual 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 has succeeded. Nevertheless, such 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 even if doing so we will technically go against the security policy.

We then extend the AbU calculus with a declassification primitive to support controlled release of sensitive information, in the spirit of *delimited release* [15]. Formally, we introduce a syntactic construct $(\cdot)_\ell$, with $\ell \in \mathrm{SL}$, into expressions syntax. Intuitively, $(\varepsilon)_\ell$ means that the expression ε, potentially containing data at security level greater than ℓ, can be declassified to the (lower) security level ℓ. Such construct is used for verification purposes only, hence it does not affect the AbU semantics (Fig. 1). Indeed, the evaluation $(\cdot)_\ell$ is equal to the evaluation of ε, namely, $[\![(\varepsilon)_\ell]\!]\Sigma = [\![\varepsilon]\!]\Sigma$. Furthermore, we forbid nested declassifications, namely we assume that in $(\varepsilon)_\ell$ the expression ε cannot contain other instances of the declassification construct.

In order to define noninterference up to declassification, we need a notion of state equivalence which accounts for the declassified expressions. Intuitively, we need to consider not only states with equivalent ℓ services but also states with equivalent declassified expressions. Given an AbU node, with rule list R, its declassification points are fixed and finite in number. For this reason, we define a *declassification strategy* δ as a list $(\varepsilon_1)_{\ell_1} \dots (\varepsilon_n)_{\ell_n}$ of declassification construct instances. Note that, in the two-points security lattice $\{\mathsf{L}, \mathsf{H}\}$, only H to L declassification is meaningful.

Definition 6 *(ℓ-equivalence up to declassification).* Let $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ be a security policy, $\ell \in \mathrm{SL}$ a security level and $\delta \triangleq (\varepsilon_1)_{\ell_1} \dots (\varepsilon_n)_{\ell_n}$ a declassification strategy. We say that the AbU nodes states Σ_1 and Σ_2 are *ℓ-equivalent up to declassification* δ, written $\Sigma_1 \equiv_\ell^\delta \Sigma_2$, if both the following hold:

- for each resource $x \in \mathbb{X}$ we have that $\mathcal{P}(x) \sqsubseteq \ell$ entails $\Sigma_1(x) = \Sigma_2(x)$ (ℓ-equivalence); and
- for each $i \in [1..n]$ we have that $\ell_i \sqsubseteq \ell$ entails $[\![\varepsilon_i]\!]\Sigma = [\![\varepsilon_i]\!]\Sigma'$.

Analogously to the case without declassification, we can extend this notion to arbitrary sets of states yielding from an AbU system: the state set $\overline{\Sigma} = \{\Sigma_1, \dots, \Sigma_n\}$ is ℓ-equivalent up to declassification δ to another state set $\overline{\Sigma}' = \{\Sigma'_1, \dots, \Sigma'_m\}$ when for all $i \in [1..n]$ we have that $\Sigma_i \equiv_\ell^\delta \Sigma'_i$. Again, we abuse notation by using the symbol $\equiv_\ell^\delta$ for both ℓ-equivalence up to declassification δ of states and state sets.

Nothing changes in the definition of hiding bisimulation, hence, we are ready to define (presence-sensitive) noninterference up to declassification, by reformulating Definition 4 using the declassified version of state equivalence. We can define the presence-insensitive version of noninterference up to declassification, by modifying in a similar way Definition 3.

Definition 7 *(AbU presence-sensitive ℓ-noninterference up to declassification).* Let $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ be a security policy, $\ell \in \mathrm{SL}$ a security level and $\delta \triangleq (\varepsilon_1)_{\ell_1} \dots (\varepsilon_n)_{\ell_n}$ a declassification strategy. We say that the AbU rule list set $\overline{R} = \{R_1, \dots, R_n\}$ and the invariants set $\overline{\imath}$ are *ℓ-level presence-sensitive interference-free up to declassification* δ, written ℓ-$\mathsf{PNI}(\overline{R}, \overline{\imath})_\delta$, whenever:

$$\forall \overline{\Sigma} \in \mathsf{comp}(\overline{R}, \overline{\imath})\, \forall \overline{\Sigma}' \in \mathsf{comp}(\overline{R_\ell}, \overline{\imath_\ell})\,.\, \overline{\Sigma} \equiv_\ell^\delta \overline{\Sigma}' \implies \mathsf{sys}(\overline{R}, \overline{\imath}, \overline{\Sigma}) \approx_{h_\ell} \mathsf{sys}(\overline{R_\ell}, \overline{\imath_\ell}, \overline{\Sigma}')$$

It is easy to note that Definition 4 implies Definition 7 and for AbU systems without the declassification constructs the two definitions coincide.

Verifying security up to declassification In order to verify noninterference up to declassification, we just need to slightly modify Algorithm 1. In particular, we have to modify the constancy analysis of expressions, adding the cases where the declassification construct appears. The analysis is parametric on the security level ℓ on which we check noninterference. If an expression is not declassified, the constancy analysis is defined as in Section 4, otherwise it is defined inductively on the structure of φ as:

$$\mathtt{Const}^\ell((\mathtt{ff})_{\ell'}) = \mathtt{Const}^\ell((\mathtt{tt})_{\ell'}) \triangleq \kappa$$

$$\mathtt{Const}^\ell((\neg\varphi)_{\ell'}) \triangleq \mathtt{Const}^\ell((\varphi)_{\ell'})$$

$$\mathtt{Const}^\ell((\varphi_1 \wedge \varphi_2)_{\ell'}) \triangleq \mathtt{Const}^\ell((\varphi_1)_{\ell'}) \Cup \mathtt{Const}^\ell((\varphi_2)_{\ell'})$$

$$\mathtt{Const}^\ell((\varphi_1 \vee \varphi_2)_{\ell'}) \triangleq \mathtt{Const}^\ell((\varphi_1)_{\ell'}) \Cup \mathtt{Const}^\ell((\varphi_2)_{\ell'})$$

$$\mathtt{Const}^\ell((\varepsilon_1 \bowtie \varepsilon_2)_{\ell'}) \triangleq \mathtt{Const}^\ell((\varepsilon_1)_{\ell'}) \Cup \mathtt{Const}^\ell((\varepsilon_2)_{\ell'})$$

Here, $\Cup$ is the join operator of the complete lattice $\{\kappa, \not\kappa\}$, with partial order $\Subset \triangleq \{(\kappa,\kappa), (\kappa,\not\kappa), (\not\kappa,\not\kappa)\}$. The constancy analysis for declassified value expressions is defined inductively on the structure of ε:

$$\mathtt{Const}^\ell((v)_{\ell'}) \triangleq \kappa$$

$$\mathtt{Const}^\ell((x)_{\ell'}) = \mathtt{Const}^\ell((\overline{x})_{\ell'}) \triangleq \begin{cases} \kappa & \text{if } (\ell' \sqsubset \mathcal{P}(x) \wedge \ell' \sqsubseteq \ell) \vee (\mathcal{P}(x) \sqsubseteq \ell \wedge \ell' \sqsubseteq \ell) \\ \not\kappa & \text{otherwise} \end{cases}$$

$$\mathtt{Const}^\ell((\varepsilon_1 \otimes \varepsilon_2)_{\ell'}) \triangleq \mathtt{Const}^\ell((\varepsilon_1)_{\ell'}) \Cup \mathtt{Const}^\ell((\varepsilon_2)_{\ell'})$$

Note that, since we cannot declassify rule events, the declassification does not affect in any way the detection of presence leaks.

