sensors

MDPI

Article

IoT Security Configurability with Security-by-Contract

Alberto Giaretta [1,*], Nicola Dragoni [1,2,*] and Fabio Massacci [3,*]

1 Centre for Applied Autonomous Sensors Systems (AASS), Örebro University, 701 82 Örebro, Sweden
2 DTU Compute, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
3 Department of Information Science and Engineering, University of Trento, 38123 Trento, Italy
* Correspondence: alberto.giaretta@oru.se (A.G.); ndra@dtu.dk (N.D.); fabio.massacci@unitn.it (F.M.)

Received: 15 August 2019; Accepted: 19 September 2019; Published: 23 September 2019

Abstract: Cybersecurity is one of the biggest challenges in the Internet of Things (IoT) domain, as well as one of its most embarrassing failures. As a matter of fact, nowadays IoT devices still exhibit various shortcomings. For example, they lack secure default configurations and sufficient security configurability. They also lack rich behavioural descriptions, failing to list provided and required services. To answer this problem, we envision a future where IoT devices carry behavioural contracts and Fog nodes store network policies. One requirement is that contract consistency must be easy to prove. Moreover, contracts must be easy to verify against network policies. In this paper, we propose to combine the security-by-contract ($S \times C$) paradigm with Fog computing to secure IoT devices. Following our previous work, first we formally define the pillars of our proposal. Then, by means of a running case study, we show that we can model communication flows and prevent information leaks. Last, we show that our contribution enables a holistic approach to IoT security, and that it can also prevent unexpected chains of events.

Keywords: IoT; configurability; Fog computing; security-by-contract; security

1. Introduction

Pervasive computing foresees a future where computing appears anytime and everywhere. To realise such a visionary paradigm, everyday objects have to embed two important features: computational power and connectivity. The disruptive advent of Internet of Things (IoT) represents a first important step towards the realisation of such a pervasive computing vision. Not only can IoT devices form complex interconnected systems that exchange any kind of information, they can also connect to the Internet and interpret such information in a broader context. However, as for every technical result, some aspects of the IoT proved to be problematic. In particular, cybersecurity has been so far one of the biggest challenges for the IoT, as well as one of its most embarrassing failures [1].

Since the advent of IoT, the security research community has made an effort to raise awareness on the security challenges that the new paradigm brought. Some challenges, such as weak passwords or the lack of security updates, stem from best practices overlooked by manufacturers at the design phase [1]. Others, such as integration and automatic configuration, derive from the heterogeneous nature of the IoT itself [2].

As a matter of fact, IoT devices can have radically different goals. They range from smart plugs that turn appliances on/off, to fleets of devices that connect whole factories. These devices have different hardware and software requirements. They can equip different sensors, different communication protocols, and even different amounts of computing power. Heterogeneity is a strong point of IoT, and a weak one at the same time. On the one hand, the variety of devices enables to tailor solutions on specific requirements. On the other hand, achieving a robust and secure infrastructure is far harder than it used to be with traditional networks.

Moreover, many IoT devices are computationally limited and, aiming to overcome this limitation, manufacturers rely upon Cloud computing to undertake heavy workloads. The key problem with this two layer approach is that Cloud computing has not been designed for the volume, variety, and velocity of data that IoT generates. Some intrinsic drawbacks, such as unpredictable network latency and uncertain storage location, are particularly problematic for security-related IoT generated data.

To counterbalance these issues, Fog computing has been proposed as an extension to Cloud computing [3]. The key idea is to move critical services from the Cloud to the edge of the network, close to IoT devices. Typically installed in a network in the form of a dedicated device (i.e., the Fog node), the Fog provides a virtualized middle layer that sits between latency-sensitive applications and the Cloud. Its goal is to remedy some of the Cloud drawbacks by centralizing sensors data, undertaking time-bound tasks, and serving as a data traffic gatekeeper. Moreover, a Fog node is a trustworthy device that can store security-related data, such as network policies, and perform real-time policy enforcement techniques (e.g., enforcement based on traffic monitoring).

1.1. Contributions of the Paper

In this paper, we propose to combine security-by-contract (S $\times$ C) and Fog computing to strengthen IoT systems security. S $\times$ C have previously been applied for similar goals to various domains, such as mobile applications [4,5] and multi-application smart cards [6].

This work focuses on addressing one of the yet to be solved classes of threats concerning IoT systems, namely insufficient security configurability [2]. In a nutshell, current IoT devices offer little ways for configuring them according to to users' necessities. For example, often IoT devices entirely neglect access control granularity, since they only provide the administrator login. In particular, the contributions of this paper are manifold:

- We provide a formal description of the pillars of the S $\times$ C approach, such as security rules, contracts, policies.
- We show how these concepts can be combined together to secure IoT systems.
- We define the notion of consistency for both contracts and policies.
- Based on the above concepts, we define several properties useful for the security of IoT systems, such as contract-policy matching and illegal information flow.
- We provide algorithmic versions of our formal definitions and methods, laying the foundation for a real-world implementation.
- We illustrate these concepts by means of a running case study, based on real-world smart home (E-care@home (http://ecareathome.se)).
- We show how S $\times$ C can successfully be integrated with Fog computing to secure IoT systems.
- We extensively discuss pros and cons of our approach in a threat model.

1.2. Paper Outline

This paper is organized as follows. In Section 2 we give an overview of IoT open security challenges. In Section 3 we present the general idea of the application of S $\times$ C to the IoT, and the running case study we use throughout our work. Then, in Sections 4–6 we define and describe the building blocks of our proposal, namely security rules, security contracts, and security policies. In Section 7 we show that our approach can also identify and prevent hidden communication paths stemming from physical communication channels. In Section 8 we summarize the potential threats to our Fog-based solution. Then, we discuss benefits and limitations of S $\times$ C in this context. In Section 9 we report the relevant related work. In Section 10 we discuss the ongoing development and the future work we have planned. Last, in Section 11 we wrap up our contribution.

2. IoT Smart Services: Background and Challenges

In this section, we give a brief bird's-eye view on IoT specific security challenges, we summarize the related challenges, and then we clarify where our contribution sits with respect to this analysis.

In a recent paper, Zhou et al. [2] analysed the main security challenges (named features) that are characteristic of the IoT domain. They have identified eight different features, namely:

- interdependence
- diversity
- constrained
- myriad
- unattended
- intimacy
- mobile
- ubiquitous

In particular, under the umbrella of interdependence and ubiquitous features, the authors highlighted that the main threat for IoT devices is the default insecure configurations provided by the manufacturers, as well as the insufficient security configurability.

Over time, interactions between devices grow in complexity, while the human role in managing such interactions becomes less and less important. This is a complex problem from a security point of view. Manufacturers and researchers often focus on the robustness of the single device, or on the direct communications between devices. However, the devices' interdependence makes hard to define specific security rules for every and each of them, often resulting in excessive privileges, granted preemptively. Moreover, the fact that IoT devices are ubiquitous and pervasive within modern society amplifies the aforementioned threats, particularly when IoT devices are manufactured with insecure default configurations.

At the time of writing, there are plenty of examples of insecure configurations and insufficient security configurability. Passwords are oftentimes hardcoded within the IoT devices and, if they can be changed, there are no entropy checks in place. There is no permission granularity, meaning that either the user logs in as admin or not at all, which impedes to develop layered security approaches which involve such devices. Other devices have services and connections ports open for everyone in the name of availability, in case those services are ever needed, and usually no monitoring over information flows are in place within IoT local networks.

Moreover, little self-configuration capabilities are present in current IoT devices. As pointed out by Athreya et al. [7], the cloud-centric approach to IoT management is likely to fail in the future, due to the scalability limitation that it entails. IoT devices need to be in charge of configuring and adapting themselves to the environment they belong to, without negatively affecting other devices and services. However, this is possible only if IoT devices declare to the environment their behavioural specifications.

Without behavioural descriptions, plugging an IoT device into an established infrastructure opens up to several uncertainties. What services does the device offer? Which ones does it need? Over which communication protocols? Does this device disrupt or disturb other running services? These are just few examples, out of many more.

Aiming to address this specific issue, Cisco proposed Manufacturer Usage Descriptions (MUD) [8], an authoritative identifier that allows manufacturers to describe the identity and the intended behaviour of the devices they produce. Intuitively, IoT devices carry an URI that redirects to a certified website, where the MUD file describing their behaviour is hosted. A trusted device within the private network, namely the MUD server, is responsible for reading the URI, pulling the MUD file, and granting privileges accordingly. Even though MUD has shown the way to tractable descriptions and configurations, in its current form it merely provides a device identification and a list of the addresses/ports used. In fact, MUD profiles are not rich enough to model complex behaviour, such as device-based permissions. Moreover, MUD profiles do not allow easy verification of device behaviour against the network security policy, nor against the intended behaviour of other devices in the same network. In this paper, we contribute to partially amending such shortcomings.

Last, in the aforementioned work, Zhou et al. [2] classified the body of literature about IoT security, and identified seven different classes of threats on which researchers have focused until today:

- privacy leaks
- insecure network communication/protocol
- vulnerable cloud/web service
- insecure mobile application
- vulnerable system/firmware
- insufficient security configurability
- other threats

Their analysis showed that the largest amount of research went into privacy leaks and insecure network communication issues, with different percentages for different scenarios. What stands out is that, although critical for the IoT, the insufficient configurability has still to be thoroughly addressed by the community. We believe that this research gap is due to the complex traits of this problem. Our effort aims at specifically addressing this key open challenge.

Running Case Study: E-Care@Home

E-care@Home is a Swedish interdisciplinary distributed research environment that pulls together competences in artificial intelligence, semantic web, IoT and sensors for health [9]. Ängen, a real smart home composed by several IoT devices, was created within the E-care@home initiative (Figure 1 shows Ängen layout). Each room is equipped with PIR (passive infrared) motion sensors, running over different boards, such as Pycom WiPy 3.0 boards, and different boards running Contiki OS. Bed, sofa, and chairs use binary pressure switches to detect whether someone is sitting. In the kitchen, the oven has an on/off sensor. A smart light Philips Hue White v3.0, a Philips Hue Motion v1.0 motion sensor, and a smart speaker Amazon Echo v1.0 are installed in the living room. Moreover, in the living room we have Pepper, a humanoid robot from manufacturer SoftBank. For safety reasons, in the future there will be a D-Link DCS-933L security IoT camera right outside the facility door. Last, all devices are integrated through the E-care@Home middleware, from which a context reasoner extracts the necessary information. Currently, a simple laptop collects the ground truth, and infers information through a reasoner. In our future plans, captured by Figure 1, a Fog node will replace the laptop, and take on the necessary security tasks.

