S×C4IoT: A Security-by-contract Framework for Dynamic Evolving IoT Devices

ALBERTO GIARETTA, Centre for Applied Autonomous Sensor Systems (AASS), Örebro University, Sweden
NICOLA DRAGONI, DTU Compute, Technical University of Denmark, Denmark
FABIO MASSACCI, Department of Information Sciences and Engineering, University of Trento, Italy and Vrije Universiteit Amsterdam, Netherlands

The Internet of Things (IoT) revolutionised the way devices, and human beings, cooperate and interact. The interconnectivity and mobility brought by IoT devices led to extremely variable networks, as well as unpredictable information flows. In turn, security proved to be a serious issue for the IoT, far more serious than it has been in the past for other technologies. We claim that IoT devices need detailed descriptions of their behaviour to achieve secure default configurations, sufficient security configurability, and self-configurability. In this article, we propose S×C4IoT, a framework that addresses these issues by combining two paradigms: Security by Contract (S×C) and Fog computing. First, we summarise the necessary background such as the basic S×C definitions. Then, we describe how devices interact within S×C4IoT and how our framework manages the dynamic evolution that naturally result from IoT devices life-cycles. Furthermore, we show that S×C4IoT can allow legacy S×C-noncompliant devices to participate with an S×C network, we illustrate two different integration approaches, and we show how they fit into S×C4IoT. Last, we implement the framework as a proof-of-concept. We show the feasibility of S×C4IoT and we run different experiments to evaluate its impact in terms of communication and storage space overhead.

CCS Concepts: • **Security and privacy** → **Distributed systems security**; *Network security*; *Access control*; • **Computer systems organization** → *Sensor networks;*

Additional Key Words and Phrases: IoT, internet of things, security, security-by-contract, S×C, fog computing, configurability, self-configurability, declarative security

ACM Reference format:
Alberto Giaretta, Nicola Dragoni, and Fabio Massacci. 2021. S×C4IoT: A Security-by-contract Framework for Dynamic Evolving IoT Devices. *ACM Trans. Sen. Netw.* 18, 1, Article 12 (September 2021), 51 pages.
https://doi.org/10.1145/3480462

1 INTRODUCTION

The **Internet of Things (IoT)** paradigm has been one of the key enablers of pervasive computing. IoT devices not only are capable of forming complex interconnected systems with other IoT

Authors' addresses: A. Giaretta, Centre for Applied Autonomous Sensor Systems (AASS), Örebro University, Fakultetsgatan 1, 70182 Örebro, Sweden; email: alberto.giaretta@oru.se; N. Dragoni, DTU Compute, Technical University of Denmark, Richard Petersens Plads, 2800 Kongens Lyngby, Denmark; email: ndra@dtu.dk; F. Massacci, Department of Information Sciences and Engineering, University of Trento, Via Sommarive 9, 38123 Povo, Trento, Italy and Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, Netherlands; email: fabio.massacci@unitn.it.

1550-4859/2021/09-ART12 $15.00
https://doi.org/10.1145/3480462

devices; they can also connect to the Internet and interpret their computational results in a broader context. However, the IoT and its improved connectivity brought critical implications and shortcomings from a cyber-security point of view [13].

Since the advent of IoT, the research community raised concerns about the security challenges that this new paradigm brought [11, 40, 55, 56, 61]. Some challenges, such as weak passwords and lack of security updates, stem from best practices overlooked at the design phase [13]. Others, such as integration and automatic configuration, derive from the heterogeneous nature of the IoT itself [62]. To make matters worse, IoT devices can be designed for radically different goals. Devices range from smart plugs that turn appliances on/off, to fleets of devices that connect whole factories. Having to achieve different tasks, IoT devices have a large variety of requirements and capabilities regarding their hardware and software. They can equip different sensors, different communication protocols, and even different amounts of computing power. Heterogeneity is a strong point of IoT, as well as a weak one. On the one hand, the variety of devices enables to tailor solutions for specific requirements. On the other hand, achieving a robust and secure infrastructure is far harder than it used to be with traditional computer networks.

Even though computing capabilities vary across different IoT devices, most of those devices are computationally limited. 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.

Fog computing is a Cloud computing extension [7] proposed for counterbalancing the aforementioned Cloud issues. The key idea is to move critical services from the Cloud to the edge of the network, closer 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. The Fog goal is to remedy part of Cloud drawbacks by centralising sensor data, undertaking time-bound tasks, and gatekeeping data traffic. Moreover, a Fog node can be viewed as 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 Article

In this article, we address two IoT security issues yet to be solved. Namely, the insecure default configuration problem and the insufficient security configurability [62] problem. The insecure default configuration problem states that IoT devices are routinely shipped with poor configurations, with respect to cybersecurity best practices. The insufficient security configurability problem states that current IoT devices do not offer enough tools for configuring them according to users' necessities. In the next section, we further discuss these issues.

To tackle these issues, we propose to combine the **security-by-contract (S×C)** paradigm and Fog computing. S×C allows to bind a behavioural description of a device to its embedded software and to formally prove such binding. S×C has been successfully applied to different scenarios, such as mobile applications [14, 15] and multi-application smart cards [12]. In our previous work [25, 26], we already discussed how S×C can be combined with Fog computing and we provided the necessary formal definitions, paving the way for S×C4IoT, an S×C framework for IoT devices.

However, in our previous work, we did not tackle some critical aspects. Formal definitions allowed us to show that S×C could work, but we did not provide a working proof-of-concept to show that our approach works. Consequently, we did not conduct experiments for evaluating to which extent contracts can impact devices with limited resources. Moreover, in our previous work, we

did not take into consideration the natural life-cycle of modern devices, which can change their on-board software and their emergent behaviour. Last, in other studies [44, 59], we considered the problem of allowing legacy IoT devices to participate with an S×C network, but we did not undertake the task of integrating these techniques in a cogent framework such as S×C4IoT and we did not define how they would operate with dynamic evolution flows.

Therefore, the contributions of this article are manifold:

- We propose a framework for security configurations, configurability, and self-configurability, which are important requirements in the fast-growing IoT network.
- We summarise the fundamental S×C formal definitions previously defined in Reference [25].
- We recall two approaches for allowing S×C-noncompliant devices to participate with an S×C network, and we show how these techniques fit into the S×C4IoT framework.
- We go beyond fictitious static networks assumed in Reference [25], and we undertake the management of dynamic evolving IoT devices and their cooperation with the entire S×C network.
- We provide an architectural diagram and a Java-based implementation of S×C4IoT, a proof-of-concept S×C framework for IoT devices, proving the feasibility of an IoT S×C network envisioned in our previous work [25].
- We run experiments to evaluate S×C4IoT performance and penalties, showing that our proposal is technically sound and the resulting overhead reasonable.

1.2 Article Outline

This article is organised as follows: In Section 2, we state the problem addressed in this work. In Section 3, we describe S×C4IoT from a high-level perspective, including an overview on the critical services offered by the S×C4IoT framework, and the threat model taken into consideration throughout the article. In Section 4, we describe the pillars for managing S×C-compliant devices, whereas in Section 5, we describe two different techniques for allowing S×C-noncompliant devices to interact with an S×C network. In Section 6, we provide the workflows that enable S×C4IoT to deal with the dynamic evolutions that naturally occur in every device life-cycle. In the same section, we describe the proof-of-concept we implemented, and we evaluate the performance. In Section 8, we discuss the limitations of this study and potential future work directions. Finally, in Section 9, we give an overview of the related work, and in Section 10, we draw our conclusions.

2 PROBLEM STATEMENT

Zhou et al. [62] analysed the main security challenges that the IoT will face in the near future. They conducted a quantitative analysis and found out that, until now, research has mostly focused on privacy leaks and insecure network communication issues, leaving some open challenges. Among the most relevant challenges, the authors highlighted two threats to nowadays IoT devices: default insecure configurations provided by the manufacturers and insufficient security configurability.

As previously defined in Section 1.1, the first problem highlights that IoT devices are routinely shipped with default configurations that do not meet minimum security standards. This is particularly dangerous, given that users usually do not change their devices' default configurations. The second problem notices that common IoT devices do not offer meaningful granularity in their configuration options, preventing to grant detailed permissions to specific users.

As an example of insecure default configuration, some IoT devices are shipped with default admin passwords that are easy to guess and rarely changed. Furthermore, in the name of availability, many IoT devices expose all their services to any incoming request. IoT devices might exhibit services that are never used and that, at the same time, unnecessarily increase their chance of suffering

vulnerabilities exploitation. This is done because manufacturers prefer to ensure functionality and produce devices with less restrictive privileges, instead of risking devices not working off-the-shelf. A better solution would be to equip the devices with formal descriptions that list the conditions under which services are offered and required. Formal behavioural descriptions would allow IoT devices to achieve better security off-the-shelf without sacrificing availability.

Regarding insufficient security configurability, the prime example is that IoT devices usually provide only administrator login, lacking any permission granularity. Another example, IoT devices do not provide tools for defining and announcing their behaviour within a network, which prevents from regulating complex interactions between actors, whether they are human beings or devices. For example, an administrator might want to allow access to an IoT camera live-stream only to devices that do not communicate over the Internet. Achieving this goal requires a number of conditions. First, the administrator must express and formalise the constraint. Second, devices that require access to the live-stream must prove that they meet the requirement of not communicating over the Internet. Last, the IoT camera must be able to selectively grant access to the live stream, but not to other services. These conditions are easy to describe and understand in natural language. However, implemented as formal rules, they can grow in number and decrease in intelligibility. A high-level semantics, both understandable by human beings and formally interpretable by machines, is critical for improving IoT devices configurability.