5.3. Advantages of the AbU model

Compositionality of security In the IoT and, in general, in "smart" systems, it quite common to *reconfigure* a deployed system. For instance, new IoT devices may be added to an already running system, or some deployed devices may be substituted (e.g., in case of hardware or software updates). When this happens, it is claimed that the modified system still fulfills the requirements that the previous one satisfies. In other words, the newly added or updated nodes should not harm the already deployed ones. For instance, a secure system (w.r.t. Definition 4) should be still secure when we add a new device (i.e., the latter should not add forbidden

information flows). Usually, this requires to perform the security check on the whole new system, since in general noninterference is not compositional. But, due to its inherently simple control flow structure, in AbU noninterference *is compositional*. Indeed, if we have an information flow involving two ECA rules, then it must be that one of the two rule already exhibits an information flow.

Proposition 5. *Let* $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ *be a security policy and* $\ell \in \mathrm{SL}$. *Let* $\overline{R^{\mathsf{S}}}$ *and* $\overline{\imath^{\mathsf{S}}}$ *be the rule list and invariant sets for the AbU system* S, *and* $\overline{R^{\mathsf{R}}}$ *and* $\overline{\imath^{\mathsf{R}}}$ *be the rule list and invariant sets of the AbU system* R. *If* ℓ-PNI$(\overline{R^{\mathsf{S}}}, \overline{\imath^{\mathsf{S}}})$, *then we have:*

$$\ell\text{-PNI}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{S}}} \cup \overline{\imath^{\mathsf{R}}}) \iff \ell\text{-PNI}(\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}})$$

This tantamount to say that when we have a secure system, possibly composed by thousands of rules, and we want to add a new device, possibly composed by few rules, we do not need to analyze again the whole system, since the potential information flow violation can only be due to the added rules. This is a huge simplification in term of verification, that follows directly from AbU programming paradigm.

Increasing permissiveness Despite declassification, AbU provides another way of increasing the permissiveness of the security verification mechanism. Consider the case of an IoT system that is not secure, namely that exhibits forbidden information flows w.r.t. Definition 4. We can still deploy such system, paying attention to the resources that may yield harmful information flows. Indeed, Algorithm 1 provides the set of all events that may lead to insecure scenarios. We call this set the *attack surface* of an AbU system.

Precisely delimiting the attack surface has two main practical implications. On the one hand, to guarantee security and mitigate information leaks, we can simply (externally) monitor the resources that lie in the attack surface. On the other hand, if we add new rules to the system, and we check that those rules do not act on the resources in the attack surface, we are guaranteed to not leak information in the original system.

Proposition 6. *Let* S, R *be two AbU systems, and let* R^{S}, R^{R} *and* $R^{\mathsf{S}} \oplus R^{\mathsf{R}}$ *be the list comprising all rules of* S, *of* R *and of* S ∥ R, *respectively. If the sinks of* R *do not overlap with the attack surface of* S, *then:*

$$\texttt{IFRules}(R^{\mathsf{S}} \oplus R^{\mathsf{R}}) \setminus \texttt{IFRules}(R^{\mathsf{S}}) = \texttt{IFRules}(R^{\mathsf{R}})$$

In other words, if we add the rules of R to the rules of S, we do not introduce in S harmful information flows that were not already present in S.

6. Related work

Attribute-based interactions AbU [3] is inspired by the AbC calculus [16,5], from which it takes the idea of *attribute-based communication*. AbC is a core calculus of SCEL [17], a language introduced to model Collective Adaptive Systems (CAS) [18] and particularly suited for autonomic computing. Like SCEL, AbC adopts a message-based, procedural-oriented model. On the other hand, AbU aims to adapt attribute-based communication to fit the ECA programming model, which is data-oriented and rule-based, in a way transparent to the user. We refer to [3] for the comparison of AbU with related approaches.

Security and safety of ECA platforms for IoT Security and safety of IoT devices is a critical problem; among many works, we refer to recent surveys [19,20] which overview these risks in the IoT from a general point of view. Here, we recall the closely related work about security and safety of platforms based on ECA rules and about information-flow control for the IoT. For an overview on information-flow control in process algebra, we refer the reader to Focardi and Gorrieri [21].

The ECA paradigm is the standard for programming IoT devices, adopted by all major IoT platforms (like IFTTT, Samsung SmartThings, Microsoft PowerAutomate, etc.). In this context, IoT devices are managed by means of *apps* that users can downloaded (and customize) from the platform store. Recent studies point out the security and safety risks regarding this kind of apps, based of ECA rules. Surbatovich et al. [22] analyzed a dataset of 20K IFTTT apps, providing an empirical evaluation of potential secrecy and integrity violations, including violations due to cross-app interactions. Celik et al. [23,24] proposed some mechanisms to enforce (statically and dynamically) cross-app interaction vulnerabilities. Chi et al. [25] proposed a systematic categorization of threats arising from unintentional or malicious interaction of apps in IoT platforms. To detect cross-app interactions, they use symbolic execution techniques to analyze the apps code. Ding et al. [26] proposed a framework combining device physical channel analysis and static analysis to generate all potential interaction chains among IoT apps. They leverage Natural Language Processing to identify similarities between services, and proposed a risk-based approach to classify the actual risks of the discovered interaction chains. Nguyen et al. [27] designed IoTSan, a verification mechanism based on model-checking to reveal cross-app interaction flows. Similarly, SafeChain by Hsu et al. [28] leverage model checking techniques to identify cross-app vulnerabilities in IFTTT apps.

Another line of work focuses on enforcement mechanisms for checking security and safety of a *single* app, rather than an ensemble of apps. Fernandes et al. [29] presented FlowFence, an approach for building secure apps via information-flow tracking and controlled declassification. Celik et al. [30] leveraged static taint tracking to identify sensitive data leaks in an IoT app. Bastys et al. [31,32] identified new attack vectors in IFTTT apps and showed that 30% of apps from their dataset can be subject to such attacks. As a countermeasure, they investigated static and dynamic information-flow tracking via security types. Fernandes et al. [33] proposed the use of decentralization and fine-grained authentication tokens to limit privileges and prevent unauthorized actions inside an app.

Even if grounded by the same programming paradigm, i.e., based on ECA rules, all the above-mentioned work focuses on specific platforms, restricting the applicability to specific use cases. Instead, the requirements we propose in this work are built on top of AbU, thus providing a general setting in which security and safety can be verified interdependently from the application scenario.

Concerning more general ECA programming, [34,35] presented verification mechanisms to check properties (such as termination, confluence, redundant or contradicting rules) on IRON [36], a language based on ECA rules for the IoT domain. Other works proposed approaches to verify ECA programs by using Petri Nets [37] and BDD [38]. In [39,1], the authors presented a tool-supported method for verifying and controlling the correct interactions of ECA rules. All these works, differently from AbU, are not designed for distributed systems.

Information-flow control for the IoT Several works proposed information-flow control for enforcing confidentiality and integrity policies in the IoT domain. Newcomb et al. [40] proposed IOTA, a calculus for home automation. Based on the core formalism of IOTA, the authors developed an analysis for detecting whenever an event can trigger two conflicting actions, and an analysis for determining the action(s) that may influence a given event. Bodei et al. [41] proposed a calculus, IoT-LySa, supporting an information-flow analysis that safely approximates the abstract behavior of IoT systems. The calculus adopts asynchronous multi-party communication among nodes taking care of node proximity. Again, all the above-mentioned work focuses on specific platforms, while our approach based on AbU can be easily adapted to multiple application scenarios.

In their seminal work, Volpano and Smith [42] presented a *flow-insensitive* type system for imperative languages. Flow-insensitive type systems result very often too restrictive, rejecting lots of (practically) secure programs. To gain more permissiveness, Hunt and Sands [43] proposed a type system for an imperative language which is *flow-sensitive*. The latter has been further extended by Balliu et al. [44,2] in order to fit the IoT setting (in particular, apps based on ECA rules). The proposed type system verifies a notion of non-interference based on a suitable *hiding bisimulation* (a particularly designed weak bisimulation). We take inspiration from [2] to define the notion of security and safety requirements of this paper, generalizing the definition of hiding bisimulation.