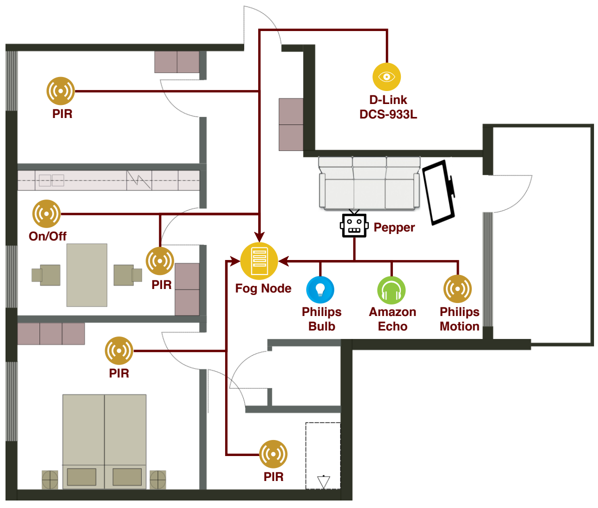

Figure 1. Ängen context-aware smart home, equipped with various sensors, like motion and pressure ones.

3. S × C Fundamentals

The S × C framework [4,5] is based on two key concepts, namely *contract* and *policy*.

A security contract (or simply contract) represents the specification of the behaviour of an IoT device for what concerns relevant security actions. An IoT device is equipped with a contract, and this contract has to be exhibited to the Fog node before being accepted to participate to the network.

A security policy (or simply policy) is instead a specification of the acceptable behaviour of the IoT devices a Fog node is responsible for, for what concerns relevant security actions.

It is clear that, to refine the concepts of contract and policy in the IoT domain, it is necessary to narrow down the meaning of relevant security actions. For the scope of this paper, we restrict the set of possible relevant security actions to the one of the possible interactions among IoT devices governed by a Fog node (for a broader set of actions we refer to Giaretta et al. [10]). In this particular setting, a contract describes which resources are necessary for the device to operate, and which resources the device provides to others. A policy declares what actions are allowed within that specific context, and what resources all the included devices need/provide.

As shown in Figure 2, whenever an IoT device attempts to join the network its contract is parsed by the Fog node, which runs the matching algorithm and decides if the IoT device is compatible with the network policy, or not. Besides, if the device contract or its software are updated, the matching process re-starts, to ensure that the device is still compatible with the network policy.

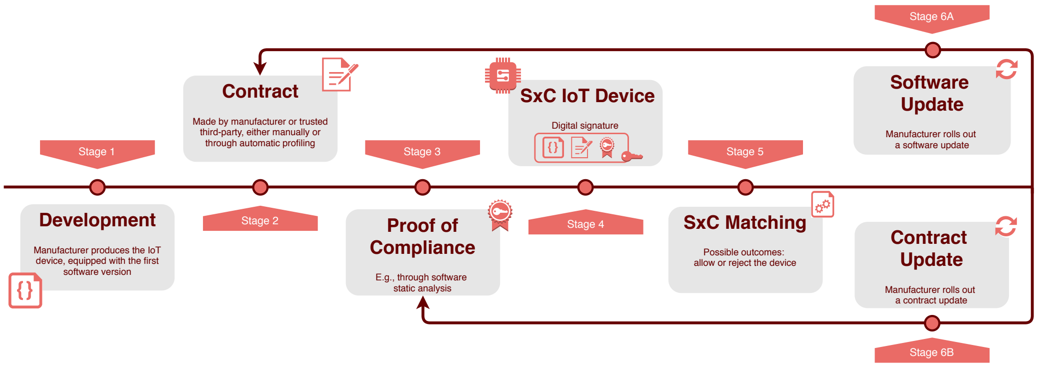
Stage 1
Development
Manufacturer produces the IoT device, equipped with the first software version
Stage 2
Contract
Made by manufacturer or trusted third-party, either manually or through automatic profiling
Stage 3
Proof of Compliance
E.g., through software static analysis
Stage 4
SxC IoT Device
Digital signature
Stage 5
SxC Matching
Possible outcomes: allow or reject the device
Stage 6A
Software Update
Manufacturer rolls out a software update
Stage 6B
Contract Update
Manufacturer rolls out a contract update

Figure 2. Security-by-contract (S × C) contract lifecycle. In Stage 4, the manufacturer digitally signs all the components, achieving software validation and non-repudiability: S × C provides semantics for digital signatures on an IoT code.

3.1. Who Writes Contracts and Policies?

A question that naturally comes when we talk about contracts and policies is who are the actors that define them.

With regards to contracts, ideally they should be defined by the company manufacturing the IoT devices, for two main reasons. First, the user should have the assurance that, whenever he buys a new device and plugs it into a networked environment, right away the device is going to do exactly what it claims, without interfering with existing services. To this aim, the manufacturer can equip the device with proof-of-compliance (PoC), which binds the IoT software to the contract (proving that the software is compliant with the contract). This is a fundamental component of the S × C approach that, for space limitation, we do not touch on in this paper (more details in [10] and S × C literature). Second, it would be much easier for the manufacturer to create complete contracts from the beginning, instead of trying to define later a contract through reverse-engineering approaches which might leave out some hidden behaviour. Besides, this should put some pressure on manufacturers, forcing them to implement healthy lifecycles for their products.

However, this is not achievable for already-produced IoT devices. In this case, it is possible to extract (possibly incomplete) contracts through profiling routines. A device can be sandboxed and the consequent analysis can be performed over its traffic, in order to figure out its observed behaviour. A similar approach has been used with MUD policies by Hamza et al. [11], for extracting MUD profiles from legacy IoT devices. We also foresee that contracts can be customized at the time of first booting. As an example, when a smart light is installed it might ask which smartphone will be controlling it, through the manufacturer app.

On the other hand, policies are more complex than contracts and their sources might be quite different. For example, policies could be handed out by smart home manufacturers, consumer protection organizations, security companies, or non-profit organizations (e.g., the Electronic Frontier Foundation), in the same way that virtual operating system (OS) boxes are distributed (https://www.osboxes.org/virtualbox-images/). Moreover, policies could be defined from scratch by the network administrator himself, or created using the aforementioned ones as templates.

3.2. Implications

Security solutions should be kept simple in order to be effective. Overcomplicated tasks are likely to drive users and administrators away from security, or to encourage dangerous shortcuts. With this in mind S × C aims to have as simple results as possible.

With respect to simple users, S × C is completely transparent. Users' devices store their own contract and, at connection time, the Fog node is responsible for verifying them. In the case that the contract is compliant with the policy, the device will be simply accepted. On the contrary, if the contract violates the policy, the user will be notified that the device is not compliant with the network policies.

Regarding network administrators, a little overhead is required for them. Administrators are responsible for managing the network and its policies, and the S × C policy would be part of their responsibility. However, as shown throughout this paper, policies are simple security rules, relatively easy to write and edit for administrators. Moreover, as mentioned in 3.1, S × C policies and templates could be written by trusted third parties. It would be easy for an administrator to start with a template and edit it according to his network necessities.

Last, manufacturers are the most affected by the introduction of S × C. First, they have the responsibility for writing their products' contracts. Second, they write the PoC which formally binds code and contract. Moreover, whenever the manufacturer rolls out an update, it has to update both contract and PoC, accordingly. Even though this is a considerable overhead for a manufacturer, it can also be considered a great opportunity for improving product lifecycles. As a matter of fact, following an update with a consistency check between contract and actual behaviour not only protects the customers, it also helps manufacturers themselves to keep track of their own products. In Table 1 we summarise roles, actions, and implications.

Table 1. Roles, actions, and implications in the S × C setting.

	Actions	**Implications**
User	Connects his S × C device to the network	Completely transparent for the user
Administrator	Manages the S × C network policy	Small overhead, in the form of policy creation and management
Manufacturer	Write device contract and provides PoC	Overhead in the form of contract and PoC certificate creation

4. S × C for IoT: Security Rules

In this section, we define the building blocks of our proposal. We start by describing how we refer to a device and its services. Then we define the concept of security rule.

Definition 1 ((IoT device))**.** *A device* D *is a well formed composition of a device name* D *and a manufacturer of the device* M*, expressed as* M.D*.*

Definition 2 (Service)**.** *A service* S *provided by an IoT device* D = M.D *is a well formed composition of a service name* S*, the servicing IoT device name* D *and the manufacturer of the device* M*. As a result,* S *is expressed as* M.D.S*.*

Example 1 (IoT device A)**.** *Referring to the camera* D-LINK.933L*, previously mentioned in Section 2, this device provides two different services. The first service is called* SETDAYNIGHT*, and it is responsible for switching on/off the device LED; the second service is* SETSYSTEMMOTION*, which is responsible for enabling and disabling the integrated motion sensor. In terms of an S × C contract, we would refer to these two services as* D-LINK.933L.SETDAYNIGHT *and* D-LINK.933L.SETSYSTEMMOTION*, respectively.*

Example 2 (IoT device B). *The device* PHILIPS.HUEWHITE *we defined in Section 2 provides three different services called* ON, BRI, *and* HUE. ON *is used to remotely turn on/off the light, whereas* BRI *changes the brightness, and* HUE *changes the hue, according to the user preferences. Following our formalization, in a contract we would refer to these services as* PHILIPS.HUEWHITE.ON, PHILIPS.HUEWHITE.BRI, *and* PHILIPS.HUEWHITE.HUE, *respectively.*

Definition 3 (Domain). *A domain* DOM *is a non-empty string identifying the context (in terms of network domain, such as a local area network (LAN)) where a security rule applies.*

4.1. Security Rule

Definition 4 (Security rule). *A security rule (or simply, rule)* R *is a 5-tuple represented by the fields listed in Table 2.*

Table 2. Security rule structure.

Device D	The device name D and manufacturer M of the device, expressed as M.D.
Domain DOM	The domain where the rule applies. For instance, DOM = LAN for rules that apply within the local network, or DOM = * for rules that apply to any domain.
SHARES	List of devices that the device D can interact with, in the domain DOM. We use * to denote that anything applies. Examples: M.* specifies any device from a specific manufacturer M. Similarly, *.* (or simply *) specifies any device from any manufacturer.
PROVIDES	List of services s_1, ..., s_n, $n \geq 0$, that the IoT device D provides to the devices in SHARES(D).
REQUIRES	List of services s_1, ..., s_m, $m \geq 0$ that the IoT device D requires to function.