Last, but not least, Athreya et al. [5] took into consideration the scalability issues that the IoT will face in the future. Gartner predicts that by the end of 2020, around 20 billion IoT devices will be connected to the Internet, and 5.8 billion of those IoT devices will be enterprise endpoints, such as automotive sensors and smart meters [27]. Current Cloud infrastructures are unlikely to keep up with the computational resources required to manage this number of devices. IoT devices will have to be able to adapt to the environment they are immersed in and configure themselves accordingly.

To achieve self-configurability, IoT devices need to be aware of the surrounding environment in terms of devices and services. However, the answer cannot come from IoT devices' direct observation of network traffic. Direct observation entails data extrapolation and data analysis, and it requires a considerable amount of time and computing power, unavailable to most IoT devices. Therefore, behavioural descriptions are of paramount importance for allowing IoT devices to self-configure and cooperate. From this point of view, this issue is strongly related both to the insufficient security configurability problem and the insecure default configuration one.

3 S×C4IOT: A S×C FRAMEWORK FOR IOT

In this section, first, we illustrate the S×C framework from a high-level point of view, as shown in Figure 1. Then, we describe the threat model assumed in this work, and we list the services that IoT devices and Fog Nodes should offer to achieve a robust and functional S×C framework.

The S×C framework [15, 26] is based on two fundamental concepts, the security *contract* and the security *policy*. A security contract (or simply, a contract) specifies an IoT device behaviour for what concerns relevant security actions. Every S×C-compliant device stores a contract and exhibits it to the network before being allowed to participate in the network. The manufacturers create the S×C contracts for their own IoT devices (Stage 2A in Figure 1).

Similarly, a security policy (or simply, a policy) specifies the acceptable behaviour of the IoT devices for what concerns their relevant security actions. A policy is stored by a trustworthy device within a network, such as a Fog node, which is responsible for verifying that IoT devices' behaviour complies with the security policy. We refer to the process of verifying a contract against a policy as *contract/policy matching*, as shown in Figure 1 Stage 5. For the scope of this article, we restrict the set of possible relevant security actions to the possible interactions between IoT devices (for a

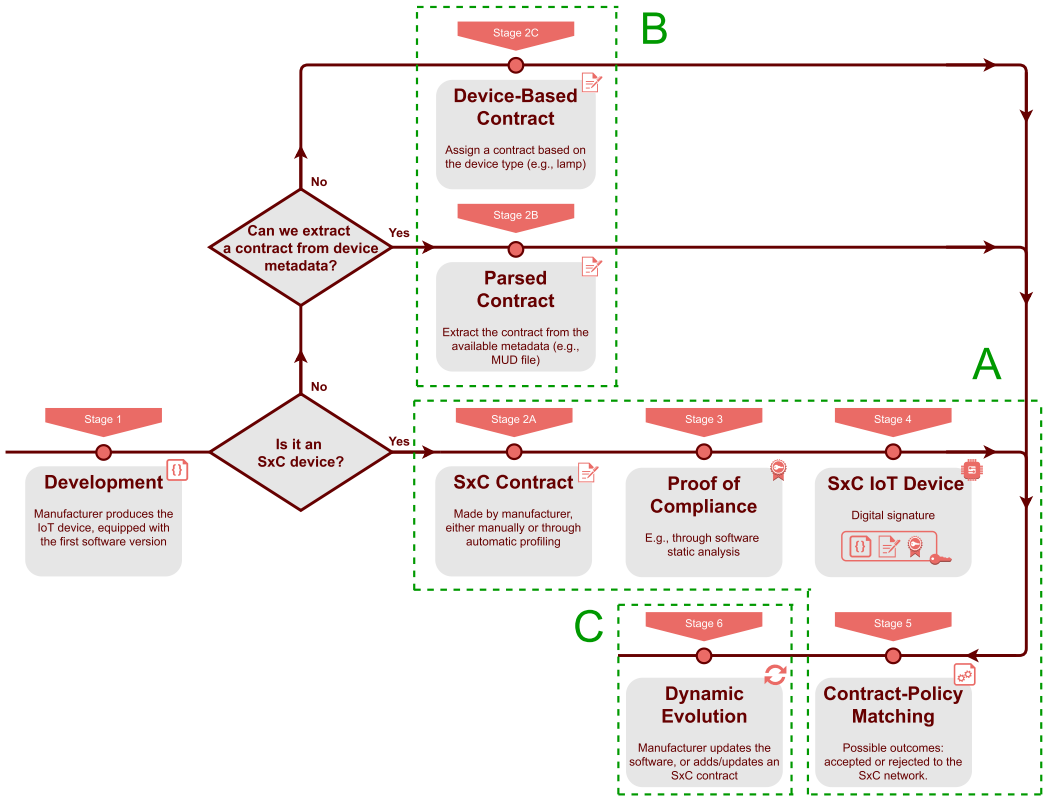

Fig. 1. S×C4IoT Framework, main phases overview. Box A encompasses the basic components necessary for an S×C framework and includes Stages 2A, 3, 4, and 5, outlined in Section 4; for a recap of the formal definitions, refer to Appendix A. Box B, composed of Stage 2B and 2C, describes two techniques that can be implemented for allowing S×C-noncompliant devices to be included in S×C4IoT. These techniques are not critical for the base functionalities of S×C4IoT, and different techniques for identifying IoT devices could be similarly used. Last, Box C corresponds to Stage 6 and details the algorithms for managing IoT devices dynamic evolution as described in Section 6.

broader set of actions, we refer the reader to Giaretta et al. [26]). Concerning this work, a contract describes which resources might be necessary for the device to operate and which resources it provides to other devices. Similarly, a policy declares what actions are allowed within the network it regulates and what resources its devices need/provide.

However, contracts alone are not enough. A device might carry a contract that describes a behaviour that complies with the security policy, but it might not adhere to the description. It is necessary to bind the actual behaviour (i.e., the software) to the description (i.e., the contract), in a way that the framework can formally verify the correspondence. In S×C, this is done by means of a **Proof-of-Compliance (PoC)**, shown in Figure 1, Stage 3.

The PoC is a proof, signed by the manufacturer and stored in the IoT device, which binds the device software to its contract. As envisioned by Reference [15], the PoC is an instantiation of Necula and Lee **Proof-Carrying Code (PCC)** [46, 48]. A valid PoC should be hard to create and easy to formally verify using an external validator. PCC [46, 48], Sekar et al. **Model-Carrying Code (MCC)** [53, 54] is a notable example of security-enriched execution codes. Clearly inspired by PCC, MCC envisions the enrichment in the form of a model that captures the security relevant

behaviour of the software, instead of a formal proof. In turn, instead of verifying the formal proof validity, the end consumers check their policy against the model and determine whether the code will violate the policy or not. Necula further discussed the complexity of producing proofs for executables that are compiled with optimised compilers and correctness proofs in general [47, 49]. Moreover, Zhang and Zuck recently conducted a survey about the formal verification of executable code produced with optimising compilers [60].

Following Necula and Lee PCC, in S×C a manufacturer produces a proof regarding the safety of the on-board software and stores it inside the IoT device storage. The PoC can be automatically generated running static analysis techniques on the device software, checking for the properties listed in the S×C contract. If the PoC matches both code and contract, then the triplet *<Software, Contract, PoC>*is *consistent*. If the three elements do not match, then the triplet is *inconsistent*. To tamper with an S×C device, an attacker would have to forge a new contract, a new PoC, and sign everything with the manufacturer private key. When an S×C device attempts to join the network, or when it behaves unexpectedly, as shown in Section 6, the network (i.e., the Fog node) checks the correctness of the proof by means of a proof validator check. As aforementioned, the PoC is generated by the manufacturer and stored inside the IoT device at production time. Therefore, S×C-compliant IoT devices do not require integrated PoC computing schemes. One upside of this approach is that simple S×C devices differ from complex ones only in terms of contract size and PoC size. Consequently, instantiating an IoT device as an S×C-compliant device does not require additional components nor computational power and only produces reasonable storage and transmission overhead, as discussed in Section 7.2. Stages from 2A to 5 in Figure 1, grouped in box A (in dashed green lines), compose the backbone structure of the S×C4IoT framework. For an in-depth treatment of the subject, we refer the reader to Appendix A and to our previous work [25, 26].

In this work, it is assumed that an IoT device could be either S×C-compliant or not, as shown in Figure 1 Stages 2B and 2C (highlighted with box B). In the case that a manufacturer produces an S×C-compliant device, it is part of its duty to produce a valid S×C security contract and a valid PoC and sign the triplet *<Software, Contract, PoC>*. At this point, the device is S×C-compliant and can be verified against an S×C network policy. However, if a legacy device does not comply with S×C, then it is necessary to produce a valid contract for allowing the device in an S×C network. In Section 5, we show two different approaches for creating such contracts. Namely, **Parsing Contracts (PC)** in Figure 1, Stage 2B, and **Device-Based Contracts (DBC)** in Figure 1, Stage 2C. PC contracts are created by parsing the device metadata, such as a **Manufacturer Usage Description profile (MUD)** (for more information, refer to Section 4), extracting the device behaviour, and producing an S×C contract that describes the behaviour. However, PC might not be sufficient, as metadata is often non-standardised and incomplete. As a fallback approach, the framework analyses the available metadata, discovers the device type (e.g., a lamp) and assigns to the device a general contract that suits the discovered type. These generic contracts are the DBC contracts. In Section 5.1 and Section 5.2, we illustrate in detail how these two approaches work. For the scope of this section, it is worth noting that these methods might produce a contract or not, and that the produced contract might not be complete.