7. Conclusion

In this paper, we have studied security and safety requirements of AbU systems, a new model for distributed computation merging the simplicity of ECA programming with attribute-based communication. AbU is particularly suited to program IoT devices, in a decentralized setting. Hence, these requirements can be used to tackle security and safety issues in the IoT. The first is a form of *noninterference*: we can assess if an AbU system does not exhibit forbidden information flows between resources, according to a given security policy. The second is a form of *non-interaction*: we can assess whether different nodes will not interact by acting on shared resources in unexpected ways.

To formally capture these requirements we have introduced two suitable bisimulations, generalizing the notion of *hiding bisimilarity* of [2], in order to deal with specific aspects of AbU systems. Leveraging these definitions, we have then given two sound verification mechanisms to *statically* check noninterference and non-interaction of AbU systems.

In addition, we considered a problem particularly common in the IoT, that is implicit interactions, i.e., not syntactically expressed interactions between resources, that may yield unsafe behaviors in IoT devices due to semantic correlations between resources, proposing a solution to mitigate such issue. Finally, we investigated the problem of intentional information release. Indeed, in some practical scenarios, noninterference is a too restrictive requirement, and a controlled release of confidential information is desirable. To this end, we added a *declassification mechanism* to downgrade the security level of expressions.

Future work Semantic dependencies are an out-of-band information that must be externally provided. Indeed, is not part of the AbU language and comes from external environmental factors (e.g., temperature can be influenced by walls insulation). Nevertheless, we can leverage Natural Language Processing techniques or machine learning in order to compute (i.e., infer) this information, starting from AbU rules. We plan to enhance our verification mechanisms with *heuristics for implicit interactions* as a future work.

As already mentioned at the end of Section 4, we plan to improve the precision of the information flows detection algorithm. In particular, we aim to develop an inter-procedural constancy analysis, leveraging model-checking techniques. In fact, modal and temporal logics are often used for reasoning about correctness of distributed systems, with both fully automatic tools and in interactive proof assistants [45–47].

Static analysis is sometimes too restrictive. So, we can move from static to *dynamic verification* (i.e., runtime monitoring), in order to detect violations at runtime of the security and safety requirements. This would enhance permissiveness at the expense of soundness. Then, the system developer would be in charge of choosing the strategy that best fits the particular application scenario.

Finally, we plan to develop other requirements, not strictly related to security and safety aspects. Indeed, *correctness requirements* are important as well in general as in the IoT context. An example is *rules confluence*: in some practical IoT scenarios it is important to ensure that rules execution order does not impact the overall system behavior. To this end, it can be useful to model AbU systems as (graph) rewriting systems, as it has been done for multi-agent systems in, e.g., [48].

CRediT authorship contribution statement

Michele Pasqua: Conceptualization, Formal analysis, Investigation, Methodology, Validation, Writing – original draft, Writing – review & editing. **Marino Miculan:** Conceptualization, Investigation, Methodology, Writing – original draft, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Proofs

A.1. Proofs of Section 4

We recall (and generalize) here the definition of the system initialization function, that takes a rule list set, an invariant set, a state set and a pool set, and it returns an AbU system, with the specified rules, invariants, states and pools. Formally, given $\overline{R} = \{R_1, \ldots, R_n\}$, $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$, $\overline{\Sigma} = \{\Sigma_1, \ldots, \Sigma_n\} \in \mathsf{comp}(\overline{R}, \overline{\iota})$ and $\overline{\Theta} = \{\Theta_1, \ldots, \Theta_n\}$, we define $\mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}, \overline{\Theta})$ as: $R_1, \iota_1\langle\Sigma_1, \Theta_1\rangle \parallel \ldots \parallel R_n, \iota_n\langle\Sigma_n, \Theta_n\rangle$. When all pools are empty we just write $\mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma})$ in place of $\mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}, \{\emptyset, \ldots, \emptyset\})$.

We also recall the notion of ℓ-level twin (given in Section 3) of a AbU rule list set $\overline{R} = \{R_1, \ldots, R_n\}$ and an invariants set $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$, i.e., the rule list set and invariants set pair $(\overline{R_\ell}, \overline{\iota_\ell})$ where all resources in $\mathsf{evset}^\ell(\overline{R}, \overline{\iota})$ are substituted in $\overline{R}$ and $\overline{\iota}$ with their primed version. Here, the ℓ-higher events set of $(\overline{R}, \overline{\iota})$ is $\mathsf{evset}^\ell(\overline{R}, \overline{\iota}) \triangleq \bigcup_{1 \le i \le n} \mathsf{evset}^\ell(R_i)$, with $\mathsf{evset}^\ell(\mathsf{rule}_1 \ldots \mathsf{rule}_m) \triangleq \bigcup_{1 \le j \le m} \mathsf{evset}^\ell(\mathsf{rule}_j)$ and $\mathsf{evset}^\ell(x_1 \ldots x_k \gg \mathsf{act}, \mathsf{task}) \triangleq \{x_i \mid i \in [1..k] \wedge \ell \sqsubset \mathcal{P}(x_i)\}$.

Proof of Theorem 2 We prove here the soundness of the proposed *security* verification mechanism, namely we prove that if Algorithm 1 marks an AbU system as secure then the system satisfies the (presence-sensitive) noninterference of Definition 4.

Before going in the detail of the proof, we need a preliminary result and an auxiliary definition. Indeed, it is easy to note that Algorithm 1 is not affected by resources renaming.

Proposition 7. *Consider the rule list set and invariants set pair $(\overline{R}, \overline{\iota})$ and its ℓ-level twin $(\overline{R_\ell}, \overline{\iota_\ell})$. Let R and R_ℓ be the list all rules in $\overline{R}$ and $\overline{R_\ell}$, respectively. We have that* $\texttt{IFRules}(R) = \emptyset \iff \texttt{IFRules}(R_\ell) = \emptyset$.

Indeed, if an AbU system does not contain harmful information flows then also the same system with all resources with security level greater than ℓ renamed does not contain harmful information flows. Proposition 7 just says that such correspondence also holds when considering Algorithm 1: if the procedure `IFRules` (of Algorithm 1) says that a system is secure (i.e., it does have harmful information flows) then such procedure would also say that the renamed system is secure (assuming 'fresh' names in the renaming).

Furthermore, we define an equivalence relation between rule list set and invariants set pairs, basically saying that a pair and its ℓ-level twin are equivalent. In particular, the pairs $(\overline{R}, \overline{\iota})$ and $(\overline{R'}, \overline{\iota'})$ are equivalent when $(\overline{R'}, \overline{\iota'})$ is the ℓ-level twin of $(\overline{R}, \overline{\iota})$ or $(\overline{R}, \overline{\iota})$ is the ℓ-level twin of $(\overline{R'}, \overline{\iota})$.

Definition 8 *(ℓ-level twin equivalence).* Given two AbU rule list set and invariants set pairs $(\overline{R}, \overline{\iota})$ and $(\overline{R'}, \overline{\iota'})$, we say that $(\overline{R}, \overline{\iota})$ and $(\overline{R'}, \overline{\iota'})$ are *ℓ-level twin equivalent*, written $(\overline{R}, \overline{\iota}) \overset{\ell}{\approx} (\overline{R'}, \overline{\iota'})$, when:

$$(\overline{R'}, \overline{\iota'}) = (\overline{R_\ell}, \overline{\iota_\ell}) \vee (\overline{R}, \overline{\iota}) = (\overline{R'_\ell}, \overline{\iota'_\ell})$$

Finally, we need an equivalence between execution pools, saying that two pools are equal except for updates containing renamed resources. We say that two updates are *primed equivalent* when they are identical or when they differ for primed resources only. For instance, $(l_1, 3)(l_2, 1)$ and $(l_1, 3)(l_2, 1)$ are primed equivalent, $(h_1, 3)(l_2, 1)$ and $(h'_1, 3)(l_2, 1)$ are primed equivalent, but $(l_1, 3)(l_2, 1)$ and $(h_1, 3)(l_2, 1)$ are not primed equivalent. Note that, primed equivalence of updates does consider order. For instance, $(l_1, 3)(l_2, 1)$ and $(l'_2, 1)(l'_1, 3)$ are not primed equivalent.