From now on, we will use the notation R[D] to denote the device D of a security rule R. Analogously, R[DOM], R[SHARES], R[PROVIDES], and R[REQUIRES] denote the related fields of a rule R.

Example 3 (Security rule R_{A1}). *The IoT camera surveillance* D-LINK.933L *allows the owner to enable/disable the camera LED from his smartphone, over any domain. As mentioned in Section 3.1, we depict that, for example, at deployment time a smartphone can be chosen as a main controlling device. In order to express this, the resulting security rule would look as shown in Table 3.*

Table 3. Security rule R_{A1}.

	Rule R_{A1}
D	D-LINK.933L
DOM	*
SHARES	APPLE.LUKEPHONE
PROVIDES	SETDAYNIGHT
REQUIRES	-

Example 4 (Security rule R_{A2}). *Let us now suppose that the IoT camera surveillance* D-LINK.933L *enables the owner to access a* PHILIPS.HUEWHITE *device, and turn on the lightbulb whenever the internal motion sensor of the camera detects a movement. Therefore, the camera requires service* ON, *but it does not want to provide any service to other devices. In order to express this, the resulting security rule would look as shown in Table 4.*

Table 4. Security rule R_{A2}.

	Rule R_{A2}
D	D-LINK.933L
DOM	*
SHARES	-
PROVIDES	-
REQUIRES	PHILIPS.HUEWHITE.ON

Example 5 (Security rule R_{B1}). *The lightbulb* PHILIPS.HUEWHITE *provides three different services (namely* ON*,* BRI*, and* HUE*) to every* PHILIPS *device. Moreover, the device requires a motion sensor* HUEMOTION *(produced by the same manufacturer,* PHILIPS*), which provides service* PRESENCE *and enables to switch on/off the lights, depending on the presence of people in the room. Last,* HUEWHITE *is designed to communicate only within the LAN. A security rule that expresses these requirements is shown in Table 5.*

Table 5. Security rule R_{B1}.

	Rule R_{B1}
D	PHILIPS.HUEWHITE
DOM	LAN
SHARES	PHILIPS.*
PROVIDES	ON, BRI, HUE
REQUIRES	PHILIPS.HUEMOTION.PRESENCE

4.2. Well Formed and Core Security Rules

In order to avoid malformed security rules, we introduce a notion of well formed security rule. Intuitively, a security rule R concerning a device D is well formed if the rule specifies which IoT devices can use the services provided by D. Since these devices are listed in R[SHARES], we have that R[SHARES] must not be empty in case D provides some services in R (i.e., R[PROVIDES] is not empty). Note that this has several meanings: R[SHARES] could include a specific list of devices, or whatever device of one or more manufacturers, or in general whatever device of whatever manufacturer. However, R[SHARES] must not be empty if D provides some services.

Definition 5 (Well formed security rule). *A security rule* R *is said to be well formed if and only if the following condition holds: If* R[PROVIDES] *is not empty, then* R[SHARES] *must not be empty.*

In Algorithm 1, we provide a pseudo-algorithm that verifies if a rule is well-formed and returns true/false, accordingly.

Algorithm 1: IsWellFormedRule Function

```
Require: R
Ensure: $TrueorFalse$
  if $\text{R}[\text{PROVIDES}] \neq \emptyset$ and $\text{R}[\text{SHARES}] = \emptyset$ then
    return $False$
  end if
  return $True$
```

Example 6 (Well formed security rule). *The security rule defined in Example 4 is an example of a well formed security rule.*

Example 7 (Malformed security rule). *Let us assume that we modify the security rule defined in Example 4, Table 4, and that we change both the* PROVIDES *and* REQUIRES *fields. Such modifications are shown in Table 6.*

Table 6. Security rule R_{MA}.

	Rule R_{MA}
D	D-LINK.933L
DOM	*
SHARES	-
PROVIDES	SETDAYNIGHT
REQUIRES	-

We would obtain a malformed security rule, according to Definition 5. In particular, rule $\mathrm{R_{MA}}$ claims that the device provides the SETDAYNIGHT service. However, $\mathrm{R_{MA}}$[SHARES] is empty, which violates Definition 5.

Ensuring that a security rule is well formed is necessary, but not sufficient, to achieve consistency among security rules. Intuitively, that is because two rules can be well formed with respect to themselves, but they could contradict each other. Thus, we want to capture the property that no rule concerning a device restricts another rule concerning the same device. Here restricts means that the rule provides less or the same number of services to the same set, or a subset, of devices. This concept of core rule will form the basis of the notion of consistent security contract in Section 5, where a set of rules represents the contract of a device.

Definition 6 (Core rule). *Given a set of security rules* SET = $\{\mathrm{R}_1, \ldots, \mathrm{R}_m\}$, $m>0$, *a rule* R $\in$ SET *is said to be core with respect to* SET, *if and only if the following condition holds:*

R $\in$ SET *is* core *iff* $\nexists$ $\mathrm{R}_\star \in$ SET: $(\mathrm{R}[\mathrm{D}] = \mathrm{R}_\star[\mathrm{D}]) \wedge (\mathrm{R}[\mathrm{DOM}] = \mathrm{R}_\star[\mathrm{DOM}]) \wedge \mathrm{R}_\star[\mathrm{SHARES}] \neq \emptyset \wedge (\mathrm{R}_\star[\mathrm{SHARES}] \subseteq \mathrm{R}[\mathrm{SHARES}]) \wedge (\mathrm{R}_\star[\mathrm{PROVIDES}] \subseteq \mathrm{R}[\mathrm{PROVIDES}])$.

In Algorithm 2, we provide a pseudo-algorithm that verifies if a rule is core and returns true/false, accordingly.

Algorithm 2: IsCoreRule Function

```
Require: R, SET
Ensure: TrueorFalse
  for $\forall$ $R_\star \neq R \in$ SET do
    if $R[D] = R_\star[D]$ and $R[DOM] = R_\star[DOM]$ then
      $provideFlag \leftarrow True$, $sharesFlag \leftarrow True$
      if $R_\star[SHARES] = \emptyset$ then
        $sharesFlag \leftarrow False$
      else
        for $\forall$ $D_\triangle \in R_\star[SHARES]$ do
          if $D_\triangle \notin R[SHARES]$ then
            $sharesFlag \leftarrow False$
          end if
        end for
      end if
      for $\forall$ $S_\triangle \in R_\star[PROVIDES]$ do
        if $S_\triangle \notin R[PROVIDES]$ then
          $provideFlag \leftarrow False$
        end if
      end for
      if $provideFlag = True$ and $sharesFlag = True$ then
        return $False$
      end if
    end if
  end for
  return $True$
```

5. S $\times$ C for IoT: Security Contract

In the previous section we described the building blocks of our formalization. Next, we need to define security contracts. Informally, a security contract of an IoT device is a non-empty set of security rules describing the security behaviour of the device. Formally:

Definition 7 (Security contract)**.** *A security contract (or simply contract)* $\mathrm{C_D}$ *of an IoT device* D *is a non-empty and non-ordered set of security rules concerning the device* D*, such that:*

$$\mathrm{C_D} = \{\mathrm{R_1}, \ldots, \mathrm{R_n}\},$$

where $n > 0$ *and* $\forall\, i \neq j$: $\mathrm{R_i}[D] = \mathrm{R_j}[D]$, $\mathrm{R_i} \neq \mathrm{R_j}$.

Example 8 (Security contract $\mathrm{C_A}$)**.** *An IoT camera surveillance enables Luke to enable/disable the camera LED from his smartphone over any domain, through the service* SETDAYNIGHT*. However, it enables him to access the administration panel only from within the network. For this to work, we need the* 933L *camera to have the contract in Table 7. Without considering security vulnerabilities resulting from vulnerable embedded software, this contract can easily prevent potential stalking episodes, which are more probable with currently manufactured IP cameras [12].*

Table 7. Security contract $\mathrm{C_A}$.

	Rule $\mathrm{R_{A1}}$	**Rule $\mathrm{R_{A3}}$**
D	D-LINK.933L	D-LINK.933L
DOM	*	LAN
SHARES	APPLE.LUKEPHONE	APPLE.LUKEPHONE
PROVIDES	SETDAYNIGHT	SETDAYNIGHT, ADMINPANEL
REQUIRES	-	-

Example 9 (Security contract $\mathrm{C_B}$)**.** *In this example, let us assume that* PHILIPS.HUEWHITE *exposes rule* $\mathrm{R_{B1}}$ *which provides to* PHILIPS *devices within the LAN domain three different services, namely* ON*,* BRI*, and* HUE*. Moreover, the device requires service* PHILIPS.HUEMOTION.PRESENCE *for switching on/off the light, based on the presence of people in the room. Last,* HUEWHITE *exposes a second rule* $\mathrm{R_{B2}}$*, which provides access to services* ON *and* BRI *to any device, within the LAN domain . A security rule that express these requirements is shown in Table 8.*

Table 8. Security contract $\mathrm{C_B}$.

	Rule $\mathrm{R_{B1}}$	**Rule $\mathrm{R_{B2}}$**
D	PHILIPS.HUEWHITE	PHILIPS.HUEWHITE
DOM	LAN	LAN
SHARES	PHILIPS.*	*.*
PROVIDES	ON, BRI, HUE	ON, BRI
REQUIRES	PHILIPS.HUEMOTION.PRESENCE	-

Within the context of the Ängen smarthome (described previously in Figure 1), let us suppose that the device PHILIPS.HUEWHITE is equipped with Contract $\mathrm{C_B}$. In Figure 3 we show the interactions that PHILIPS.HUEWHITE would perform with the other devices, according to the behaviour defined in its contract. In particular, the figure shows that the only device that benefits from service PHILIPS.HUEWHITE.HUE is PHILIPS.HUEMOTION.

The next key concept we want to express in our framework is the consistent contract concept. Consistency is seen here as a quality check of a contract. First, we require all the rules in a consistent contract to be well formed. Second, we want that all the rules in the contract are core, to be sure that no rule in the contract restricts another rule in the same contract. As a result, a consistent contract is a set of well formed and core rules.