Last, throughout the life-cycles of devices and networks, updates are possible and desirable. IoT devices might receive software updates, S×C contract updates, or a new S×C contract for the first time. Network policies can change as well, and devices might have to be removed from the network for being non-compliant or prevented from applying updates that would cause illegal information exchanges. In any case, a successful contract/policy match does not guarantee an everlasting match. It is necessary to predict and manage every potential dynamic evolution that could happen. These cases are taken into account in Figure 1, Stage 6 (green box C), and discussed in-depth in Section 6.

Table 1. S×C4IoT Framework Services

Fog Node Service	Stage	Description
addDevice	5, 6	Allows (or forbids) an IoTDev to join the network.
removeDevice	6	Removes an IoTDev from the network. If it is a critical device, warns the administrator and tries to prevent the removal.
updateSoftware	6	Verifies if an IoTDev software update complies with the network policy. Update is allowed or rejected.
updateContract	6	Verifies if an IoTDev contract update complies with the network policy. Update is allowed or rejected.
addContract	6	Adds a single contract to the Fog Node.
updatePolicy	6	Updates the Fog Node policy with a new one. Tries to keep as many contracts as possible from the old policy, so every device already present in the network is maintained.
compliantWithPolicy	5, 6	Verifies if a contract is compliant with the policy currently enforced by the Fog Node.
containsContract	5, 6	Verifies if a contract is already part of the policy currently enforced by the Fog Node.
validPoC	5, 6	Verifies if IoTDev has a valid PoC, whether for a potential new contract or its current one.
Iot Device Service	**Stage**	**Description**
joinNetwork	5, 6	Calls the method *addDevice* when joining the network. The Fog Node might allow or reject it.
leaveNetwork	6	Calls the method *removeDevice* when leaving the network. The Fog Node might allow or reject it.
updateContract	6	Calls the method *updateContract* when updating the contract. The Fog Node might allow or reject it.
updateSoftware	6	Calls the method *updateSoftware* when updating the software. The Fog Node might allow or reject it.
DBC Service	**Stage**	**Description**
canExtractContract	2B, 2C	Verifies if the DBC component can extract (or parse) a contract for an IoTDev.
extractContract	2B, 2C	Extracts (or parse) a contract for IoTDev.

3.1 S×C Services

For providing the core features presented in Figure 1, an S×C4IoT framework should provide certain minimum services. For example, the framework must be able to assess a triplet <*Software, Contract, PoC*>validity and verify a device contract against a policy. Moreover, the framework should provide every service that is necessary for managing foreseen dynamic variations.

In Table 1, we list the services of a functional S×C4IoT framework, covering both mandatory and optional service. In this table, we divide the services based on the component that provides them, we give a description of the core functionalities, and we specify for which phases these services are necessary with respect to the framework in Figure 1.

3.2 Threat Model

In our previous work [25], we gave a bird's-eye view on how S×C4IoT can cope with different malicious events. However, in Reference [25], we only assumed semistatic scenarios. A single device did not change, nor introduce inconsistencies over time, and was solely evaluated against one (or more) static devices. Even in more complex examples, where we modelled hidden paths and security degradation scenarios, we showed that S×C can deal with those problems only by means of static comparison. In this work, we show that S×C can effectively apply also to real dynamic scenarios where device triplets <*Software, Contract, PoC*>evolve throughout time.

Before introducing the threat model, let us split every possible action within an S×C network into two groups. Namely, *Fog-actionable* events and *Fog non-actionable* events. Fog-actionable events are the ones that a Fog node can observe, detect, and react to; Fog non-actionable events are all those events (such as intrusions and malfunctions) that cannot be detected by a monitoring Fog node.

In this article, S×C4IoT detects and reacts to Fog-actionable events. An example of a Fog-actionable event is an attacker that compromises a device behaviour for communicating with a

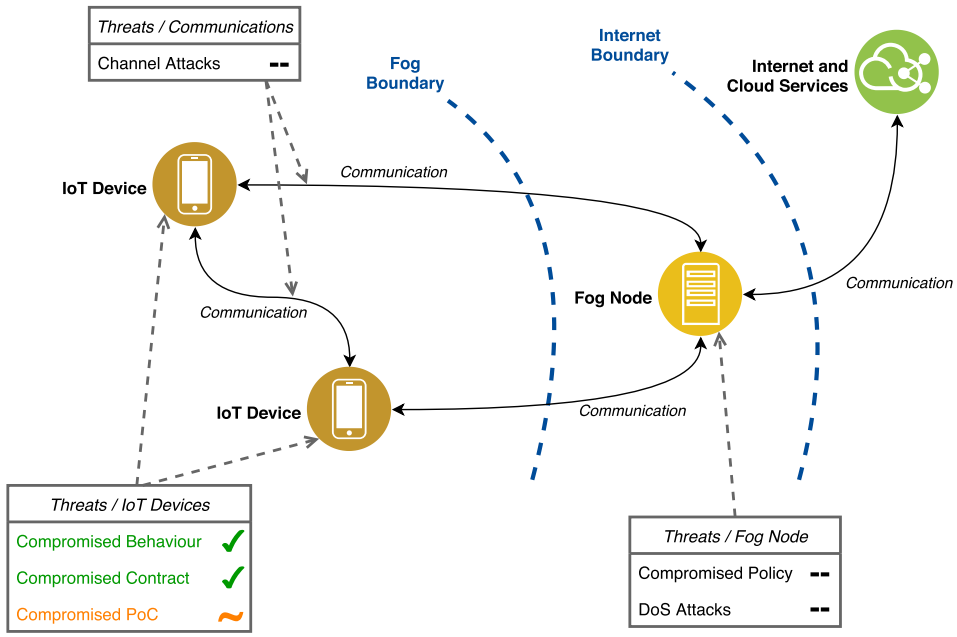

Fig. 2. Graphical representation of the architecture threat model. In this article, we do not deal with potential threats to the communication channels between devices, as well as potential threats to the Fog node integrity. We assume that the policy stored in the Fog node cannot be tampered with. Also, we assume that the communications between the devices are secure.

device not listed in the contract. In this case, the Fog node can detect a discrepancy between device contract and behaviour. This is a Fog-actionable event that a Fog node can identify and react to by isolating the device or removing it from the network. In S×C4IoT, malicious attackers can corrupt a PoC but cannot create a valid one, hence any illegal modification of the triplet or any event that violates its semantics can be detected by the Fog node. Different strategies can be used to achieve this monitoring. A Fog node could randomly poll IoT devices and ask to exhibit their triplet, achieving random challenges. Alternatively, if a device receives messages from a compromised device, and these messages violate the network policy, then the recipient device could trigger a warning to the Fog node, which then proceeds to verify the suspicious device. In Section 6, we provide a list of dynamic evolution events that are Fog-actionable events, showing that S×C4IoT manages them correctly.

However, let us assume that a device is infected with a malware that does not modify any component of the triplet <*Software, Contract, PoC*>, but performs malicious actions within the contract boundaries. An example would be a malware that infects a temperature sensor and alters the temperature readings. Any other action of the sensor works as expected, only the raw data is corrupted. This would be a Fog non-actionable event because, from the point of view of the Fog node, the device is acting according to its intended behaviour and the triplet <*Software, Contract, PoC*>results intact. Fog non-actionable events cannot be mitigated with S×C4IoT, but different strategies can be integrated into our framework to complement it, as we discuss in Section 8.

As shown in Figure 2, for the scope of this work, we lay these assumptions:

(1) The network is equipped with a Fog node that monitors which devices join the network and when communication sessions start;

(2) The Fog node is trustworthy and cannot be tampered with. For example, DoS attacks against the Fog node cannot happen;
(3) An attacker cannot create a valid PoC;
(4) An attacker cannot get hold of the private key. Therefore, an attacker cannot tamper with the triplet <*Software, Contract, PoC*>and then sign it to make it valid;
(5) Communication channels are secure, so replay attacks or **man-in-the-middle (MitM)** attacks cannot be performed;
(6) Fog non-actionable events, such as raw data tampering, are out of scope.

Taking into consideration the assumptions we just mentioned, our framework considers possible, and addresses, the following threats:

(1) An attacker can compromise an IoT device software;
(2) Following the previous item, an IoT device could exhibit unexpected behaviour, such as requesting to communicate with another device it is not supposed to communicate with or performing DoS attacks on other devices (Fog node excluded);
(3) An attacker can compromise an IoT device contract;
(4) Following the previous item, an IoT device can exhibit a different behaviour than the one described in its contract;
(5) An attacker can compromise the PoC (but not create a valid one, as previously stated);
(6) An S×C4IoT Fog node can detect Fog-actionable events and react.

4 TERMINOLOGY

This section provides an overview on the basic terminology used in this article. The related formal definitions are given in Appendix A.

Rule. A rule is a 5-tuple that lists: the name of the device D that the rule relates to; Dom, the name of the domain within the rule applies to; Provides, the list of the services that are provided by the device; Shares, the list of the devices that can access to the provided services; and last, Invokes, the list of the services that might be invoked by the device for providing their services.