Definition 9 *(Primed equivalence).* Given two AbU execution pools Θ_1 and Θ_2, we say that Θ_1 and Θ_2 are *primed equivalent*, written $\Theta_1 \dot{\equiv} \Theta_2$, when: for each upd in Θ_1 there exists upd' in Θ_2 such that upd and upd' are primed equivalent; and for each upd in Θ_2 there exists upd' in Θ_1 such that upd and upd' are primed equivalent.

We can trivially extend the previous definition to pool sets, and we abuse notation by using the same symbol $\dot{\equiv}$ to denote primed equivalence for pools and pool sets.

Theorem 2 (Soundness for security) *Let $\mathcal{P} \in \mathbb{X} \to \mathrm{SL}$ be a security policy and $\ell \in \mathrm{SL}$. Consider an AbU rule list set $\overline{R} = \{R_1, \ldots, R_n\}$ and invariants set $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$. Let R be the list comprising all rules of all elements in $\overline{R}$. If* $\texttt{IFRules}(R) = \emptyset$ *then ℓ-*$\mathsf{NI}(\overline{R}, \overline{\iota})$ *holds.*

Proof. Let $\overline{R} = \{R_1, \ldots, R_n\}$ and $\overline{\iota} = \{\iota_1, \ldots, \iota_n\}$. Let R be the list comprising all rules of all elements in $\{R_1, \ldots, R_n\}$. Assume that $\texttt{IFRules}(R) = \emptyset$, then we have to prove that for all $\overline{\Sigma} \in \mathsf{comp}(\overline{R}, \overline{\iota})$ and $\overline{\Sigma}' \in \mathsf{comp}(\overline{R_\ell}, \overline{\iota_\ell})$ such that $\overline{\Sigma} \equiv_\ell \overline{\Sigma}'$, we have that $\mathsf{sys}(\overline{R}, \overline{\iota}, \overline{\Sigma}) \approx_{h_\ell} \mathsf{sys}(\overline{R_\ell}, \overline{\iota_\ell}, \overline{\Sigma}')$, where h_ℓ maps labels of the form T and of the forms $(x_1, v_1) \ldots (x_k, v_k) \triangleright T$ and $(x_1, v_1) \ldots (x_k, v_k) \blacktriangleright T$,

with $\ell \sqsubset \bigsqcap_{i\in[1..n]} \mathcal{P}(x_i)$, to $\diamond$; and maps labels of the forms $(x_1,v_1)\dots(x_k,v_k)\rhd T$ and $(x_1,v_1)\dots(x_k,v_k)\blacktriangleright T$, with $\bigsqcap_{i\in[1..n]} \mathcal{P}(x_i) \sqsubseteq \ell$, to $(x_1,v_1)\dots(x_k,v_k)\restriction_\ell$. Let $\mathcal{R}$ be the following binary and symmetric relation over AbU systems:

$$\mathcal{R} \triangleq \left\{ (\mathsf{sys}(\overline{R_1},\overline{\iota_1},\overline{\Sigma_1},\overline{\Theta_1}),\mathsf{sys}(\overline{R_2},\overline{\iota_2},\overline{\Sigma_2},\overline{\Theta_2})) \;\middle|\; \begin{array}{l} \overline{R_1} \overset{\ell}{\approx} \overline{R_2} \wedge \overline{\Sigma_1} \in \mathsf{comp}(\overline{R_1},\overline{\iota_1}) \wedge \\ \overline{\Sigma_2} \in \mathsf{comp}(\overline{R_2},\overline{\iota_2}) \wedge \overline{\Sigma_1} \equiv_\ell \overline{\Sigma_2} \wedge \\ \mathtt{IFRules}(R_1) = \varnothing \wedge \\ \mathtt{IFRules}(R_2) = \varnothing \wedge \\ \overline{\Theta_1} \dot{\equiv} \overline{\Theta_2} \end{array} \right\}$$

where R_1 and R_2 are the lists comprising all rules of all elements in $\overline{R}_1$ and $\overline{R}_2$, respectively. Note that, given two generic rule list sets $\overline{R}$ and $\overline{R'}$, we have that $\overline{R} \overset{\ell}{\approx} \overline{R'}$ implies $\mathtt{IFRules}(R) = \varnothing$ iff $\mathtt{IFRules}(R') = \varnothing$ (due to Proposition 1 and Proposition 7). By definition, $(\mathsf{sys}(\overline{R},\overline{\iota},\overline{\Sigma}),\mathsf{sys}(\overline{R_\ell},\overline{\iota_\ell},\overline{\Sigma'})) \in \mathcal{R}$, so we have to prove that $\mathcal{R}$ is an AbU hiding bisimulation, parametric on h_ℓ.

Let $(\mathsf{S}_a,\mathsf{S}_b) \in \mathcal{R}$ and $\mathsf{S}_a \xrightarrow{\alpha} \mathsf{S}'_a$, for some AbU system label α. We have to show that there exists a system S'_b such that $\mathsf{S}_b \overset{\alpha}{\Rightarrow}_{h_\ell} \mathsf{S}'_b$, with $(\mathsf{S}'_a,\mathsf{S}'_b) \in \mathcal{R}$. Note that, since AbU rules do not change their syntax during execution, we have that the rule list set of S_a, say $\overline{R_a}$, is the same of the rule list set of S'_a, say $\overline{R'_a}$. This implies that $\mathtt{IFRules}(R'_a) = \mathtt{IFRules}(R_a)$, where R'_a is the list comprising all rules of all elements in $\overline{R_a}'$ and R_a is the list comprising all rules of all elements in $\overline{R_a}$ (the same applies for S_b and S'_b). The fact that $\overline{R'_a} \overset{\ell}{\approx} \overline{R'_b}$ follows immediately (the ℓ-level twin equivalence $\overset{\ell}{\approx}$ is defined in terms of rule lists and invariants sets syntax only, but rules syntax does not change during execution). In a similar way, if the state set of S_a, say $\overline{\Sigma_a}$, is compatible with $\overline{R_a}$ and $\overline{\iota_a}$, then also the state set of S'_a, say $\overline{\Sigma'_a}$, is compatible with $\overline{R'_a}$ and $\overline{\iota'_a}$, where $\overline{\iota'_a}$ and $\overline{\iota'_b}$ are the invariants sets of S'_a and S'_b, respectively. (the same applies for S_b and S'_b). Hence, we just have to prove that there exists S'_b such that $\mathsf{S}_b \overset{\alpha}{\Rightarrow}_{h_\ell} \mathsf{S}'_b$, $\overline{\Sigma'_a} \equiv_\ell \overline{\Sigma'_b}$ and $\overline{\Theta'_a} \dot{\equiv} \overline{\Theta'_b}$, where $\overline{\Theta'_a}$ and $\overline{\Theta'_b}$ are the pool sets of S'_a and S'_b, respectively. The proof proceeds by case analysis on the label α.