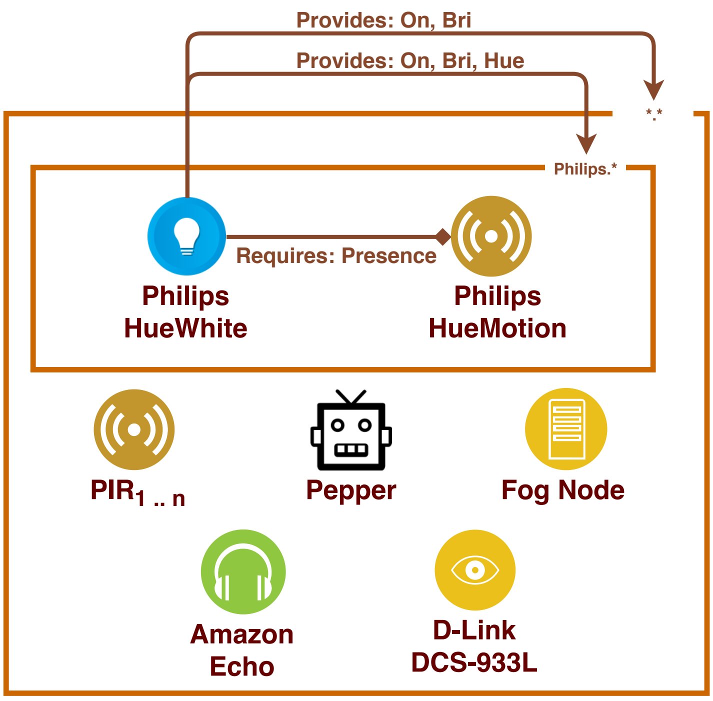

Figure 3. Security contract C_B. PHILIPS.HUEWHITE shares services ON and BRI with all the devices in the LAN (e.g., with Amazon Echo). Service HUE is shared only with PHILIPS devices, which means that in this scenario only PHILIPS.HUEMOTION is allowed to use it. Moreover, PHILIPS.HUEWHITE requires from PHILIPS.HUEMOTION the service PRESENCE.

Definition 8 (Consistent security contract). *A security contract* C_D *of a device* D *is said to be* consistent *if and only if the following two conditions hold:*

1. $\forall$ R $\in C_D$, R *is well formed*
2. $\forall$ R $\in C_D$, R *is core in* C_D

In Algorithm 3, we provide a pseudo-algorithm that verifies if a contract is consistent and returns true/false accordingly.

Algorithm 3: IsConsistentContract Function

```
Require: $C_D$
Ensure: $TrueorFalse$
  for $\forall$ R $\in C_D$ do
    if IsWellFormedRule(R) $= False$ then
      return $False$
    end if
  end for
  for $\forall$ R $\in C_D$ do
    if IsCoreRule(R, $C_D$) $= False$ then
      return $False$
    end if
  end for
  return $True$
```

Example 10 (Consistent security contract C_B). *The security contract described in Example 9 is an example of a consistent security contract, with respect to Definition 8.*

Example 11 (Inconsistent security contract C_{IB})**.** *Let us assume that we modify the security contract described in Example 9. In particular, let us remove both* BRI *and* HUE *from rule* R_{B1} *(obtaining rule* R_{B3}*). The contract shown in Table 9 formalizes our goal.*

Table 9. Security contract C_{IB}.

	Rule R_{B3}	**Rule R_{B2}**
D	PHILIPS.HUEWHITE	PHILIPS.HUEWHITE
DOM	LAN	LAN
SHARES	PHILIPS.*	*.*
PROVIDES	ON	ON, BRI
REQUIRES	PHILIPS.HUEMOTION.PRESENCE	-

In this case, the resulting contract C_{IB} *contradicts Definition 8, because rule* R_{B3} *restricts rule* R_{B2}*. In particular, less services are provided to less devices in the same domain (i.e.,* R_{B3} [PROVIDES] $\subseteq$ R_{B2}[PROVIDES] $\wedge$ R_{B3}[SHARES] $\subseteq$ R_{B2}[SHARES]*).*

6. S × C for IoT: Security Policy

Building on top of the previous sections, in this section we describe security policies. Informally, a security policy of a Fog node is a non-empty set of security rules describing the allowed security behaviour of the devices for which the Fog node is responsible. Formally:

Definition 9 (Security policy)**.** *A security policy (or simply policy)* P_F *of a Fog node* F *is a non-empty and non-ordered set of security rules, such that:*

$$P_F = \{R_1, \ldots, R_m\}, \text{where } m > 0 \text{ and } R_i \neq R_j, i \neq j.$$

From a practical perspective, the security rules within a policy P_F can derive from two different sources: The contracts C_{D_1}, ..., C_{D_n} of the IoT devices the Fog node F is responsible for, and a number of additional contracts $C^A_{D_1}$, ..., $C^A_{D_p}$ defined by the administrator A, which apply to the respective devices D_1, ..., D_p. This set of additional contracts can be empty, in case the administrator does not define any specific security rule. As a result, a policy of a Fog node F can be seen as union of the contracts of the IoT devices and the contracts defined by the administrator of the Fog node.

$$P_F = C_{D_1}, ..., C_{D_n}, C^A_{D_1}, ..., C^A_{D_p}$$

Example 12 (Security policy P_A)**.** *In the example we show in Table 10, a policy is a composition of three different rules, one from device* D-LINK.933L*, and two from device* PHILIPS.HUEWHITE*. It is worth noticing that a security policy can have rules for different devices, as long as these rules do not contradict each other (we will better clarify this point in the rest of the section).*

Table 10. Security policy P_A.

	Rule R_{A1}	**Rule R_{B1}**	Rule R_{B2}
D	D-LINK.933L	PHILIPS.HUEWHITE	PHILIPS.HUEWHITE
DOM	*	LAN	LAN
SHARES	APPLE.LUKEPHONE	PHILIPS.*	*.*
PROVIDES	SETDAYNIGHT	ON, BRI, HUE	ON, BRI
REQUIRES	-	PHILIPS.HUEMOTION.PRESENCE	-

Example 13 (Security policy $\mathrm{P_B}$)**.** *In this second example, shown in Table 11, let us suppose that the administrator wants to add a rule to his security policy. This rule (namely, rule* $\mathrm{R_{Admin1}}$*) allows every device from manufacturer* PHILIPS *to use service* D-LINK.933L.SETDAYNIGHT *over the Internet.*

Table 11. Security policy $\mathrm{P_B}$.

	Rule $\mathrm{R_{A1}}$	**Rule $\mathrm{R_{B2}}$**	**Rule $\mathrm{R_{Admin1}}$**
D	D-LINK.933L	PHILIPS.HUEWHITE	PHILIPS.*
DOM	*	LAN	Internet
SHARES	APPLE.LUKEPHONE	*.*	-
PROVIDES	SETDAYNIGHT	ON, BRI	-
REQUIRES	-	-	D-LINK.933L.SETDAYNIGHT

Analogously to the security contract case, what we want to capture in our S × C framework is the concept of consistent security policy. Since a policy includes rules of contracts from different devices, the notion of consistency given in the contract case is not enough. This is because the contracts of two different devices might lead to a leak of information (or illegal information exchange). This is the case of a device $\mathrm{D_1}$ that (1) requires some services provided by a device $\mathrm{D_2}$, but (2) $\mathrm{D_2}$ cannot share these services to $\mathrm{D_1}$ because it would lead to information leakage (due to the fact that $\mathrm{D_1}$ can communicate with devices that $\mathrm{D_2}$ refuses to communicate with). In the following, we formalise (1) by means of the concept of direct communication among two IoT devices, and the consequent concept of allowed information flow. Then, we formalise (2) by means of the concept of forbidden information flow. We will then combine these concepts to define the notion of illegal information exchange, which is what we want to avoid in a (consistent) policy.

Definition 10 (Direct Communication)**.** *Given a device* $\mathrm{D_1}$ *with consistent contract* $\mathrm{C_{D_1}}$ *and a device* $\mathrm{D_2}$ *with consistent contract* $\mathrm{C_{D_2}}$*, we say that* $\mathrm{D_1}$ *directly communicates with* $\mathrm{D_2}$*, denoted* $\mathrm{D_1} \rightsquigarrow \mathrm{D_2}$*, if* $\exists\, \mathrm{R_\star} \in \mathrm{C_{D_1}}, \mathrm{R_\circ} \in \mathrm{C_{D_2}}$: $\mathrm{R_\star}$*[*REQUIRES*]* $\cap$ $\mathrm{R_\circ}$*[*PROVIDES*]* $\neq \emptyset$*.*

In Algorithm 4, we provide a pseudo-algorithm that verifies if two devices directly communicate.

Definition 11 (Allowed direct communication)**.** *Given a device* $\mathrm{D_1}$ *with consistent contract* $\mathrm{C_{D_1}}$ *and a device* $\mathrm{D_2}$ *with consistent contract* $\mathrm{C_{D_2}}$*, such that* $\mathrm{D_1} \rightsquigarrow \mathrm{D_2}$*, we say that* $\mathrm{D_1}$ *is allowed to directly communicate with* $\mathrm{D_2}$*, denoted* $\mathrm{D_1} \rightarrow \mathrm{D_2}$*, if* $\forall\, \mathrm{R_\star} \in \mathrm{C_{D_1}}, \mathrm{R_\circ} \in \mathrm{C_{D_2}}$: $\mathrm{R_\star}$*[*REQUIRES*]* $\cap$ $\mathrm{R_\circ}$*[*PROVIDES*]* $\neq \emptyset$*, we have* $\mathrm{D_1} \in \mathrm{R_\circ}$*[*SHARES*].*

Algorithm 4: DirectCommunication Function

```
  Require: $\mathrm{C_{D_1}}, \mathrm{C_{D_2}}$
  Ensure: $TrueorFalse$
    for $\forall\, \mathrm{R_\star} \in \mathrm{C_{D_1}}$ do
      for $\forall\, \mathrm{R_\circ} \in \mathrm{C_{D_2}}$ do
        if $\mathrm{R_\star}[\mathrm{DOM}] \cap \mathrm{R_\circ}[\mathrm{DOM}]$
        and $\mathrm{R_\star}[\mathrm{REQUIRES}] \cap \mathrm{R_\circ}[\mathrm{PROVIDES}] \neq \emptyset$ then
          return $True$
        end if
      end for
    end for
    return $False$
```

In Algorithm 5, we provide a pseudo-algorithm that verifies if two devices are allowed to directly communicate.