Contract. A contract is a set of rules that describe the specification of the behaviour of an IoT device, for what concerns its relevant security actions. Therefore, the necessary condition is that every rule that compose a contract must report the same device name in the D field.

Consistent Contract. A contract is consistent if all the rules that compose the contract are well formed and core. 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. Core contract means that no rule in a contract restricts any other rule in the same contract.

Policy. Similarly to a contract, a policy is a set of rules that describes the acceptable behaviour of the IoT devices within the network, for what concerns their relevant security actions. Given that the goal of a policy is to manage the behaviour of different devices, there is no restriction about the D field, unlike contracts.

Allowed Direct Communication. Two devices have an allowed direct communication if D_A provides one or more services to D_B, and D_B invokes one or more services provided by D_A.

Illegal Information Exchange. If two devices exhibit an illegal information exchange, then D_B cannot use the information obtained by invoking D_A services, for providing services to a third device D_C, unless D_A provides explicitly the services also to D_C.

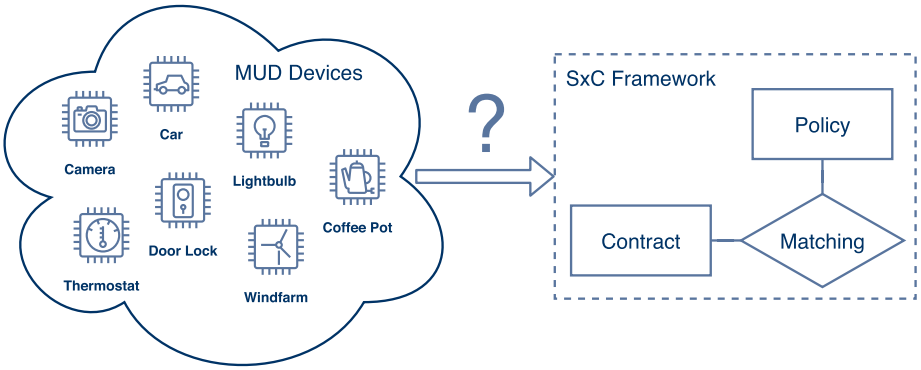

Fig. 3. MUD-compliant devices are growing in number. How can we integrate such devices with an S×C framework? We need a method for extracting S×C contracts from MUD definitions.

Consistent Policy. A policy is consistent if the rules that compose it are well formed and core, and if there is no illegal information exchange between any contract that belongs to the policy.

Contract-policy Matching. A consistent contract matches a consistent policy if the policy that results from their combination preserves its consistency.

Manufacturer Usage Description (MUD). MUD [36] is an IETF specification (RFC8520), in which manufacturers specify which hosts and ports their devices need to operate correctly. This specification comes in form of a file that lists the devices requirements. For more details, we refer the reader to Section 9.

5 MANAGING S×C-NONCOMPLIANT DEVICES

As aforementioned, assuming IoT devices to be S×C-compliant is a considerable restriction, which would require S×C to become a standard among devices manufacturers. As shown in Figure 3, it is critical to envision a strategy for allowing legacy devices to join S×C networks.

In this section, we investigate the problem using MUD devices as a case study. We select MUD because an increasing number of commercial IoT devices are compliant with MUD, showing that our framework could work on top of other technologies deployed on the market. In particular, we summarize two different approaches that we previously proposed in different works, and we show how they fit into a comprehensive S×C4IoT framework. The first approach is called **Parsed Contracts (PC)** and consists of a parser that extracts from a MUD file the necessary information and creates an S×C contract (Figure 1, Stage 2B) [44].

However, the MUD standard envisions few mandatory fields and many fields of a MUD file might result empty. Therefore, the PC approach might not allow to gather enough information. The second approach, called Device-Based Contracts (DBC, in Figure 1, Stage 2C), tries to overcome PC shortcomings, analysing the MUD file fields and discovering the device type (e.g., a lamp or a camera) [59]. According to the discovered device type, the Fog node can assign to the MUD device a generic contract, from a set of default predefined contracts. For example, a smart home owner might want to prevent cameras to transmit information outside the network, for privacy reasons. Consequently, the generic camera contract would specify that the device can communicate with anyone within the network and impede communications over the Internet.

A clear upside of this approach is that it is not necessary to produce a PoC for legacy devices: The classification is performed by the Fog node and the default contracts assigned to legacy devices

are pre-defined and stored by the Fog node itself. A PoC that binds an S×C-noncompliant device to its contract is irrelevant, since that the underlying assumption is that the contract is based on the device type. While this approach is suboptimal, it is a necessary compromise for ensuring retro-compatibility with devices that would never achieve S×C-compliancy.

It is worth noting that, even though DBC focuses on the MUD devices case study, in principle it can be applied to different paradigms. DBC works on the assumption that IoT devices belonging to same categories (e.g., smart lamps) exhibit similar behaviour; therefore, if an IoT device is correctly classified, then it is possible to assign a correct default S×C contract. In light of this, any classification technique could be used, as long as it classifies an IoT device. There are a number of notable examples of classification approaches, such as banner-grabbing, a technique for extracting relevant information from devices' application layer. Durumeric et al. [17] proposed Censys, a public search engine that scans devices over the Internet, identifies them, and annotates the results with community-maintained annotations. Feng et al. [19] proposed an acquisitional rule-based engine for classifying and labelling IoT devices, and Mavrogiorgou et al. [45] compared different classification algorithms on a dataset containing specifications of different IoT devices.

We implemented a proof-of-concept for PC and DBC in Python, as it provides the largest choice of different **Machine Learning (ML)** libraries. We deliberately decided not to integrate the Python scripts in the S×C4IoT Java framework (described later in Section 6.6), even though it would have been easy using Jython [32], a Java implementation of Python. With this choice, we intend to highlight that retro-compatibility functions are useful but optional, and that they can be implemented as distributed services, uncoupled from the main S×C4IoT core framework.

5.1 Parsed Contract

As previously mentioned, MUD files exhibit metadata that can be mapped to S×C fields. For example, MUD files store the domains that might be used by IoT devices in the *access-lists* field. After we analysed MUD files' structure, we determined that we can reliably extract the following information: ports used for communication (grouped by LAN/internet and local/remote) and internet domains the device can communicate with (grouped by inbound/outbound communications). However, this information is not enough for creating an S×C contract—more steps are necessary.

MUD files might contain information that cannot be translated directly to a S×C contract, but that can help to gather necessary data from other sources. Our device profiling combines MUD metadata, **access control list (ACL)** analysis, **dynamic host configuration protocol (DHCP)** fingerprints, and queries to the Fingerbank **application programming interface (API)**. Our solution [44] is shown in Figure 4 and consists in the following steps:

(1) **Receive DHCP DISCOVER request containing MUD URL**
(2) **Extract DHCP fingerprint, user-agent string, media access control (MAC) address, and MUD URL** This info is included in the DHCP DISCOVER packet.
(3) **Query the Fingerbank API for manufacturer and device names using the extracted client information** The API accepts a DHCP fingerprint, a user-agent string, and a MAC address.
(4) **Retrieve MUD file from extracted URL and extract the ACLs** We made some base assumptions about the ACL data: (1) each local port represents a provided service and (2) each remote port represents a required service.
(5) **Generate an ACL profile object using manufacturer name, device name, and ACLs** We defined a data model class, the ACL profile, to encapsulate relevant ACL information, along with the functionality to instantiate such objects from raw ACL data. The protocols

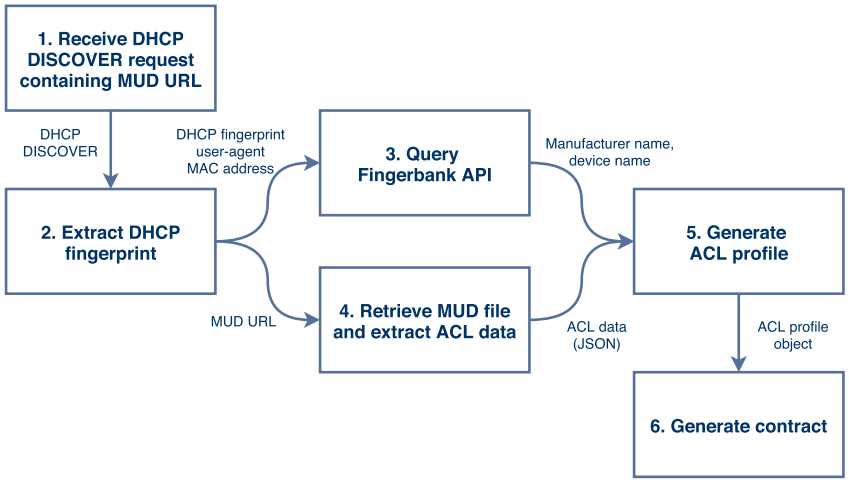

Fig. 4. This workflow shows our proposed approach for extracting S×C contracts from MUD ACLs.

used are also available for extraction, but it is unclear how they would help in constructing contracts for S×C, so they were not included in this model.

(6) **Generate a contract object using the ACL profile object** We defined a second data model class, the contract, to represent a security contract, complete with functionality to instantiate contracts from ACL profile objects.

Part of our solution relies upon obtaining devices' DHCP fingerprints, feeding them to the Fingerprint API, and obtaining the pair device/manufacturer. However, online there are few DHCP fingerprints for the MUD devices we used in our experiment, and we did not have access to the physical devices for extracting our own fingerprints. If we assume that Fingerbank APIs can reliably recognise a device from its DHCP fingerprint, then we can circumvent the issue. Fingerbank assigns unique IDs to the stored devices: We can manually find such IDs, use them for querying Fingerbank, and then extract the pair device/manufacturer.