Case $\alpha = T$. By definition of the AbU semantics (Fig. 1), the label T can only be generated by a system composed by a single node, by applying the rule (Disc). That is, $\mathsf{S}_a = R, \iota\langle\Sigma_a,\Theta_a\rangle$, for some node $R, \iota\langle\Sigma_a,\Theta_a\rangle$, and $\mathsf{S}_a \xrightarrow{\alpha} \mathsf{S}'_a = R, \iota\langle\Sigma_a,\Theta'_a\rangle$, for some pool Θ'_a. Since $(\mathsf{S}_a,\mathsf{S}_b) \in \mathcal{R}$, we have by hypothesis that S_b is the ℓ-level twin of S_a. This means that also S_b is a single node system and, in particular, $\mathsf{S}_b = R_\ell, \iota_\ell\langle\Sigma_b,\Theta_b\rangle$, for some state Σ_b and pool Θ_b. By definition of the AbU semantics (Fig. 1), we have that a single node can always perform any label T', by applying the rule (Disc). Hence, S_b can indeed perform the label T: $R_\ell, \iota_\ell\langle\Sigma_b,\Theta_b\rangle \xrightarrow{T} R_\ell, \iota_\ell\langle\Sigma_b,\Theta'_b\rangle$, for some Θ'_b. Let $T = \mathsf{task}_1 \dots \mathsf{task}_n$, for some tasks $\mathsf{task}_1,\dots,\mathsf{task}_n$. Since discovery rules do not modify node states and rules syntax does not change during execution, all conditions of the bisimulation concerning node rules, invariants and states trivially hold. What is left to prove is that $\Theta'_a \dot{\equiv} \Theta'_b$. By definition of the rule (Disc), we have that $\Theta'_a = \Theta_a \cup \{[\![\mathsf{act}]\!]\Sigma_a \mid \exists i \in [1..n]\,.\,\mathsf{task}_i = \varphi : \mathsf{act} \wedge \Sigma_a \vDash \varphi\}$ and $\Theta'_b = \Theta_b \cup \{[\![\mathsf{act}]\!]\Sigma_b \mid \exists i \in [1..n]\,.\,\mathsf{task}_i = \varphi : \mathsf{act} \wedge \Sigma_b \vDash \varphi\}$. Since $\Theta_a \dot{\equiv} \Theta_b$ by hypothesis, we just have to prove that $\tilde{\Theta}_a = \{[\![\mathsf{act}]\!]\Sigma_a \mid \exists i \in [1..n]\,.\,\mathsf{task}_i = \varphi : \mathsf{act} \wedge \Sigma_a \vDash \varphi\} \dot{\equiv} \{[\![\mathsf{act}]\!]\Sigma_b \mid \exists i \in [1..n]\,.\,\mathsf{task}_i = \varphi : \mathsf{act} \wedge \Sigma_b \vDash \varphi\} = \tilde{\Theta}_b$. Let $\tilde{\Theta}_a = \{\mathsf{upd}_1,\dots,\mathsf{upd}_n\}$ and $\tilde{\Theta}_b = \{\mathsf{upd}'_1,\dots,\mathsf{upd}'_m\}$, with $m \le n$. We have to prove that for each $i \in [1..n]$ there exists $j \in [1..m]$ such that upd_i is primed equivalent to upd'_j. Take an arbitrary $i \in [1..n]$, if upd_i contains only resources with security level greater than ℓ then any update in $\tilde{\Theta}_b$ is primed equivalent to upd_i (if $\tilde{\Theta}_b$ is empty we can take any update in Θ_b). Otherwise, we have to prove that there exists $j \in [1..m]$ such that $(\mathsf{upd}_i)\restriction_\ell = (\mathsf{upd}_j)\restriction_\ell$, since only resources with security level greater than ℓ are renamed. Since S_b is the ℓ-level twin of S_a, we have that the rules in R_ℓ differ from the rules in R only for resources with security level greater than ℓ. In addition, $\Sigma_a \equiv_\ell \Sigma_b$ by hypothesis, hence the two nodes agree on the values of resources with security level less or equal than ℓ. This means that such j does not exists only when a task $\varphi : \mathsf{act}$ in T is such that: *(i)* φ contains a resource with security level greater than ℓ and act assigns a resource with security level less or equal than ℓ; or *(ii)* φ does not contain any resource with security level greater than ℓ but act assigns a resource with security level less or equal than ℓ with an expression containing a resource with security level greater than ℓ. Nevertheless, neither *(i)* nor *(ii)* can happen, since by hypothesis both R and R_ℓ have not information flows from ℓ' greater than ℓ to ℓ'' less or equal than ℓ (conditions $\mathtt{IFRules}(R) = \varnothing$ and $\mathtt{IFRules}(R_\ell) = \varnothing$ of the bisimulation). Hence, it follows that $(\mathsf{S}'_a,\mathsf{S}'_b) \in \mathcal{R}$, with $\mathsf{S}'_a = R, \iota\langle\Sigma_a,\Theta'_a\rangle$ and $\mathsf{S}'_b = R_\ell, \iota_\ell\langle\Sigma_b,\Theta'_b\rangle$.

Case $\alpha = \mathsf{upd} \blacktriangleright T$. This label can be only generated by an application of the rule (StepL) (the case of (StepR) is analogous) of the AbU systems semantics (Fig. 1), where one of the nodes in S_a has applied the rule (Input) of the AbU node semantics (Fig. 1). Let $\overline{\Sigma_a} = \{\Sigma_{a,1},\dots,\Sigma_{a,n}\}$ and $\mathsf{upd} = (x_1,v_1)\dots(x_k,v_k)$. Suppose that the input has been performed by the i^{th} node, with $i \in [1..n]$. By definition of (Input), we have that $\overline{\Sigma'_a} = \overline{\Sigma_a}[\Sigma'_{a,i}/\Sigma_{a,i}]$, with $\Sigma'_{a,i} = \Sigma_{a,i}[v_1/x_1 \dots v_k/x_k]$. However, an input denotes a modification of the resources made by an external entity. Thus, this label does not depend on the actual system and can always be performed both by S_a and S_b (we have to maintain fairness, i.e., external inputs have to be sent to both systems). Note that $\overline{R_a}$ and $\overline{R_b}$ differ only in some resources with clearance greater than ℓ, that are renamed, but rules structures are identical. Hence, we can assume that the input can be performed by the i^{th} node of S_b, that is $\mathsf{S}_b \overset{\beta}{\Rightarrow}_{h_\ell} \mathsf{S}'_b$, with $\beta = \mathsf{upd} \blacktriangleright T'$, for some T'. Again, by definition of (Input), we have that $\overline{\Sigma'_b} = \overline{\Sigma_b}[\Sigma'_{b,i}/\Sigma_{b,i}]$, with $\Sigma'_{b,i} = \Sigma_{b,i}[v_1/x_1 \dots v_k/x_k]$. Since $\overline{\Sigma_a} \equiv_\ell \overline{\Sigma_b}$ and states are updated in the same manner, we have that $\overline{\Sigma'_a} \equiv_\ell \overline{\Sigma'_b}$. The only problem may arise when upd

contains resources that have been renamed: in this case S_b cannot update them. But renamed resources can only be on resources with clearance greater than ℓ, hence they do not affect states ℓ-equivalence. Since we remove from $\overline{\Theta_a}$ and $\overline{\Theta_b}$ a pair of updates primed equivalent, obtaining the pool sets $\overline{\Theta'_a}$ and $\overline{\Theta'_b}$, we have that $\overline{\Theta'_a}$ and $\overline{\Theta'_b}$ are primed equivalent. Hence, it follows that $(\mathsf{S}'_a, \mathsf{S}'_b) \in \mathcal{R}$.

Case $\alpha = \mathsf{upd} \triangleright T$**.** This action can be only derived by an application of the rule (StepL) (the case of (StepR) is analogous) of the AbU systems semantics (Fig. 1), where one of the nodes in S_a has applied the rule (Exec) of the AbU node semantics (Fig. 1). Suppose that $\mathsf{upd} = (x_1, v_1) \dots (x_k, v_k)$. We have three sub-cases, depending on the security level of the resources $x_1, \dots, x_k$.