Definition 12 (Allowed information flow). *Given a device* D_1 *with consistent contract* C_{D_1} *and a device* D_2 *with consistent contract* C_{D_2}*, such that* $D_1 \rightarrow D_2$*, there is an allowed information flow between* D_1 *and* D_2*, denoted* $D_1 \overrightarrow{ok} D_2$*, if* $\forall\, R_\star \in C_{D_1}, R_\circ \in C_{D_2}$*:* $R_\star[\textsc{Requires}] \cap R_\circ[\textsc{Provides}] \neq \emptyset$*, we have* $R_\star[\textsc{Shares}] \subseteq R_\circ[\textsc{Shares}]$*.*

Algorithm 5: AllowedDirectCommunication Function

```
Require: $C_{D_1}$, $C_{D_2}$
Ensure: $True or False$
  if DirectCommunication($C_{D_1}$, $C_{D_2}$) = $False$ then
    return $False$
  end if
  for $\forall R_\star \in C_{D_1}$ do
    for $\forall R_\circ \in C_{D_2}$ do
      if $R_\star[\textsc{Dom}] \cap R_\circ[\textsc{Dom}]$
      and $D_1 \notin R_\circ[\textsc{Shares}]$
      and $R_\star[\textsc{Requires}] \cap R_\circ[\textsc{Provides}] \neq \emptyset$ then
        return $False$
      end if
    end for
  end for
  return $True$
```

Definition 13 (Forbidden information flow). *Given a device* D_1 *with consistent contract* C_{D_1} *and a device* D_2 *with consistent contract* C_{D_2}*, such that* $D_1 \rightarrow D_2$*, there is a forbidden information flow between* D_1 *and* D_2*, denoted* $D_1 \overrightarrow{no} D_2$*, if* $\exists\, R_\star \in C_{D_1}, R_\circ \in C_{D_2}$*:* $R_\star[\textsc{Requires}] \cap R_\circ[\textsc{Provides}] \neq \emptyset \wedge R_\star[\textsc{Shares}] \nsubseteq R_\circ[\textsc{Shares}]$*.*

In Algorithm 6, we provide a pseudo-algorithm that verifies if a forbidden information flow exists between two devices.

Algorithm 6: ForbiddenInformationFlow Function

```
Require: $C_{D_1}$, $C_{D_2}$
Ensure: $True or False$
  if AllowedDirectCommunication($C_{D_1}$, $C_{D_2}$) = $False$ then
    return $False$
  end if
  for $\forall R_\star \in C_{D_1}$ do
    for $\forall R_\circ \in C_{D_2}$ do
      if $R_\star[\textsc{Dom}] \cap R_\circ[\textsc{Dom}]$
      and $R_\star[\textsc{Shares}] \nsubseteq R_\circ[\textsc{Shares}]$
      and $R_\star[\textsc{Requires}] \cap R_\circ[\textsc{Provides}] \neq \emptyset$ then
        return $True$
      end if
    end for
  end for
  return $False$
```

Definition 14 (Illegal information exchange). *Given a consistent contract* C_{D_1} *of a device* D_1 *and a security policy* P_F *of a Fog node* F*, there is an illegal information exchange, denoted* $C_{D_1} \nRightarrow P_F$*, if there exists a contract* $C_{D_2} \in P_F$ *such that* $(D_1 \overrightarrow{no} D_2) \vee (D_2 \overrightarrow{no} D_1)$*.*

In Algorithm 7, we provide a pseudo-algorithm that verifies if two devices generate an illegal information exchange.

Algorithm 7: IllegalInformationExchange Function

```
Require: $C_{D_1}$, $P_F$
Ensure: $True or False$
  for $\forall C_{D_2} \in P_F$ do
    if ForbiddenInformationFlow($C_{D_1}$, $C_{D_2}$) or
    ForbiddenInformationFlow($C_{D_2}$, $C_{D_1}$) then
      return True
    end if
  end for
  return False
```

Example 14 (Illegal information exchange). *For the sake of simplicity, in Table 12 we suppose that* C_{D_1} *and* $C_{D_2} \in P_F$ *are composed of a single rule, respectively.*

According to Definition 14, this scenario would cause an illegal information exchange. First, we can verify that C_{D_2}*[REQUIRES]* $\cap$ C_{D_1}*[PROVIDES]* $= \emptyset$*, therefore there is no allowed direct communication* $D_2 \rightarrow D_1$ *and the second condition of the illegal information exchange (*$D_2 \overrightarrow{no} D_1$*) can be ignored.*

Second, we can verify that $D_1 \rightarrow D_2$*, since that* C_{D_1}*[REQUIRES]* $\cap$ C_{D_2}*[PROVIDES]* $=$ ON *(therefore,* $\neq \emptyset$*), and that* D-LINK.933L $\in C_{D_2}$*[SHARES]. However,* C_{D_1}*[SHARES] (= {*APPLE.LUKEPHONE*})* $\not\subseteq$ C_{D_2}*[SHARES] (= {*D-LINK.933L, PHILIPS.HUEWHITE*}). Therefore, we have* $D_1 \overrightarrow{no} D_2$ *and an illegal information exchange.*

Looking at Figure 4, the outgoing arrows for D-LINK.933L *and* PHILIPS.HUEMOTION *clearly show that it an illegal information exchange exists between the two devices. In particular,* D-LINK.933L *shares a set of services with* PHILIPS.HUEMOTION *and* PHILIPS. HUEMOTION *shares another set with* APPLE.LUKEPHONE. *However,* D-LINK.933L *does not share any service with* APPLE.LUKEPHONE *and this flow might leak unintended information. Therefore, an illegal information exchange error is raised here.*

Table 12. Illegal information exchange $C_{D_1} \not\Rightarrow P_F$.

	C_{D_1}	$C_{D_2} \in P_F$
D	D-LINK.933L	PHILIPS.HUEMOTION
DOM	LAN	LAN
SHARES	APPLE.LUKEPHONE	D-LINK.933L, PHILIPS.HUEWHITE
PROVIDES	SETDAYNIGHT	ON, BRI
REQUIRES	PHILIPS.HUEMOTION.ON	-

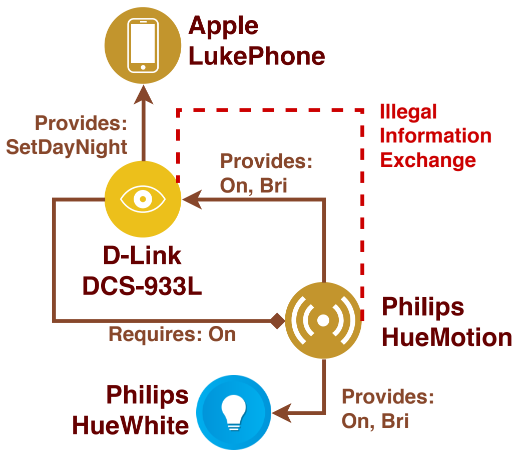

Figure 4. Illegal information exchange $C_{D_1} \not\Rightarrow P_F$. D-LINK.933L does not want to share any service with APPLE.LUKEPHONE (i.e., there are no outgoing arrows), but PHILIPS.HUEMOTION can potentially act as a bridge between the two devices. This might lead to critical information leak, therefore it has to be prevented.

Definition 15 (Consistent security policy)**.** *A security policy* $\mathrm{P_F}$ *of a Fog node* F *is said to be consistent if and only if the following conditions hold:*

1. $\forall\ \mathrm{R} \in \mathrm{P_F}$, R *is well formed*
2. $\forall\ \mathrm{R} \in \mathrm{P_F}$, R *is core in* $\mathrm{P_F}$
3. $\nexists\ \mathrm{C_D} \in \mathrm{P_F}, \mathrm{C_D} \nRightarrow\ \mathrm{P_F}$
4. $\nexists\ \mathrm{C_D^A} \in \mathrm{P_F}, \mathrm{C_D^A} \nRightarrow\ \mathrm{P_F}$

In Algorithm 8, we provide a pseudo-algorithm that verifies if a policy is consistent.

Example 15 (Consistent security policy $\mathrm{P_{IA}}$)**.** *The security policy described in Example 12 is an example of a consistent security policy, with respect to our own Definition 15.*

Algorithm 8: IsConsistentPolicy Function

```
Require: $\mathrm{P_F}$
Ensure: $TrueorFalse$
  for $\forall\ \mathrm{R} \in \mathrm{P_F}$ do
    if IsWellFormedRule(R) $= False$ then
      return $False$
    end if
  end for
  for $\forall\ \mathrm{R} \in \mathrm{P_F}$ do
    if IsCoreRule(R, $\mathrm{P_F}$) $= False$ then
      return $False$
    end if
  end for
  for $\forall\ \mathrm{C_D} \in \mathrm{P_F}$ do
    if IllegalInformationExchange($\mathrm{C_D}$, $\mathrm{P_F}$) then
      return $False$
    end if
  end for
  for $\forall\ \mathrm{C_D^A} \in \mathrm{P_F}$ do
    if IllegalInformationExchange($\mathrm{C_D^A}$, $\mathrm{P_F}$) then
      return $False$
    end if
  end for
  return $True$
```

Example 16 (Inconsistent security policy $\mathrm{P_{IB}}$)**.** *Let us suppose that the administrator modifies the policy previously defined in Example 13. In particular, let us suppose that the administrator substitutes rule* $\mathrm{R_{Admin1}}$ *with rule* $\mathrm{R_{Admin2}}$*, enabling* PHILIPS.HUEWHITE *to provide every* PHILIPS *device three services, as shown in Table 13. In this case, the policy would not be consistent, since rule* $\mathrm{R_{B2}}$ *would restrict Rule* $\mathrm{R_{Admin2}}$*. In particular, rule* $\mathrm{R_{Admin2}}$ *would not be core, violating the fourth condition defined in Definition 15.*

Table 13. Inconsistent security policy $\mathrm{P_{IB}}$.

	Rule $\mathrm{R_{A1}}$	**Rule $\mathrm{R_{B2}}$**	**Rule $\mathrm{R_{Admin2}}$**
D	D-LINK.933L	PHILIPS.HUEWHITE	PHILIPS.HUEWHITE
DOM	*	LAN	LAN
SHARES	APPLE.LUKEPHONE	*.*	*.*
PROVIDES	SETDAYNIGHT	ON, BRI	ON, BRI, HUE
REQUIRES	-	-	-

The last concept we have to define concerns the matching between a contract of a device and the policy of a Fog node. This is the core idea behind the $S \times C$ approach: A device is accepted in a network of devices governed by a Fog node if and only if the contract of the device matches the policy of the Fog node.