Evaluation. We ran our algorithm for 28 MUD files, provided by Hamza et al. [31]. Figure 5 shows an output example for a Belkin camera device; the raw test output can be found on the GitHub repository [44]. The generated partial contract represents the following communication profile:

- Camera communicates on port 5104, over both LAN and Internet;
- Camera communicates on ports 53, 67, and 3478, and transmits to remote port 1900 over LAN;
- Camera communicates on ports 8899, 123, 3475, 8443, and 443, over Internet.

The basic ACL data contains information about the internet domains contacted, as well as the protocols used. For the sake of simplicity, we excluded the protocols from the basic contract model. In Appendix B, we present the results in a condensed format, where the domain names identified are omitted. Instead, for each device, we report the number of domains we extracted from the MUD ACLs. We also report the identified ports used by each device for inbound and outbound communication, both on the local network and the internet. This data is necessary, but not sufficient, for the details required by S×C contracts. Indeed, MUD descriptions do not carry enough information for producing complete S×C contracts. However, our experiment shows that it is possible to parse

```
Device type:                Camera
Classification score:       0.17817759870529015
Name:                       Belkin.NetCam
Mud file ACLs:
Security contract Belkin.NetCam:
        Rule Belkin.NetCam.all:
                Device:     Belkin.NetCam
                Domain:     *
                Shares:     *
                Provides:   {'5104'}
                Invokes:    {'5104'}
        Rule Belkin.NetCam.lan:
                Device:     Belkin.NetCam
                Domain:     LAN
                Shares:     *
                Provides:   {'67', '3478', '53'}
                Invokes:    {'67', '3478', '1900', '53'}
        Rule Belkin.NetCam.net:
                Device:     Belkin.NetCam
                Domain:     Internet
                Shares:     *
                Provides:   ['443', '8443', '8899', '123', '3475']
                Invokes:    ['443', '8443', '8899', '123', '3475']

Contacted domains:
        nat.xbcs.net
        api.xbcs.net
        [..]
```

Fig. 5. Test output for a Belkin Camera device.

MUD files in partial S×C contracts. With this approach, it is possible to create a high-level contract that defines if a device communicates over LAN, Internet, or both.

5.2 Device-based Contract

In the previous section, we have shown that it is possible to create a partial S×C contract from a MUD file. The parser technique presumes that the device is MUD-compliant, that the MUD file contains enough metadata, and that a DHCP fingerprint is available for the device. However, these assumptions might not hold and a fallback solution is of paramount importance.

Our second solution [59] supposes that similar devices have similar requirements and constraints. Let us suppose that we set up a smart home; it is reasonable to expect that all our security cameras should behave in a similar way. As an example, we might want to prevent the cameras to communicate outside the LAN domain for privacy reasons. We can do so by assigning a predefined *camera contract* C_{Camera} to each camera, but this raises a categorisation problem. We need to be able to tell whether an IoT device is a security camera or another type of device.

To quickly identify IoT devices, network traffic analysis cannot be used: Capturing and analysing pcap traces could require hours of unrestricted interaction. This is not desirable if we intend to use the classification for enforcing network security policies as soon as a device joins the network. Our proposed device classification is split in two distinct parts: the information retrieval and the text classification. The information retrieval part (phase 1 and phase 2, in red, in Figure 6) is responsible

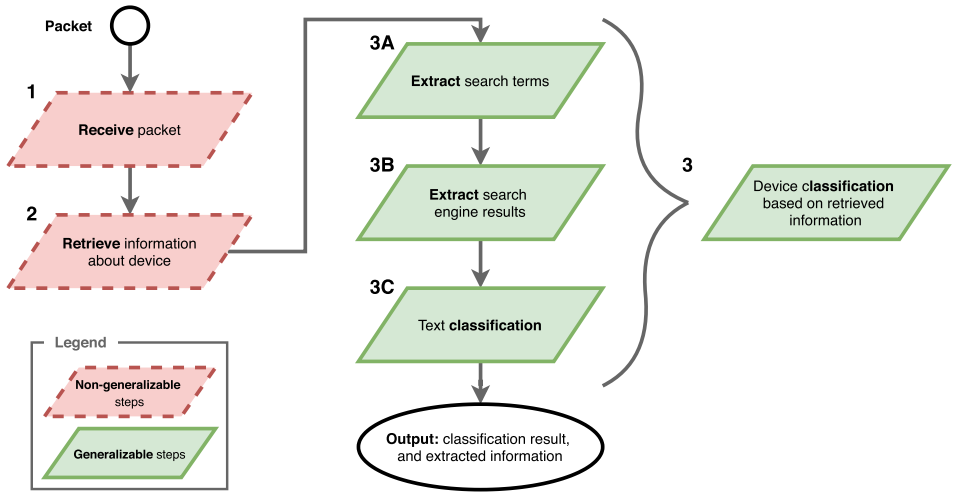

Fig. 6. Overview of a device profiling steps. The red steps cannot be generalised, since they depend on the specific protocol stack. The green steps, all part of the text classification phase, are generalisable.

for extracting relevant texts from the device fields. Since this part depends on the IoT protocol stack, it cannot be generalised for every IoT device. Phase 3, coloured in green in Figure 6, is responsible for extracting relevant keywords from the texts (phase 3A), feeding them to web search engines and extracting the search results (phase 3B), and finally classifying the device (phase 3C).

The computing demand for gathering relevant information and classifying the device type might vary, depending on the complexity of the task. In our vision, the classification algorithm would run on a Fog node service. Due to its trustworthiness, computing power, and capability of offloading tasks to the Cloud layer [39, 52], the Fog node is a critical component in the S×C framework. Running as a Fog service, our solution would have the flexibility to run either on the Fog node layer or on the Cloud layer (we refer the reader to the architectural model in Reference [39]). If we assume that the Fog node stores the S×C contracts for every device category, then it is trivial to assign to a device the corresponding S×C generic contract once the classification algorithm categorises the device.

Proof of Concept. Taking Figure 6 into consideration, the initial steps cannot be generalised for consumer devices and industrial ones. Regarding the information retrieval step, we narrow down our scope and choose relevant standards and protocols. For general consumer IoT devices, we focus on the MUD standard. For industrial IoT devices, we take into consideration the BACnet protocol. The overall approach for extracting information is therefore different.

There are differences in which keywords can be extracted from MUD files and BACnet objects. MUD does not specify what information *systeminfo* should contain, whereas BACnet *device object* contains specific information about vendor and device model. For this implementation, both the data extracted from the MUD files and the BACnet objects is simply used as is, without any attempt to discard irrelevant text from the field. We refer the reader to References [58, 59] for more details.

To gather as much information as possible, we feed the extracted keywords to Google and Bing. Our goal is to increase the amount of information at our disposal so we improve our chances of identifying the devices. Initially, we scraped the websites and ran our feature extraction algorithm on the obtained data. We called this approach **Cumulative Squared Scoring (CSS)**. However, scraping a website could infringe the **Terms of Service (TOS)** of a website and lead to legal

Table 2. Device Categories Used for Testing

General Devices		**BACnet**
Alarm	Speaker	Industry Environment Sensor
Camera	Light	Industry Controller
Doorbell	Media Player	
Electric Control	Motion Sensor	
Health Monitor	House Environment Monitor	

consequences. To minimise this eventuality, instead of scraping the websites by ourselves, we used the websites summary snippets provided by the search engines. Once we obtain the relevant texts, we can start the classification phase.

We have conducted a number of experiments to identify the best combination of feature extraction and classification algorithms. After our experiments, we opted for a combination of **Term Frequency-Inverse Document Frequency (TF-IDF)**, and **Support Vector Machines (SVM)**. In particular, SVMs perform well if the feature vectors have high dimensionality, which is the case for text classification [50]. Therefore, we opted for SVM, in the form of the LibSVM library. We refer the reader to Reference [59] for the experimental results that led to our decision.

Evaluation. After we have chosen the appropriate feature extraction and text classification algorithms, we defined a set of tests. Our goal is to verify that our approach can identify correctly MUD and BACnet devices. First, we created 2 different training datasets to train the classification models. One dataset contains texts for 10 categories of general consumer IoT devices. The other dataset contains the previous 10 categories of texts, plus 2 categories specific for industrial IoT devices. In Table 2, we list all 12 categories.

Additionally, we created 2 testing datasets. The first contains 24 general consumer IoT devices, in the form of MUD files generated by Hamza et al. [31]. The second contains the first dataset items, plus 9 industrial IoT devices in the form of BACnet objects. This led to the following tests:

MUD -: Testing MUD devices. Training dataset contains only texts and categories of general consumer IoT devices.
MUD +: Testing MUD devices. Training dataset contains texts and categories of both general consumer and industrial IoT devices.
BACnet +: Testing BACnet devices. Training dataset contains texts and categories of both general consumer and industrial IoT devices.

The MUD - test evaluates how well the implementation performs when we consider only MUD devices and general consumer devices. As an extension, with MUD +, we evaluated the same set of IoT devices, but we expanded the categories to include industrial IoT devices. It is worth noting that we did not perform BACnet - tests. We only have two categories for industrial IoT devices, so the resulting training dataset would be too small to be reliable. Last, in BACnet +, we tried to identify BACnet- enabled devices using the same categories we used for MUD +.