Sub-case $\ell \sqsubset \bigsqcap_{i\in[1..k]} \mathcal{P}(x_i)$**.** Then, we have that all resources in the update have clearance greater than ℓ and, hence, $h_\ell(\alpha) = \diamond$. By definition of AbU hiding bisimulation, α can always be mimicked by an arbitrary number (possibly 0) of hidden actions (i.e., labels β such that $h_\ell(\beta) = \diamond$). Since $\overline{\Theta_a}$ and $\overline{\Theta_b}$ are primed equivalent, we can select $\beta = \mathsf{upd}' \triangleright T'$ such that upd and upd′ are primed equivalent, for some T'. Hence, we can perform $\mathsf{S}_b \overset{\beta}{\Rightarrow}_{h_\ell} \mathsf{S}'_b$, since $h_\ell(\beta) = \diamond$. Note that, all resources in upd and upd′ have clearance greater than ℓ, so resources at security level ℓ or below are not modified, implying $\overline{\Sigma'_a} \equiv_\ell \overline{\Sigma'_b}$. Since we remove from $\overline{\Theta_a}$ and $\overline{\Theta_b}$ a pair of updates primed equivalent, obtaining the pool sets $\overline{\Theta'_a}$ and $\overline{\Theta'_b}$, we have that $\overline{\Theta'_a} \doteq \overline{\Theta'_b}$. Hence, it follows that $(\mathsf{S}'_a, \mathsf{S}'_b) \in \mathcal{R}$.

Sub-case $\bigsqcap_{i\in[1..k]} \mathcal{P}(x_i) \sqsubseteq \ell$ **and** $\ell \not\sqsubseteq \bigsqcup_{i\in[1..k]} \mathcal{P}(x_i)$**.** Then, we have that at least one (but not all) resource in the update is at security level ℓ or below and, hence, $h_\ell(\alpha) = \mathsf{upd}|_\ell \neq \mathsf{upd}$. Since $\overline{\Theta_a} \doteq \overline{\Theta_b}$, then there exists upd′ in the i^{th} pool of $\overline{\Theta_b}$ such that upd and upd′ are primed equivalent. Since primed equivalent updates potentially differ on primed resources only and primed resources can only have clearance greater than ℓ, we have that $\mathsf{upd}'|_\ell = \mathsf{upd}|_\ell$. This implies that $h_\ell(\beta) = \mathsf{upd}|_\ell$, where $\beta = \mathsf{upd}' \triangleright T'$, for some T'. Hence, we can perform $\mathsf{S}_b \overset{\beta}{\Rightarrow}_{h_\ell} \mathsf{S}'_b$. Since in both systems only resources with clearance greater than ℓ are modified, we have that $\overline{\Sigma'_a} \equiv_\ell \overline{\Sigma'_b}$. Finally, since we remove from $\overline{\Theta_a}$ and $\overline{\Theta_b}$ a pair of updates primed equivalent, obtaining the pool sets $\overline{\Theta'_a}$ and $\overline{\Theta'_b}$, we have that $\overline{\Theta'_a} \doteq \overline{\Theta'_b}$. Hence, it follows that $(\mathsf{S}'_a, \mathsf{S}'_b) \in \mathcal{R}$.

Sub-case $\bigsqcap_{i\in[1..k]} \mathcal{P}(x_i) \sqsubseteq \ell$**.** Then, we have that all resources in the update are at security level ℓ or below and, hence, we have that $h_\ell(\alpha) = \mathsf{upd}|_\ell = \mathsf{upd}$. Since $\overline{\Theta_a} \doteq \overline{\Theta_b}$ and upd does not contain primed resources, we have that S_b can perform the same update, i.e., upd is in the i^{th} pool of $\overline{\Theta_b}$. Hence, we can perform $\mathsf{S}_b \overset{\alpha}{\Rightarrow}_{h_\ell} \mathsf{S}'_b$. Since both systems perform the same action, we trivially have that $\overline{\Sigma'_a} \equiv_\ell \overline{\Sigma'_b}$ and $\overline{\Theta'_a} \doteq \overline{\Theta'_b}$. Hence, it follows that $(\mathsf{S}'_a, \mathsf{S}'_b) \in \mathcal{R}$. □

Proof of Theorem 4 We prove here the soundness of the proposed *safety* verification mechanism, namely that the syntactic check provided in Algorithm 2 implies the (semantic) transparency of Definition 5.

Theorem 4 (Soundness for Safety) *Let* $\overline{R^{\mathsf{S}}}$ *and* $\overline{i^{\mathsf{S}}}$ *be the rule list and invariant sets for the AbU system* S*, and* $\overline{R^{\mathsf{R}}}$ *and* $\overline{i^{\mathsf{R}}}$ *be the rule list and invariant sets of the AbU system* R*. If* $\mathtt{TransparencyCheck}(\overline{R^{\mathsf{S}}}, \overline{R^{\mathsf{R}}}) = \mathbf{true}$*, then* $(\overline{R^{\mathsf{S}}}, \overline{i^{\mathsf{S}}}) \dashv\circ (\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}})$.

Proof. Let $\overline{R^{\mathsf{S}}}$ and $\overline{i^{\mathsf{S}}}$ be the rule list and invariant sets for the AbU system S, and $\overline{R^{\mathsf{R}}}$ and $\overline{i^{\mathsf{R}}}$ be the rule list and invariant sets of the AbU system R. Assume that $\mathtt{TransparencyCheck}(\overline{R^{\mathsf{S}}}, \overline{R^{\mathsf{R}}}) = \mathbf{true}$, that implies $\mathsf{snk}(\overline{R^{\mathsf{S}}}) \cap \mathsf{src}(\overline{R^{\mathsf{R}}}) = \emptyset$. Then, we have to prove that for any $\overline{\Sigma} \in \mathsf{comp}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}})$ we have that $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma}) \approx_{H_{\mathsf{S}}} \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma})$, where h_{S} maps labels of the form T and $\mathsf{upd} \triangleright T$, with $\mathsf{source}(\mathsf{upd}) = \overline{R^{\mathsf{S}}}$, to $\diamond$; maps labels of the form $\mathsf{upd} \triangleright T$, with $\mathsf{source}(\mathsf{upd}) \neq \overline{R^{\mathsf{S}}}$, to $\mathsf{upd} \triangleright T$; and maps labels of the form $\mathsf{upd} \blacktriangleright T$ to $\mathsf{upd} \blacktriangleright T$. The proof is by contradiction.

Suppose that $\mathtt{TransparencyCheck}(\overline{R^{\mathsf{S}}}, \overline{R^{\mathsf{R}}}) = \mathbf{true}$ but $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma}) \not\approx_{H_{\mathsf{S}}} \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma})$, for some $\overline{\Sigma}$. This means that it does not exist an AbU hiding bisimulation $\mathcal{R}$, parametric on H_{S}, that contains the pair $(\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma}), \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma}))$. 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 $(\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma}), \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma}))$, the bisimulation game stops in a pair $(\mathsf{S}_a, \mathsf{S}_b)$, with S_a and S_b derivatives of $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma})$ and $\mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma})$, respectively.

This may happen because of either: S_a can perform an action labeled α that cannot be (weakly) mimicked by S_b (or vice versa); or a mimicking action is always possible but it leads us to pairs of the form $(\mathsf{S}'_a, \mathsf{S}'_b)$ that 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 the second case never applies.