Definition 16 (Contract–policy matching)**.** *Given a contract* C_D *of a device* D *and a consistent policy* P_F *of a Fog node* F*, we say that* C_D *matches* P_F *if* $P'_F = P_F \cup C_D$ *is consistent.*

In Algorithm 9 we show the pseudo-code for our matching algorithm. All the rules from the contract C_D are added one by one to a temporary policy P'_F. Then, the consistency of P'_F is verified and the match can return either true or false.

Algorithm 9: Matching Function

```
Require: $P_F$, $C_D$
Ensure: $TrueorFalse$
  $P'_F \leftarrow P_F$
  for $\forall$ R $\in C_D$ do
    $P'_F \leftarrow P'_F + R$
  end for
  if IsConsistentPolicy($P'_F$) = $True$ then
    return $True$
  end if
  return $False$
```

7. Jumping the Air Gap

Agadakos et al. [13] have recently highlighted that one of the key issues with the IoT is that security is solely considered on device and device-to-device levels. They claim that, given the complexity of interactions that happen in the IoT, the approach should be much more holistic. We totally agree with their view.

In this section, we show that our approach can nicely model the case studies proposed by Agadakos et al., as well as prevent the issue of unexpected chains of events they depicted.

7.1. Hidden Paths

In this scenario, there are communication paths, fully or partially composed of physical interactions that are hidden. As an example, Agadakos et al. consider a typical smart living room equipped with an Amazon Echo device and a Philips Smart TV, and assume that no direct cyber channels exist between the two, as shown in Figure 5. However, since the smart TV produces sound outputs, and the Amazon Echo does not have any protection over its input vocal channel (such as vocal identification), a hidden communication path exists between the two devices.

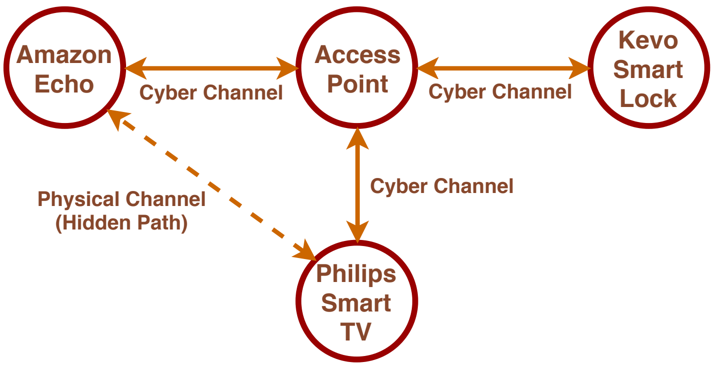

Figure 5. The topological map for the use case defined by [13].

Example 17 (Security contract C_{Echo1}). *In our proposal, a hypothetical contract carried by Amazon Echo would explicitly state that the device has a vocal service (as an example, let us call it VocalInput) which allows anyone to communicate with it, within the LAN domain. Enforcing explicit declarations about services, as shown in Table 14, would help to identify potential hidden paths, if not to avoid them completely.*

Table 14. Security contract C_{Echo1}.

	Rule R_{E1}
D	AMAZON.ECHO
DOM	LAN
SHARES	*.*
PROVIDES	VOCALINPUT
REQUIRES	-

Another issue that has been raised in the past about vocal assistants is that the vocal perimeter can easily extend outside the smart home (https://techcrunch.com/2017/05/23/alexa-dont-talk-to-strangers/). A person walking by an open window can easily talk out loud and activate a vocal assistant placed in the living room. This situation can be easily captured by S × C contracts, as shown in the following Example 18.

Example 18 (Security contract C_{Echo2}). *In order to model an extended vocal perimeter, a contract would simply have to state that the VocalInput function is reachable by anyone in any domain, meaning that anyone outside the LAN (human being or device) can activate the Amazon Echo without belonging to the same LAN. An example of this contract is shown in Table 15.*

Table 15. Security contract C_{Echo2}.

	Rule R_{E2}
D	AMAZON.ECHO
DOM	*
SHARES	*.*
PROVIDES	VOCALINPUT
REQUIRES	-

The previous explicit contracts, namely C_{Echo1} and C_{Echo2}, would help the administrator to have a grasp of potential attack vectors introduced by new devices and to setup proper policies accordingly.

Example 19 (Security policy P_{FR}). *Let us suppose that a Federal Reserve building has in place policies that impede any kind of communication with the outside world, for safety reasons. In particular, the network administrator wants to avoid that a naive employee takes his brand new Amazon Echo to the office and plugs it in the network, exposing the building to external attacks. As we show in Table 16, with S × C it would be easy to avoid this scenario. Assuming that R_{FR1} is already in place, when the Amazon Echo device is plugged in and shows its Contract C_{Echo2}, Rule R_{E2} is evaluated non-core within P_{FR} (since it is restricted by rule R_{FR1}), and the Amazon Echo is not accepted. For the sake of clarity, in this example we wrote only the relevant rules for this discussion, but more rules could be part of this policy, as shown in previous sections.*

Table 16. Inconsistent security policy P_{FR}.

	Rule $\mathrm{R}_{\mathsf{FR1}}$	Rule R_{E2}
D	AMAZON.ECHO	AMAZON.ECHO
DOM	*	*
SHARES	*.*	*.*
PROVIDES	-	VOCALINPUT
REQUIRES	-	-

7.2. Security Degradation

Agadakos et al. identified another problem arising from combining IoT devices, namely the security degradation problem. This problem, similar to the confused deputy problem, states that a device which receives inputs over unauthenticated channels, but sends outputs over authenticated ones, can be tricked by malicious attackers into performing actions in their stead.

Example 20 (Security degradation as an illegal information exchange)**.** *In terms of our work, this problem is an instance of an illegal information flow. A Samsung Hub requires a service OpenClose from a Samsung Window Sensor, in order to alarm the user if the window is open. The Hub is powered by an OORT Smartplug, therefore the Hub contract declares that it shares its OnOff service with the Smartplug. Last, the Smartplug receives unauthenticated messages over a BLE channel so its contract states two things: That it shares all its services with anyone in the LAN domain, and that it requires the Samsung Hub OnOff service (to make explicit that it is capable of turning on/off the Hub). The resulting (inconsistent) security policy is shown in Table 17.*

Table 17. Inconsistent security policy $\mathrm{P}_{\mathsf{OORT}}$.

	Rule $\mathrm{R}_{\mathsf{Plug}}$	Rule $\mathrm{R}_{\mathsf{Hub}}$	Rule $\mathrm{R}_{\mathsf{Sensor}}$
D	OORT.PLUG	SAMSUNG.HUB	SAMSUNG.SENSOR
DOM	LAN	LAN	LAN
SHARES	*.*	OORT.PLUG	SAMSUNG.HUB
PROVIDES	*	ONOFF	OPENCLOSE
REQUIRES	SAMSUNG.HUB.ONOFF	SAMSUNG.SENSOR.OPENCLOSE	-

In particular, the reason why this policy is inconsistent is because there is an illegal information exchange $\mathrm{D}_{\mathsf{Hub}} \overrightarrow{no} \mathrm{D}_{\mathsf{Sensor}}$*, according to Definition 14.*

First, there is a $\mathrm{D}_{\mathsf{Hub}} \to \mathrm{D}_{\mathsf{Sensor}}$*, since* $\mathrm{C}_{\mathrm{D}_{\mathsf{Hub}}}[\mathrm{REQUIRES}] \cap \mathrm{C}_{\mathrm{D}_{\mathsf{Sensor}}}[\mathrm{PROVIDES}] = \mathrm{OPENCLOSE}$ *(i.e.,* $\neq \emptyset$*), and* $\mathrm{SAMSUNG.HUB} \in \mathrm{C}_{\mathrm{D}_{\mathsf{Sensor}}}[\mathrm{SHARES}]$*. However,* $\mathrm{D}_{\mathsf{Hub}} \overrightarrow{no} \mathrm{D}_{\mathsf{Sensor}}$ *is equally true. In particular,* $\mathrm{C}_{\mathrm{D}_{\mathsf{Hub}}}[\mathrm{SHARES}]$ *(=* $\{\mathrm{OORT.PLUG}\}$*)* $\nsubseteq \mathrm{C}_{\mathrm{D}_{\mathsf{Sensor}}}[\mathrm{SHARES}]$ *(=* $\{\mathrm{SAMSUNG.HUB}\}$*), which entails an illegal information exchange* $\mathrm{D}_{\mathsf{Hub}} \overrightarrow{no} \mathrm{D}_{\mathsf{Sensor}}$*.*

7.3. Transitions and States

As a last use case, in [13] the authors show that with their approach it is possible to model transitions and states, allowing them to identify security violations over time. In their example, there is a Roomba Vacuum that should never leave the house, and an OORT Smart Lock capable of closing/opening a garage door to the outside. Similar to Section 7.2, this issue is fuelled by the fact that the smart lock is allowed to communicate over BLE with any device, and that this is not formally declared to the network. However, if the OORT lock shows a contract similar to $\mathrm{C}_{\mathsf{OORTLock}}$ in Table 18, the administrator can have a clear view about the device characteristics, and a strong policy can easily refuse this device to join the network.

Table 18. Security contract $C_{OORTLock}$.

	Rule R_{L1}
D	OORT.LOCK
DOM	*
SHARES	*.*
PROVIDES	OPENCLOSE
REQUIRES	-

As we have previously stated in Section 7.1, enforcing devices to declare required and provided services considerably strengthens a network: Explicit behavioural descriptions allow for a much easier holistic view on the chain of events that can potentially happen in a complex IoT environment.

8. Threat Model

In this work, we have focused on two main threats affecting nowadays IoT systems (Zhou et al. [2]): Insufficient security configurability and, partially, insufficient security configurations. Of course, many other weaknesses can arise from IoT networks and it is out of the scope of this article to address all of them. As an example, with regards to the threats listed in Section 2, we do not address insecure network communications, nor vulnerable cloud/web services. Moreover, our solution partially covers other problems of the same list, as a side effect. Namely, we mitigate privacy leaks through the concept of illegal information exchange, and we mitigate vulnerable firmwares by means of signed proofs of compliance (PoCs).

In order to have a better picture about the robustness of the S × C approach, in Figure 6 we present a threat model. In the picture we highlight the attacks that could be performed on the main actors of a Fog-based IoT system. The reader will notice that our threat model does not list each and every possible attack. We focus only on the most well known and disruptive ones. In the remainder of this Section, we discuss the specific threats and the system resilience with/without S × C.