Table 3 shows the results obtained. First, regarding the accuracy, the TOS approach achieved better or equal results for all three tests. In particular, with the MUD - test the classification accuracy improved from 88% to 96%. In the table, we also show the rate of devices that could not be classified, because they fell between two different categories, consequently leading to indecision. This result stayed stable for MUD + and BACnet +, and improved for MUD -, decreasing from 4% to 0%. Last, but not least, the average classification score improved for all tests, showing that the snippet-based approach (i.e., TOS) consistently produces more robust results than the scraping-based one, CSS.

Table 3. Denoted with CSS, Results Obtained with CumuLative Scoring and a Cleaned Dataset

CSS	MUD -	MUD +	BACnet +
Accuracy	0.88	0.88	0.89
No Classification	0.04	0.04	0.00
Average Classification Score	1.17	1.09	1.22
TOS			
Accuracy	0.96	0.88	0.89
No Classification	0.00	0.04	0.00
Average Classification Score	2.53	1.82	1.63

Denoted with TOS, results obtained with search engines snippets, cumulative scoring, and a cleaned dataset.

Table 4. Average Time Required for Classifying Devices with CSS and TOS

Device	CSS (s)	TOS (s)
MUD	20.24	2.78
BACnet	48.30	7.62

For this result, we give credit to the relevancy and conciseness of the snippets returned by the search engines. Both Bing and Google deploy advanced machine learning algorithms, trained to extract the most relevant words from long texts. The results we obtained suggest that the snippets accurately portray the websites' contents, providing relevant text and very little noise.

Beyond allowing to comply with websites' terms of service, and improving algorithmic accuracy, snippets also sped up the performance of our proof-of-concept. Table 4 shows the average time required for identifying the devices with CSS and with TOS, respectively. The average runtime for identifying MUD devices is 10× shorter, while for BACnet devices is 7× shorter. Appendix C lists in details which MUD and BACnet devices were successfully identified and which ones not.

6 DYNAMIC EVOLUTION

In our previous work, we have defined the foundations for an IoT S×C framework. In particular, first, we have defined the life-cycles for contracts and policies. Second, we have defined the consistency rules that allow S×C to verify a contract against a policy and avoid potential uncertainties. However, defining how S×C components singularly evolve, and under which conditions they can cooperate, is not enough. Evolution within networks happens on different levels, from the network composition itself to the life-cycle of each single device.

Networks, and in particular IoT networks, can be highly variable within a short span of time. Trends like ***bring your own device*** **(BYOD)** are perfect examples of such variability. Day after day, organisational networks witness dozens of different devices coming and going. On arrival, employees connect their laptops through an Ethernet cable and their smartphones and personal wearables (such as smartwatches) through WiFi. Throughout the working day, the same devices can be rebooted or displaced, temporarily losing connectivity. And when employees leave for home, they disconnect everything from their office network, they carry the devices with them, and then connect the same devices to their personal network once they are back home.

Not only network composition can vary, but also devices behaviour can change throughout time. This can happen for different reasons, ranging from legitimate updates to malfunctions, from malicious tampering to programmed reactions to uncertain events. Regardless of the reason, an IoT network should be able to deal consistently with these events. In this section, as shown in Figure 1, Stage 6, we address the matter of dynamic evolution both from the point of view of legitimate and malicious behavioural evolution.

In particular, as previously discussed in Section 3.2, in this section, we discuss the Fog-actionable events that S×C4IoT is capable of detecting and managing. For each dynamic evolution, we provide

Table 5. Examples of Dynamic Evolution Managed by S×C4IoT

Example	Action	Contract	PoC	Compliance with Policy	Result
Example 1	Add Device to Network	Extracted	Invalid	Compliant	Correctly added ✓
Example 2	Add New Contract	Verified	Valid	Non-compliant	Correctly not added ✓
Example 3	Update Software	Verified	Invalid	Compliant	Correctly not updated ✓
Example 4, Example 5	Update Contract	Verified	Valid	Compliant (Could remove critical services)	Correctly updated but might cause cooperation issues ~
Example 6	Contract/Software Update	*Non-S×C*	*Non-S×C*	Compliant/Non-compliant	Correctly updated but might cause cooperation issues ~
Example 7	Remove Device	Verified /Extracted	Valid /Invalid	Compliant (Could remove critical services)	Correctly prevented from removing ✓

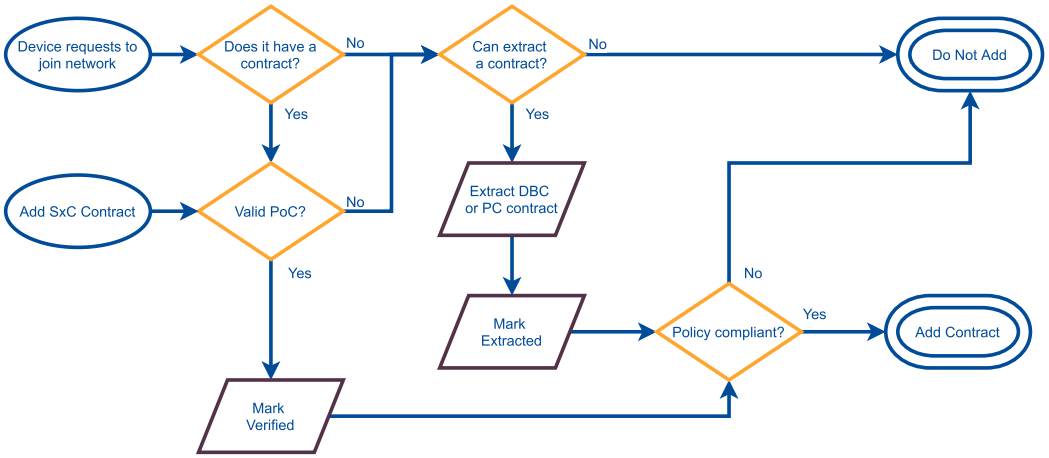

Fig. 7. Workflow for adding a new device to an S×C network and for adding an S×C Contract to a device without a previous contract.

a relevant example that demonstrates how S×C4IoT manages the situation and, for improving the article readability, we sum up these examples in Table 5.

6.1 Add Device

In Figure 7, we show how S×C4IoT reacts to a new device joining a S×C network. First, we have to verify if the device has a valid contract, together with a valid PoC. If both of them are valid, then the contract is marked *verified*. If one of the two is not valid, then the device information is passed to the DBC analyser, which tries to extract an S×C contract. If DBC succeeds, then the resulting contract is marked *extracted*; if it fails to extract a contract, then the device is simply rejected. Whether a contract is marked verified or extracted, it has to be verified against the S×C security policy. If the contract is consistent with the policy, then it is accepted to join the network, else it is rejected.

In S×C4IoT, we divide proper S×C contracts from DBC contracts. In principle, S×C contracts are more trustworthy than contracts derived from DBC routines. In case a proper S×C contract is involved, it should take precedence over a DBC one; this is why it is important to distinguish between sources. Later in this section, we show how this flag enables us to achieve our goal.

Example 1. Let us suppose a new device $D_{OORT.Lock}$ tries to join our S×C network. $D_{OORT.Lock}$ has a contract C_{Lock} (Table 6), but the PoC is *invalid*. According to our workflow, we scan $D_{OORT.Lock}$

Table 6. According to Example 1, Contract C_{Lock} (Composed of a Single Rule R_{Lock}) Does Not Have a Valid PoC

	Rule R_{Lock}	**Rule $R_{Lock\_Extracted}$**
D	OORT.Lock	OORT.Lock
Dom	LAN	LAN
Shares	Apple.LukePhone	Apple.LukePhone
Provides	OpenCloseBluetooth, OpenCloseHTTP	OpenCloseHTTP
Invokes	-	-

Therefore, we extract contract $C_{Lock\_Extracted}$, and mark it extracted. Similar to C_{Lock}, $C_{Lock\_Extracted}$ is composed of a single rule $R_{Lock\_Extracted}$.

Table 7. P_A of Example 1 Is Composed of Three Rules

	Rule R_1	**Rule R_2**	**Rule R_3**
D	OORT.*	Samsung.Hub	Samsung.Sensor
Dom	LAN	LAN	LAN
Shares	*.*	Samsung.Sensor	Samsung.Hub
Provides	-	OnOff	OpenClose
Invokes	-	Samsung.Sensor. OpenClose	-

The first one allows every device from OORT manufacturer to communicate over LAN. Second and third rules simply shape other devices' behaviour. Contract $C_{Lock\_Extracted}$ matches policy P_A, thus OORT. Lock can join the network.

with DBC, derive a contract $C_{Lock\_Extracted}$ and mark it extracted. In this example, $C_{Lock\_Extracted}$ is different from C_{Lock}, but DBC can potentially extract exactly the same contract. Last, we verify $C_{Lock\_Extracted}$ against policy P_A shown in Table 7. $D_{OORT.Lock}$ is consistent with the network security policy P_A (in particular, it does not contradict R_1) and can safely join the network.

6.2 Add Contract

Another case occurs when a device already present in the network does not have an S×C contract and it gets one. This case, shown in Figure 7, applies to legacy devices already present in the network at the time of deployment of the S×C framework.

First the triplet *<Software, Contract, PoC>*is evaluated. If the new contract is valid and the PoC consistent, then the contract is marked verified and the device is S×C-compliant. Else, the S×C framework tries to derive a contract with DBC and, if successful, marks the resulting contract as extracted. Last, as it is done in the case of a new device joining a network, the device is accepted or rejected based on its compliance to the network security policy.