Let $\mathsf{S}_a = \mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma}_a, \overline{\Theta}_a)$ and $\mathsf{S}_b = \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma}_b, \overline{\Theta}_b)$, derivatives[5] of $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{S}}} \cup \overline{i^{\mathsf{R}}}, \overline{\Sigma})$ and $\mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{i^{\mathsf{R}}}, \overline{\Sigma})$, respectively. We proceed by case analysis on the action α that would distinguish the two elements S_a and S_b.

[5] Recall that, when the pool is empty we omit it from the notation $\mathsf{sys}(\overline{R}, \overline{i}, \overline{\Sigma})$.

Case $\alpha = T$. By definition of the AbU semantics (Fig. 1), the label T can only be generated by a system composed by a single node, by applying the rule (Disc). This implies that either S or R must be empty, but empty systems are not allowed by AbU syntax. Hence, such case cannot happen.

Case $\alpha = \mathsf{upd} \blacktriangleright T$. This action can be only derived by an application of the rule (StepL) (the case of (StepR) is analogous) of the AbU systems semantics (Fig. 1), where one of the nodes in S_a has applied the rule (Input) of the AbU node semantics (Fig. 1). However, this action denotes a modification of the resources made by an external entity. Thus, this action does not depend on the actual system and can always be performed by both S_a and S_b (we have to maintain fairness, i.e., external inputs have to be sent to both systems).

Case $\alpha = \mathsf{upd} \triangleright T$. This action can be only derived by an application of the rule (StepL) (the case of (StepR) is analogous) of the AbU systems semantics (Fig. 1), where one of the nodes in S_a has applied the rule (Exec) of the AbU node semantics (Fig. 1). We have two sub-cases, depending on the rule set that this node belongs to.

Sub-case $\mathsf{source}(\mathsf{upd}) = \overline{R^{\mathsf{S}}}$. Then, we have that the node belongs to $\overline{R^{\mathsf{S}}}$ and, hence, $h_{\mathsf{S}}(\alpha) = \diamond$. By definition of AbU hiding bisimulation, α can always be mimicked by an arbitrary number (possibly 0) of hidden actions (i.e., labels β such that $h_{\mathsf{S}}(\beta) = \diamond$). In particular, the system S_b is allowed to not progress, without breaking the bisimulation game.

Sub-case $\mathsf{source}(\mathsf{upd}) \neq \overline{R^{\mathsf{S}}}$. Then, we have that the node belongs to $\overline{R^{\mathsf{R}}}$ and, hence, $h_{\mathsf{S}}(\alpha) = \mathsf{upd} \triangleright T$. As α is the distinguishing action, it follows that the node reaches different states in S_a and S_b, leading to the following situation: the update upd is possible in S_a but not in S_b (or vice versa). Since both rule sets $\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}$ and $\overline{R^{\mathsf{R}}}$ start in the same execution state set $\overline{\Sigma}$ (and with all pools empty), the rule set $\overline{R^{\mathsf{R}}}$ could exhibit different behaviors if and only if it would be affected by $\overline{R^{\mathsf{S}}}$. In particular, this means that in the execution trace leading $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{S}}} \cup \overline{\imath^{\mathsf{R}}}, \overline{\Sigma})$ to S_a, one rule in $\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}$ should have modified either: *(i)* a resource that a rule in $\overline{R^{\mathsf{R}}}$ listens on; or *(ii)* a resource that is accessed by a rule in $\overline{R^{\mathsf{R}}}$. Note that, the accessed resource not necessarily has to be used to assign other resources: it can be used into task condition in order to modify the rule's control flow. However, syntactic transparency $\mathtt{TransparencyCheck}(\overline{R^{\mathsf{S}}}, \overline{R^{\mathsf{R}}}) = \mathbf{true}$, i.e., $\mathsf{snk}(\overline{R^{\mathsf{S}}}) \cap \mathsf{src}(\overline{R^{\mathsf{R}}}) = \emptyset$, is trivially preserved by all derivatives of the initial systems (rules do not syntactically change during execution). This ensures that neither case applies.

As it does not exist a distinguishing action α, it follows that the original systems $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{S}}} \cup \overline{\imath^{\mathsf{R}}}, \overline{\Sigma})$ and $\mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}}, \overline{\Sigma})$ must be hiding bisimilar, i.e., $\mathsf{sys}(\overline{R^{\mathsf{S}}} \cup \overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{S}}} \cup \overline{\imath^{\mathsf{R}}}, \overline{\Sigma}) \approx_{H_{\mathsf{S}}} \mathsf{sys}(\overline{R^{\mathsf{R}}}, \overline{\imath^{\mathsf{R}}}, \overline{\Sigma})$, where h_{S} maps labels of the form T and $\mathsf{upd} \triangleright T$, with $\mathsf{source}(\mathsf{upd}) = \overline{R^{\mathsf{S}}}$, to $\diamond$; maps labels of the form $\mathsf{upd} \triangleright T$, with $\mathsf{source}(\mathsf{upd}) \neq \overline{R^{\mathsf{S}}}$, to $\mathsf{upd} \triangleright T$; and maps labels of the form $\mathsf{upd} \blacktriangleright T$ to $\mathsf{upd} \blacktriangleright T$. □

References

[1] J. Cano, E. Rutten, G. Delaval, Y. Benazzouz, L. Gurgen, ECA rules for IoT environment: a case study in safe design, in: 8th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops, IEEE, USA, 2014, pp. 116–121.

[2] M. Balliu, M. Merro, M. Pasqua, M. Shcherbakov, Friendly fire: cross-app interactions in IoT platforms, ACM Trans. Priv. Secur. 24 (3) (2021) 16:1–16:40, https://doi.org/10.1145/3444963.

[3] M. Miculan, M. Pasqua, A calculus for attribute-based memory updates, in: A. Cerone, P.C. Ölveczky (Eds.), Theoretical Aspects of Computing – ICTAC 2021, Springer International Publishing, Cham, 2021, pp. 366–385.

[4] Y. Abd Alrahman, R. De Nicola, M. Loreti, On the power of attribute-based communication, in: E. Albert, I. Lanese (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, Springer Int. Pub., Cham, 2016, pp. 1–18.

[5] Y. Abd Alrahman, R. De Nicola, M. Loreti, Programming interactions in collective adaptive systems by relying on attribute-based communication, Sci. Comput. Program. 192 (2020) 102428, https://doi.org/10.1016/j.scico.2020.102428.

[6] E. Cohen, Information transmission in computational systems, Oper. Syst. Rev. 11 (1977) 133–139.

[7] M. Pasqua, M. Miculan, On the security and safety of abu systems, in: Proc. SEFM, in: Lecture Notes in Computer Science, vol. 13085, Springer, 2021, pp. 178–198.

[8] M. Pasqua, M. Miculan, Distributed programming of smart systems with event-condition-action rules (short paper), in: Proc. ICTCS, in: CEUR Workshop Proceedings, vol. 3284, CEUR-WS.org, 2022, pp. 201–206.

[9] M. Pasqua, M. Miculan AbU, A calculus for distributed event-driven programming with attribute-based interaction, Theor. Comput. Sci. 958 (2023) 113841, https://doi.org/10.1016/J.TCS.2023.113841.

[10] B. Givoni, Comfort, climate analysis and building design guidelines, Energy Build. 18 (1) (1992) 11–23.

[11] W. Diffie, M. Hellman, New directions in cryptography, IEEE Trans. Inf. Theory 22 (6) (1976) 644–654.

[12] G. Barthe, P.R. D'Argenio, T. Rezk, Secure information flow by self-composition, in: Proc. of CSF, 2004, pp. 100–114.

[13] M.R. Clarkson, F.B. Schneider, Hyperproperties, J. Comput. Secur. 18 (6) (2010) 1157–1210, http://dl.acm.org/citation.cfm?id=1891823.1891830.