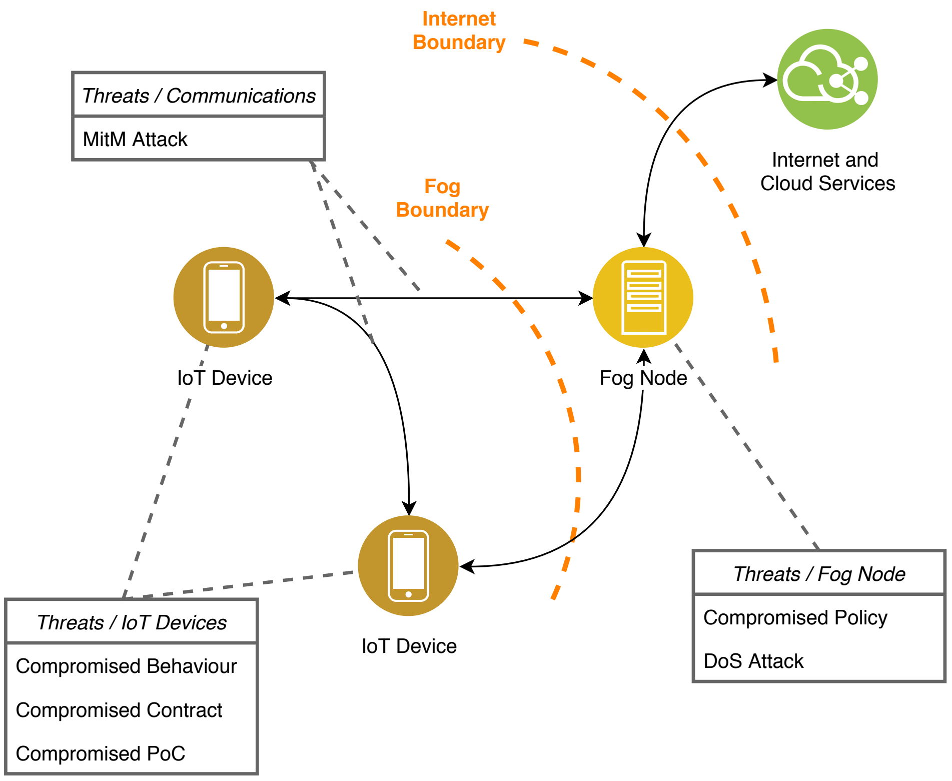

Figure 6. Graphical representation of the architecture threat model.

8.1. Threats to IoT Devices

Let us start from the threats to the IoT devices shown in Figure 6. As noted before, IoT devices currently sold on the market do not carry behavioural descriptions. Therefore, it is actually not clear and not simple to find out whether an IoT device has been tampered with or not.

With S $\times$ C, we equip IoT devices with a contract and a proof-of-compliance (PoC). The PoC is a proof, signed by the manufacturer and stored in the IoT device, which binds the onboard code to the contract. The PoC must be formally verifiable through an external validator. If the code or the contract change, due to a software update, the manufacturer has to issue and sign a new PoC. Thus, not only a malicious actor must create a new contract, in order to tamper with a device. He must also be able to forge a new PoC and sign it with the manufacturer credentials. If the PoC matches both the code and the contract, we say that the triplet <Code, Contract, PoC> is consistent. If the three elements do not match, the triplet is inconsistent.

Let us assume that a malicious actor tampers with the contract, but not with the PoC nor the code. As mentioned before, the PoC would not match with the contract and the code, thus the triplet would not be consistent. At installation time, the device can be rejected, but at runtime the Fog Node monitors only the device behaviour. Therefore, the Fog node can discover that the contract has changed only when the device behaves differently from the contract, stored in the policy. As soon as an unexpected behaviour emerges, the Fog node acts as if the contract have been updated and checks the triplet again. In this case, the Fog node finds out that the triplet is inconsistent and rejects the device.

As an example, if an attacker breaches into the Amazon Echo described in Table 14 and removes the service VOCALINPUT, the triplet is not consistent any more. At installation time the Fog node rejects the IoT device or notifies the inconsistency to the administrator. At runtime the Fog node does not realize the breach, until the device behaves unexpectedly.

The situation is similar if the attacker tampers with the PoC, but does not modify the code nor the contract. At installation time the triples is inconsistent, so the device is rejected right away. If the PoC changes at runtime, the Fog node does not discover it, but no problems arise since that the device still behaves as expected (i.e., code and contract are unaltered). Legacy IoT devices do not carry contracts nor PoCs, therefore the aforementioned scenarios only apply to S $\times$ C.

Now, suppose that a malicious actor tampers with the code, but not with the PoC nor the contract. Again, this would create an inconsistent triplet. The Fog node would notice at runtime the discrepancy as soon as the tampered-with device behaves differently from its contract. However, a non-S $\times$ C architecture cannot identify this security threat, and it would allow the IoT device to join the network.

Last, suppose that a malicious attacker manipulates the entire triplet, so that it looks consistent. It is worth highlighting that the PoC must be signed by the manufacturer, hence forging a PoC is not an easy task. Without S $\times$ C the IoT device would join the network, and it would be easy for the attacker to perform malicious actions. Keeping in mind that the Fog node performs real-time monitoring, with S $\times$ C we have two different scenarios. In cases where the IoT device violates at run-time the network policy, the Fog node can identify the inconsistency and react. On the other hand, if the code has been tampered with in a way that it still complies with the policy, S $\times$ C cannot detect the breach.

In Table 19 we summarise these concepts, showing the benefits of S $\times$ C as well as this limitation.

Table 19. Possible attacks on IoT devices (with vs without S × C).

Action	Without S × C	With S × C
Compromised contract; Unaltered code; Unaltered PoC	Do not apply, no contract exists -	Fog node rejects device at installation time ✓ Fog node discovers the breach at runtime as soon as the device behaves unexpectedly ✓
Compromised PoC; Unaltered code; Unaltered contract	Do not apply, no PoC exists -	Fog node rejects device at installation time ✓
Compromised code; Unaltered contract; Unaltered PoC	Fog node has no way to detect unintended behaviours. Device is erroneously accepted ✗	Fog node detects at runtime the breach as soon as the device behaves in a way that violates the policy ✓
Behaviour violating policy; Matching code, contract, and PoC	Fog node has no way to detect unintended behaviours. Device is erroneously accepted ✗	Fog node detects at runtime that the behaviour does not match the network policy. Device is rejected ✓
Behaviour complying with policy; Matching code, contract, and PoC	Fog node has no way to detect unintended behaviours. Device is erroneously accepted ✗	Fog node cannot detect at runtime that the behaviour does not match the network policy. Device is erroneously accepted ✗

We can better understand the last row of the S × C column in Table 19, if we take another point of view. IoT devices can receive updates over their lifecycle and their behaviour can change. This means we cannot reject IoT devices on the basis of inconsistency between current and past behaviour. We can only rely upon the triplet. Therefore, if the triplet is consistent and the PoC looks genuine, we cannot decide if the device underwent an honest update or a well-executed violation. Referring again to the contract in Table 14, let us assume that the attacker removes VocalInput and forges a consistent triplet <Behaviour, Contract, PoC>. In this case, there is no way to decide whether the contract modification was genuine or not.

Other threats might arise from misusing services, while formally complying with the security policy. For the sake of argument, we take the policy described in Table 17 and we show a consistent alternative in Table 20.

Table 20. Consistent security policy P_{OORTCons}.

	Rule R_{Plug}	Rule R_{Hub}	Rule R_{Sensor}
D	OORT.Plug	Samsung.Hub	Samsung.Sensor
Dom	LAN	LAN	LAN
Shares	*.*	OORT.Plug	Samsung.Hub, OORT.Plug
Provides	*	OnOff	OpenClose
Requires	Samsung.Hub.OnOff	Samsung.Sensor.OpenClose	-

Let us suppose that an attacker violates Samsung.Hub. Without changing anything in the device, the attacker issues hundreds of commands to Samsung.Sensors.OpenClose. This is perfectly fine with the network security policy. However, this behaviour is clearly not as intended, and it might lead to serious damages to Samsung.Sensor. The problem resides in the very nature of declarative security approaches. If the framework is not powerful enough to describe fine-grained behaviour, policies cannot mark allowed actions nor forbidden ones.

8.2. Threats to Fog Node and Communication Channels

It is worth mentioning that our solution introduces some potential threats as a side effect. The main culprit is the Fog node. As for every centralized solution, single points of trust translate into single points of failure. When we entrust a Fog node with storing the network policy, we centralize critical tasks in a single actor. As a consequence, the Fog node is going to be a likely target for a malicious attacker.

Three main issues could arise. First, an attacker could aim to breach the Fog node and manipulate the policy. As an example, he could cut communications between two critical devices by means of a restrictive policy. On the opposite spectrum, the attacker might try to allow flows the administrator wanted to impede. In this case, the attacker can be more permissive than the administrator but this will not override the devices contracts. Thus, the attacker cannot force the devices to act differently from their intended behaviour. Third, the same attacker might decide to perform a DoS attack, aiming to disrupt the entire network.

In its current form, the S $\times$ C architecture we designed is prone to such attacks. However, there are a couple of potential solutions. One solution is replication, a common strategy for fault-tolerant systems. Indeed, introducing multiple instances of a Fog node within the network, we can mitigate some risks. Another solution would be to implement fallback routines in the IoT devices, and allow them to maintain communications over machine-to-machine (M2M) channels. In this scenario, the devices would bypass the compromised Fog node and negotiate directly, according to their own contracts.

Last, but not least, we would like to mention that a multitude of man-in-the-middle (MitM) attacks are possible on any networked systems. For the sake of simplicity, in our paper we assumed the channels to be secure and tamper-proof. However, it is good to keep in mind that many dangers can come from insecure communication flows. In Table 21, we summarise the issues introduced by a single Fog node.

Table 21. Possible attacks on fog node and communication channels (with vs without S $\times$ C).