Example 2. Let us suppose a legacy device Philips.HueWhite is part of an S×C network, and that is not equipped with an S×C contract. The manufacturer decides to comply with S×C specifications and rolls out a new contract $C_{HueWhite}$ for Philips.HueWhite, as shown in Table 8. Assuming that the manufacturer attaches a valid PoC, $C_{HueWhite}$ is marked verified and undergoes the matching phase with the network security policy P_B (Table 9). However, R_{B1} in $C_{HueWhite}$ and R_{D1} in P_B would not be core rules, since $R_{B1}[\text{Provides}] \subseteq R_{D1}[\text{Provides}]$.

Table 8. Security Contract $C_{HueWhite}$

	Rule R_{B1}	**Rule R_{B2}**
D	Philips.HueWhite	Philips.HueWhite
Dom	LAN	LAN
Shares	Philips.*	*.*
Provides	On, Bri	On
Invokes	Philips.HueMotion.Presence	-

Table 9. Security Policy P_B

	Rule R_{C1}	**Rule R_{D1}**	**Rule R_{D2}**
D	D-Link.933L	Philips.HueWhite	Philips.HueWhite
Dom	*	LAN	LAN
Shares	Apple.LukePhone	Philips.*	*.*
Provides	SetDayNight	On, Bri, Hue	On
Invokes	-	Philips.HueMotion.Presence	-

Even though $C_{HueWhite}$ and PoC are valid, $C_{HueWhite}$ does not comply with P_B, since it violates the core rule principle. The legacy device is not allowed to participate with the S×C-compliant network.

6.3 Update Software

In the two previous cases, we have shown how we deal with new devices trying to join a network and with devices loading a contract for the first time. The life-cycle of a device does not necessarily stop after a user buys and installs it. Manufacturers roll out updates for their devices, sometimes for fixing bugs, other times for adding features. This entails that the behaviour of a device can change over time, while being part of a network. After an update, a device might not match the network policy anymore; it might require a service that is not currently provided within the network; it might have withdrawn a service that is necessary for other devices to operate. Besides, new (or removed) functionalities must be reflected in an updated contract. Most likely, an updated contract would be almost identical to the previous one, which means that they would clash with each other. However, the new contract is supposed to *replace* the old one, thus it does not make sense to evaluate a new contract against a policy containing its older version. In Figure 8, we show how we face software changes that naturally occur throughout a device life-cycle.

Example 3. We take as an example the device $D_{HueWhite}$, which carries a verified contract $C_{HueWhite}$ shown in Table 8. Let us suppose that $D_{HueWhite}$ receives an update notification and downloads the triplet <*Software, Contract, PoC*>. However, during the download phase, an error occurs and the PoC corrupts. Following the workflow in Figure 8, we analyse the previous contract and we confirm that it was verified. Consequently, we do not attempt to extract a (potentially) imprecise contract with DBC and we simply abort the device update. In the following days, the device will be able to retry the update and, eventually, the attempt will be successful.

6.4 Update Contract

Not only manufacturers can update their devices software. In the case of S×C-compliant devices, they can also update the contracts to provide more precise behavioural descriptions. How do we manage this dynamic evolution? The approach (shown in Figure 8) is similar to the one used for

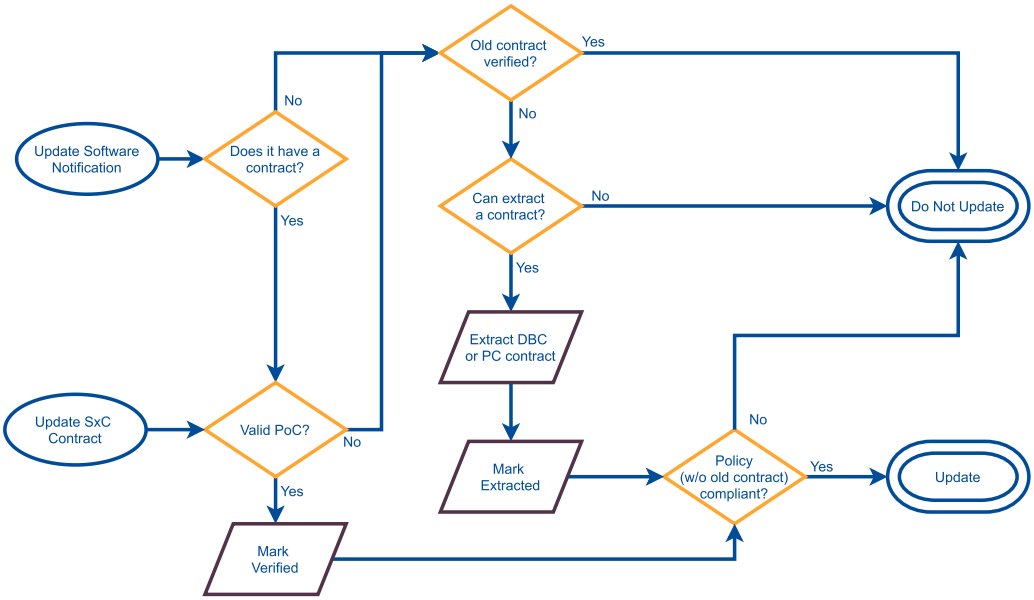

Fig. 8. Workflow for updating the software or a contract of a device already part of an S×C network.

Table 10. Contract $C_{HueMotion}$ before Update and Contract $C_{HueMotion2}$ after Update

	Rule R_M
D	Philips.HueMotion
Dom	LAN
Shares	Philips.*
Provides	On, Presence
Invokes	-

	Rule R_{M2}
D	Philips.HueMotion
Dom	LAN
Shares	Philips.*
Provides	Presence
Invokes	-

addressing software updates, with a main difference. In this case, we are updating a preexisting contract, so it is not necessary to verify if a device comes equipped with another contract.

Example 4. Let us suppose we have a device $D_{HueMotion}$ that stores $C_{HueMotion}$ (Table 10). $D_{HueMotion}$ is part of an S×C network equipped with the policy P_C, shown in Table 11. Contract $C_{HueMotion}$ is then updated with a valid contract $C_{HueMotion2}$.

$D_{HueMotion}$ software is valid and intact, as well as the PoC, therefore the contract is marked verified and can undergo the matching phase. As aforementioned, if we verify $C_{HueMotion2}$ against P_C as is, then rules R_M and R_{M2} would not be core rules any more. $C_{HueMotion2}$ is intended to eventually substitute $C_{HueMotion}$, therefore, we need to evaluate it against P_C without $C_{HueMotion}$ ($P_C \backslash \{R_M\}$).

In this case, $C_{HueMotion2}$ removes service On, which is required by device Philips.HueWhite. Even though this could potentially cause a disservice, $C_{HueMotion2}$ is compliant with P_C, so the update is allowed.

It is important to highlight that, as a design choice, we allow contract (and software) updates that might remove services that are required from other devices. The rationale behind this choice is that circular dependencies could happen between cooperating IoT devices. Such devices might

Table 11. Security Policy P_C

	Rule R_{DL}	**Rule R_W**	**Rule R_M**
D	D-Link.933L	Philips.HueWhite	Philips.HueMotion
Dom	*	LAN	LAN
Shares	Apple.LukePhone	Philips.*	Philips.*
Provides	SetDayNight	On, Bri, Hue	On, Presence
Invokes	-	Philips.HueMotion.On, Philips. HueMotion.Presence	-

Table 12. Security Policy $P_{Equilibrium}$

	Rule R_{WA}	**Rule R_{MA}**
D	Philips.HueWhite	Philips.HueMotion
Dom	LAN	LAN
Shares	Philips.*	Philips.*
Provides	On, Bri, Hue	On, Presence
Invokes	Philips.HueMotion.On, Philips. HueMotion.Presence	Philips.HueWhite.Hue

receive updates that change the cooperation terms without breaking their equilibrium. However, the single devices updates are likely to happen sequentially, leading to a temporary state of non-equilibrium. Preventing a device from updating its contract/software might lead to a permanent deadlock state, where no device will ever be updated.

Example 5. Assume a simple policy $P_{Equilibrium}$ consisting of two rules, R_{WA} for Philips. HueWhite and R_{MA} Philips. HueMotion. As shown in Table 12, these two devices can use each other's services. R_{WA}Invokes contains Philips. HueMotion.On and Philips.HueMotion. Presence, while R_{MA}Invokes contains Philips. HueWhite.Hue. At a certain point, the manufacturer updates these two devices, removing service Philips.HueMotion. Presence and service Philips.HueWhite.Hue, which entails that $P_{Equilibrium}$ would become $P_{FinalEquilibrium}$, as shown in Table 13.

It is assumed that the two devices will be updated sequentially, since simultaneous updates would require complex ad hoc routines (e.g., waiting pools for synchronising updates). Therefore, regardless of the final result, the system will go through a provisional non-equilibrium phase. For example, if we choose to update rule R_{WA} to R_{WB}, then we would lose service Philips.HueWhite.Hue that is still required by R_{MA}[Invokes], as shown in Table 14. The same would happen if R_{MA} were updated before R_{WA}. R_{MA}[Provides] would not provide any more service Presence, which is present in R_{WA}[Invokes].