[14] I. Mastroeni, M. Pasqua, Verifying bounded subset-closed hyperproperties, in: A. Podelski (Ed.), Static Analysis, Springer Int. Pub., Cham, 2018, pp. 263–283.

[15] A. Sabelfeld, A.C. Myers, A model for delimited information release, in: International Symposium - Software Security, ISSS, in: Lecture Notes in Computer Science, vol. 3233, Springer, 2003, pp. 174–191.

[16] Y. Abd Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, R. Vigo, A calculus for attribute-based communication, in: 30th Symposium on Applied Computing, ACM, 2015, pp. 1840–1845.

[17] R. De Nicola, D. Latella, A.L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, A. Vandin, The SCEL language: design, implementation, verification, in: M. Wirsing, M. Hölzl, N. Koch, P. Mayer (Eds.), Soft. Eng. for Collective Autonomic Systems, vol. 8998, Springer, 2015, pp. 3–71.

[18] S. Anderson, N. Bredeche, A. Eiben, G. Kampis, M. van Steen, Adaptive collective systems: herding black sheep, 2013.

[19] M. Balliu, I. Bastys, A. Sabelfeld, Securing IoT apps, IEEE Secur. Priv. 17 (5) (2019) 22–29, https://doi.org/10.1109/MSEC.2019.2914190.

[20] Z.B. Celik, E. Fernandes, E. Pauley, G. Tan, P. McDaniel, Program analysis of commodity IoT applications for security and privacy: challenges and opportunities, ACM Comput. Surv. 52 (4) (Aug. 2019), https://doi.org/10.1145/3333501.

[21] R. Focardi, R. Gorrieri, Classification of security properties (part I: information flow), in: FOSAD, FOSAD'00, Springer-Verlag, 2001, pp. 331–396.

[22] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, L. Jia, Some recipes can do more than spoil your appetite: analyzing the security and privacy risks of IFTTT recipes, in: WWW'17, ACM, 2017, pp. 1501–1510.
[23] Z.B. Celik, P.D. McDaniel, G. Tan, Soteria: automated IoT safety and security analysis, in: USENIX, USENIX Association, Boston, MA, 2018, pp. 147–158, https://www.usenix.org/conference/atc18/presentation/celik.
[24] Z.B. Celik, G. Tan, P.D. McDaniel, IoTGuard: dynamic enforcement of security and safety policy in commodity IoT, in: NDSS, The Internet Society, 2019.
[25] H. Chi, Q. Zeng, X. Du, J. Yu, Cross-app interference threats in smart homes: categorization, detection and handling, in: 50th Int. Con. on Dependable Systems and Networks, 2020, pp. 411–423.
[26] W. Ding, H. Hu, On the safety of IoT device physical interaction control, in: ACM CCS, CCS'18, ACM, 2018, pp. 832–846.
[27] D.T. Nguyen, C. Song, Z. Qian, S.V. Krishnamurthy, E.J.M. Colbert, P. McDaniel, IoTSan: Fortifying the Safety of IoT Systems, in: CoNEXT'18, ACM, 2018, pp. 191–203.
[28] K. Hsu, Y. Chiang, H. Hsiao SafeChain, Securing trigger-action programming from attack chains, IEEE Trans. Inf. Forensics Secur. 14 (10) (2019) 2607–2622.
[29] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, A. Prakash, FlowFence: practical data protection for emerging IoT application frameworks, in: USENIX, USENIX Association, 2016, pp. 531–548.
[30] Z.B. Celik, L. Babun, A.K. Sikder, H. Aksu, G. Tan, P.D. McDaniel, A.S. Uluagac, Sensitive information tracking in commodity IoT, in: USENIX, USENIX Association, 2018, pp. 1687–1704.
[31] I. Bastys, M. Balliu, A. Sabelfeld, If this then what? Controlling flows in IoT apps, in: ACM CCS, 2018, pp. 1102–1119.
[32] I. Bastys, F. Piessens, A. Sabelfeld, Tracking information flow via delayed output - addressing privacy in IoT and emailing apps, in: NordSec, in: LNCS, vol. 11252, Springer, 2018, pp. 19–37.
[33] E. Fernandes, A. Rahmati, J. Jung, A. Prakash, Decentralized action integrity for trigger-action IoT platforms, in: NDSS, The Internet Society, 2018.
[34] C. Vannucchi, M. Diamanti, G. Mazzante, D.R. Cacciagrano, F. Corradini, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, vIRONy: a tool for analysis and verification of ECA rules in intelligent environments, in: Int. Conf. on Intell. Environ., S. Korea, IEEE, 2017, pp. 92–99.
[35] C. Vannucchi, M. Diamanti, G. Mazzante, D.R. Cacciagrano, R. Culmone, N. Gorogiannis, L. Mostarda, F. Raimondi, Symbolic verification of event-condition-action rules in intelligent environments, J. Reliable Intell. Environ. 3 (2) (2017) 117–130, https://doi.org/10.1007/s40860-017-0036-z.
[36] F. Corradini, R. Culmone, L. Mostarda, L. Tesei, F. Raimondi, A constrained ECA language supporting formal verification of WSNs, in: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, 2015, pp. 187–192.
[37] X. Jin, Y. Lembachar, G. Ciardo, Symbolic verification of ECA rules, in: D. Moldt (Ed.), Joint Proceedings of PNSE'13 and ModBE'13, Milano, Italy, vol. 989, CEUR-WS.org, 2013, pp. 41–59, http://ceur-ws.org/Vol-989/paper17.pdf.
[38] D. Beyer, A. Stahlbauer, BDD-based software verification, Int. J. Softw. Tools Technol. Transf. 16 (5) (2014) 507–518, https://doi.org/10.1007/s10009-014-0334-1.
[39] J. Cano, G. Delaval, E. Rutten, Coordination of ECA rules by verification and control, in: E. Kühn, R. Pugliese (Eds.), Coordination Models and Languages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 33–48.
[40] J.L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, M. Sridharan, IOTA: a calculus for Internet of things automation, in: New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!, 2017, pp. 119–133.
[41] C. Bodei, P. Degano, G.L. Ferrari, L. Galletta, Tracing where IoT data are collected and aggregated, Log. Methods Comput. Sci. 13 (3) (2017) 1–38, https://doi.org/10.23638/LMCS-13(3:5)2017.
[42] D.M. Volpano, C.E. Irvine, G. Smith, A sound type system for secure flow analysis, J. Comput. Secur. 4 (2/3) (1996) 167–188.
[43] S. Hunt, D. Sands, On flow-sensitive security types, in: Conf. Rec. of the 33rd Symposium on Principles of Programming Languages, POPL'06, ACM, New York, NY, USA, 2006, pp. 79–90.
[44] M. Balliu, M. Merro, M. Pasqua, Securing cross-app interactions in IoT platforms, in: 32nd IEEE Computer Security Foundations Symposium, IEEE, Hoboken, NJ, USA, 2019, pp. 319–334.
[45] F. Honsell, M. Miculan, A natural deduction approach to dynamic logic, in: Proc. TYPES, in: Lecture Notes in Computer Science, vol. 1158, Springer, 1995, pp. 165–182.
[46] M. Miculan, On the formalization of the modal μ-calculus in the calculus of inductive constructions, Inf. Comput. 164 (1) (2001) 199–231, https://doi.org/10.1006/inco.2000.2902.
[47] K. Chaudhuri, D. Doligez, L. Lamport, S. Merz, Verifying safety properties with the TLA+ proof system, in: International Joint Conference on Automated Reasoning, Springer, 2010, pp. 142–148.
[48] A. Mansutti, M. Miculan, M. Peressotti, Multi-agent systems design and prototyping with bigraphical reactive systems, in: Proc. DAIS, in: Lecture Notes in Computer Science, vol. 8460, Springer, 2014, pp. 201–208.