Action	Without S $\times$ C	With S $\times$ C
Compromising policy	Do not apply, Fog node is not in charge of the security policy -	Compromised policy, but devices' contracts and codes are unaltered. Attacker can only impede communications through a restricting policy. A more permissive policy cannot cause unexpected information leaks, devices still respect their own contracts ~
DoS attack on Fog node	Do not apply, Fog node is not in charge of the security policy -	Compromised system, but devices' contracts and codes are unaltered. Devices can potentially negotiate M2M and keep working according to their contracts ~
Man-in-the-Middle (MitM) attack	Attacker can steal and alter data ✗	Attacker can steal and alter data ✗

9. Related Work

In recent years, IoT security has been largely discussed. Matheu-Garcia et al. [14] highlighted that manufacturers should be included in the loop, in order to create more resilient devices. The authors propose a certification methodology that delivers a measurable evaluation of IoT devices security, as well as an automatic security assessment. Moreover, they noted the lack of an IoT vulnerability database, which would enable better automatic security tests. Once these problems are amended, manufacturers could include in contracts useful information about compliance to security standards. As an example, a contract could include the last time the device software was verified against the vulnerability database, or the security level assigned by the automatic evaluation tool.

Kuusijärvi et al. [15] proposed to strengthen IoT security through a network edge device (NED), a secure device which stores the user-defined policies and enforces them on resource-constraint IoT devices. However, this kind of approach fails to identify specific requirements of the devices (i.e., they do not envision anything like a contract), offloading on the end-user the cumbersome task of defining fine-grained policies.

Other researchers proposed Fog-based policy enforcement approaches to solve different problems in the IoT world, for example, ensuring data privacy [16] and providing secure resource orchestration in Fog computing [17]. Similar to our work, they share the necessity of enforcing policies at the Fog layer over devices that might not be compliant by design.

Cisco manufacturer usage descriptions (MUD) [8] shares a number of common traits with our proposal. First and foremost, they envision a file that acts as a contract and states the device requirements. Second, the MUD file is parsed by a special node within the network (the MUD server) which defines appropriate access control lists (ACLs). However, a MUD file is pretty restrictive, as it barely describes basic information like allowed protocols and reachable hosts. Moreover, each MUD-compliant device does not carry the contract itself, but a simple URI that points the MUD manager to the online MUD file: Lack of Internet access would entail the total incapability of joining a network. Moreover, it is unclear how the contracts should be parsed, verified, and treated with respect to a security policy. Here, the key S × C concept of contract/ policy matching, as well as a way to enforce policies on untrustworthy devices, seems to be missing.

Other researchers built on MUD. Hamza et al. [18] tried to undertake the problem of enforcing policies by means of combination with a software defined network (SDN). In this context, the authors produced a translation from MUD policies to routing rules, aiming to implement the result into network switches. Again, they also noted that flow-based rules can be a first step to identify volumetric attacks, but they are not perfect. In particular, if such attacks happen on intended ports, MUD alone cannot be enough to identify the ongoing attack.

Another work proposed an automatic process, mainly (but not exclusively) targeted at manufacturers, to extract a MUD file from an IoT device traffic trace. Moreover, they sketched the necessity of formally matching MUD files with network security policies [11].

10. Future Work

As future work, we plan to further expand on the matching routines and the dynamic evolution of an S × C based environment. For instance, what happens when a new device receives a software update, or when contracts and policies receive updates, or when we try to remove from the network a device which is critical for other functionalities. Moreover, we plan to mitigate the threats discussed in Section 8. In particular, we will focus on reducing the susceptibility of the Fog node to disruptive attacks.

With this in mind, we are currently developing a Java prototype of an S × C IoT system. Our goal is to show the feasibility of our approach and demonstrate a few key points. First, we want to show that our system has the capability of detecting inconsistencies. Second, we want to show that our system solves or signals each and every inconsistency. As a consequence of the first two steps, we want to show that an S × C system can maintain consistency over time. Last, we want to show that such a system allows only intended communication flows and prevents unintended data leaks.

As an example of the ongoing implementation, in Figure 7 we show a Java code-snippet. In particular, this function tests whether our implementation of IllegalInformationExchange is correct.

```java
@Test
final void testIllegalInformationExchangeD1ToPF() {
    init();

    contractCD1 = new Contract("files/Contract_D1.json");
    contractCD2 = new Contract("files/Contract_D2.json");

    policyPF.addContract(contractCD2);

    assertTrue(contractCD1.illegalInformationExchange(policyPF),
            "CD1 shares with Apple.LukePhone, but CD2 does NOT, thus"
            + "we have an Illegal Information Exchange");
}
```

Figure 7. Code snippet that tests the implementation of IllegalInformationExchange.

Last but not least, we are working on extending our framework for integrating devices that are not S × C compliant (that is, devices that do not have a contract). We believe this represents a fundamental point for the adoption of our approach.

11. Conclusions

In this paper, we have proposed a novel approach that combines security-by-contract (S × C) and Fog computing to tackle the issue of IoT insufficient security configurability.

In our proposal devices are equipped with security contracts that can be verified against the security policy stored within the Fog node. In turn, this allows for tractable contracts and policies, efficient matching routines, and trustworthy IoT environments. By means of a running case scenario and a number of illustrative examples, we have defined all the necessary pillars to develop our proposal, from basic security rules up to consistent security policies. Moreover, we have provided pseudo-code algorithms which show the feasibility of our approach.

Author Contributions: Investigation, A.G. and N.D.; Supervision, N.D. and F.M.; Validation, A.G., N.D. and F.M.; Writing—original draft, A.G. and N.D.; Writing—review and editing, A.G. and N.D.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Dragoni, N.; Giaretta, A.; Mazzara, M. The Internet of hackable things. In *Proceedings of the 5th International Conference in Software Engineering for Defence Applications, Rome, Italy, 10 May 2016*; Ciancarini, P., Litvinov, S., Messina, A., Sillitti, A., Succi, G., Eds.; Springer: Cham, Switzerland, 2017; pp. 129–140.
2. Zhou, W.; Jia, Y.; Peng, A.; Zhang, Y.; Liu, P. The Effect of IoT New Features on Security and Privacy: New Threats, Existing Solutions, and Challenges Yet to Be Solved. *IEEE Internet Things J.* **2019**, *6*, 1606–1616. [CrossRef]
3. Bonomi, F.; Milito, R.; Natarajan, P.; Zhu, J. Fog computing: A platform for Internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*; Bessis, N., Dobre, C., Eds.; Springer: Cham, Switzerland, 2014; pp. 169–186. [CrossRef]
4. Dragoni, N.; Massacci, F.; Naliuka, K.; Siahaan, I. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Public Key Infrastructure, Proceedings of the (PKI'07) 4th European PKI Workshop: Theory and Practice, Palma de Mallorca, Spain, 28–30 June 2007*; Lopez, J., Samarati, P., Ferrer, J.L., Eds.; Springer: Cham, Switzerland, 2007; pp. 297–312.
5. Dragoni, N.; Massacci, F.; Walter, T.; Schaefer, C. What the heck is this application doing? A security-by-contract architecture for pervasive services. *Comput. Secur.* **2009**, *28*, 566–577. [CrossRef]
6. Dragoni, N.; Gadyatskaya, O.; Massacci, F. Supporting applications' evolution in multi-application smart cards by security-by-contract. In Proceedings of the 4th Workshop in Information Security Theory and Practices (WISTP 2010), Passau, Germany, 12–14 April 2010; Springer: Cham, Switzerland, 2010.
7. Athreya, A.P.; DeBruhl, B.; Tague, P. Designing for self-configuration and self-adaptation in the Internet of Things. In Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, Austin, TX, USA, 20–23 October 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 585–592. [CrossRef]
8. Lear, E.; Weis, B. Slinging MUD: Manufacturer usage descriptions: How the network can protect things. In Proceedings of the International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT), Cairo, Egypt, 11–13 April 2016; IEEE: Piscataway, NJ, USA, 2016.
9. Alirezaie, M.; Renoux, J.; Kockemann, U.; Kristoffersson, A.; Karlsson, L.; Blomqvist, E.; Tsiftes, N.; Voigt, T.; Loutfi, A. An ontology-based context-aware system for smart homes: E-care@home. *Sensors* **2017**, *17*, 1586. [CrossRef] [PubMed]
10. Giaretta, A.; Dragoni, N.; Massacci, F. Protecting the Internet of Things with security-by-contract and Fog computing. In Proceedings of the 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), Limerick, Ireland, 15–18 April 2019.
11. Hamza, A.; Ranathunga, D.; Gharakheili, H.H.; Roughan, M.; Sivaraman, V. Clear as MUD: Generating, validating and applying IoT behavioural profiles. In Proceedings of the 2018 Workshop on IoT Security and Privacy, Budapest, Hungary, 20 August 2018; ACM: New York, NY, USA, 2018; pp. 8–14. [CrossRef]
12. Eterovic-Soric, B.; Choo, K.K.R.; Ashman, H.; Mubarak, S. Stalking the stalkers–detecting and deterring stalking behaviours using technology: A review. *Comput. Secur.* **2017**, *70*, 278–289. [CrossRef]

13. Agadakos, I.; Chen, C.Y.; Campanelli, M.; Anantharaman, P.; Hasan, M.; Copos, B.; Lepoint, T.; Locasto, M.; Ciocarlie, G.F.; Lindqvist, U. Jumping the air gap: Modeling cyber-physical attack paths in the Internet-of-Things. In Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy, Dallas, TX, USA, 30 October–3 November 2017; ACM: New York, NY, USA, 2017; pp. 37–48. [CrossRef]
14. Matheu-García, S.N.; Hernández-Ramos, J.L.; Skarmeta, A.F.; Baldini, G. Risk-based automated assessment and testing for the cybersecurity certification and labelling of IoT devices. *Comput. Standard. Interfaces* **2019**, *62*, 64–83. [CrossRef]
15. Kuusijärvi, J.; Savola, R.; Savolainen, P.; Evesti, A. Mitigating IoT security threats with a trusted network element. In Proceedings of the 11th International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona, Spain, 5–7 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 260–265. [CrossRef]
16. Al-Hasnawi, A.; Mohammed, I.; Al-Gburi, A. Performance evaluation of the policy enforcement Fog module for protecting privacy of IoT data. In Proceedings of the 2018 IEEE International Conference on Electro/Information Technology (EIT), Rochester, MI, USA, 3–5 May 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 951–957. [CrossRef]
17. Dsouza, C.; Ahn, G.; Taguinod, M. Policy-driven security management for Fog computing: Preliminary framework and a case study. In Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IRI 2014), Redwood City, CA, USA, 13–15 August 2014; pp. 16–23. [CrossRef]
18. Hamza, A.; Gharakheili, H.H.; Sivaraman, V. Combining MUD policies with SDN for IoT intrusion detection. In Proceedings of the 2018 Workshop on IoT Security and Privacy, Budapest, Hungary, 20 August 2018; ACM: New York, NY, USA, 2018; pp. 1–7. [CrossRef]