While this approach prevents deadlocks, a device might receive an update that removes from its Providelist a service that is a critical service for its users or other devices. It is worth noting that this is not a problem introduced by our framework but an inherent problem that comes with automatic updates. There are different approaches to prevent this from happening, such as introducing a waiting pool that is manually managed by the network administrator; enriching the contracts with a list of optional invoked services and critical ones; preventing an update if the new contract removes services that were listed in the previous contract and that are used by other devices. However, it must be taken into account that a manufacturer might have compelling security reasons

Table 13. Security Policy $P_{FinalEquilibrium}$

	Rule R_{WB}	**Rule R_{MB}**
D	Philips.HueWhite	Philips.HueMotion
Dom	LAN	LAN
Shares	Philips.*	Philips.*
Provides	On, Bri	On
Invokes	Philips.HueMotion.On	-

Table 14. Security Policy $P_{Non-equilibrium}$ (Temporary Non-equilibrium)

	Rule R_{WB}	**Rule R_{MA}**
D	Philips.HueWhite	Philips.HueMotion
Dom	LAN	LAN
Shares	Philips.*	Philips.*
Provides	On, Bri	On, Presence
Invokes	Philips.HueMotion.On	Philips.HueWhite.Hue

Table 15. Security Contract $C_{OORTLock}$

	Rule R_{L1}
D	OORT.Lock
Dom	*
Shares	Apple.LukePhone
Provides	OpenClose
Invokes	-

Table 16. Security contract $C_{OORTLock}$

	Rule R_{L2}
D	OORT.Lock
Dom	LAN
Shares	Apple.LukePhone
Provides	OpenClose
Invokes	-

to remove a service from a device's Provideslist, so it is reasonable to allow an update that does not explicitly violate a security policy. At the same time, this might create some issues, as shown in the following example.

Example 6. Let us assume an OORT Smart Lock, which enables the owner to open and close his house door from any network. This behaviour can be translated in Rule R_{L1}, as shown in Table 15.

Suppose also that, for safety reasons, the manufacturer decides to restrict this service to the LAN domain. Consequently, the manufacturer rolls out an update that modifies R_{L1} in R_{L2}, as shown in Table 16. However, this update happens while the owner is away from home. When he comes back home, he tries to unlock the front door with his Internet-connected smartphone and fails.

6.5 Remove Device

The last case of dynamic evolution happens when a device tries to leave the S×C network. If such a device does not provide critical services for other devices' operations, then it is safe to assume that it can leave the network without causing any disservice. However, a critical device might try to leave the network, triggering a chain reaction of non-functioning services. In this case, as shown in Figure 9, the administrator should immediately receive a warning that highlights that

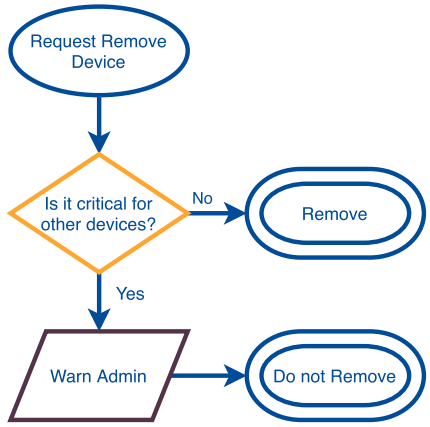

Fig. 9. Workflow for removing a device from an S×C network.

Table 17. Security Policy P_D

	Rule R_W	**Rule R_M**
D	PHILIPS.HUEWHITE	PHILIPS.HUEMOTION
DOM	LAN	LAN
SHARES	PHILIPS.*	PHILIPS.*
PROVIDES	ON, BRI, HUE	PRESENCE
INVOKES	PHILIPS.HUEMOTION.PRESENCE	-

a critical device is about to leave the network. Then, if possible, the device should be prevented from leaving.

In our workflow, we assume that the removal attempt comes from an explicit will of the users to remove a device from their network. We also assume that the removal phase is not unilateral, but it happens as a negotiation between the device and the S×C network. The workflow can be partially applicable even when the device abruptly and unilaterally leaves the network, such as when it runs out of energy or faces a hardware failure. In such cases, we cannot prevent the device from leaving the network, but we can still raise a warning for the administrator.

Example 7. Assume two devices in an S×C network, $D_{HueWhite}$ and $D_{HueMotion}$, which contracts contain, respectively, R_W and R_M, as shown in Table 17. Let us suppose that $D_{HueMotion}$ tries to leave the network. According to P_D (Table 17), $D_{HueMotion}$ provides the service PRESENCE, which is required by $D_{HueWhite}$ to function. Therefore, following the workflow we depicted in Figure 9, the S×C system raises a warning for the administrator and keeps $D_{HueMotion}$ in the network.

6.6 Architecture and Implementation

As previously mentioned in Section 5, we implemented our software in separate proof-of-concepts. In this section, we describe the core S×C framework. Written in Java, this part manages the S×C routines shown in Section 4, as well as the possible dynamic changes depicted in Section 6. Referring to Figure 1, this section covers Phase 2A, Phase 3, Phase 4, Phase 5, and Phase 6.

Our Java implementation includes the fundamental components of S×C4IoT. We have defined a Java class for each S×C component from the basic Service up to the Policy. Besides the components,

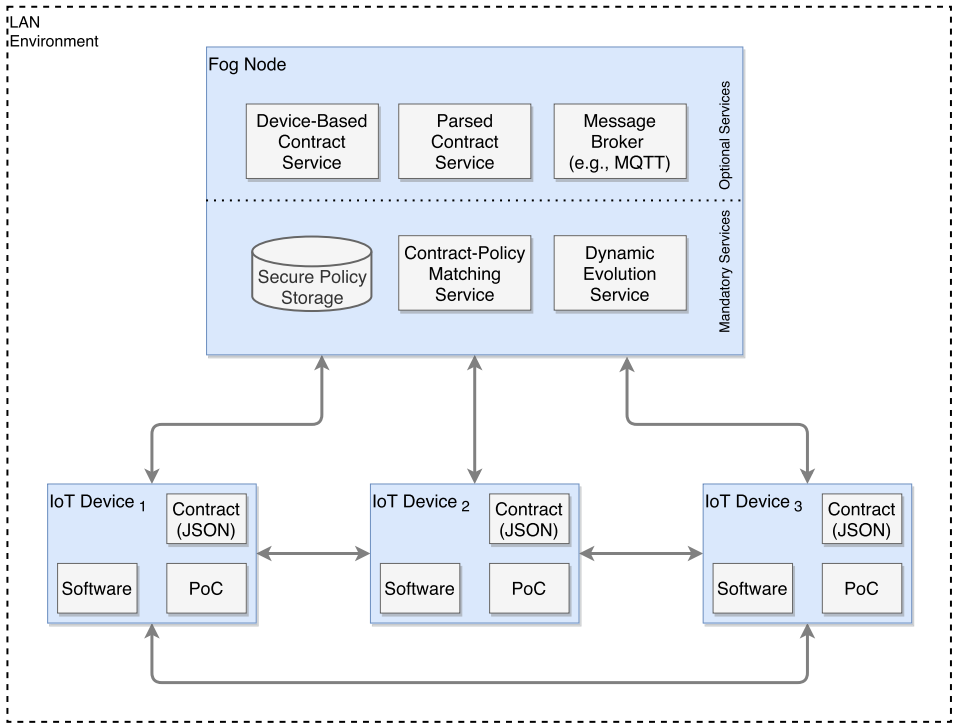

Fig. 10. Architectural diagram of the S×C4IoT proof-of-concept that we developed full stack. The optional services could be either hosted by the Fog node, as depicted, or offered as external services (either within the LAN or on the Cloud).

we also defined the necessary functions defined in Section 4. For example, the Java class Rule not only includes the semantics for S×C security rules, but also the functions for verifying if a rule is well formed or core [24].

The architectural diagram in Figure 10 shows the architecture of the proof-of-concept we developed full-stack. Each IoT device is a virtualized object that contains the triplet *<Software, Contract, PoC>*. For the scope of this article, IoT devices' onboard software and PoC are not implemented: deploying a fully functional and correct PoC is a hard task, non-critical for the goal of this virtualized proof-of-concept. Therefore, we replaced these components with appropriate stub flags and methods, which allow to achieve a fully operating proof-of-concept. In Section 8, we discuss the limitations of this choice, as well as our future plans.

Regarding the contract, we opted for serialising it with **JavaScript Object Notation (JSON)**, a data interchange format and ISO standard (ISO/IEC 21778) [10]. JSON is well-fit for this task, because it produces human-readable files while minimising the overhead. In addition, JSON files can be easily deserialised into Java objects using libraries, such as GSON [28]. This allows to use JSON files for two different scopes: as a configuration language for creating Java contracts and as an exchanging format for transmitting contracts between devices and Fog nodes.

The Fog node has been also implemented as a Java object. Due to its properties of trustworthiness and computing power capabilities, this object is equipped with different mandatory and optional services. First, the Fog node stores the network security policy. As shown in Section 3.2, we assume that adversaries cannot tamper with the Fog node and the stored policy. The second

mandatory service is the contract-policy matching one, which has the critical role of evaluating if a device contract complies or clashes with the network security policy. Alongside the contract-policy matching, the Fog node offers the third mandatory service module, the dynamic evolution. Not only does this module include the services for managing IoT devices dynamic evolution shown in Section 6, but it also allows the Fog node to evaluate the validity of triplets *<Software, Contract, PoC>*, and if a contract has been marked verified or extracted.

As optional services, we include the services PC and DBC (described in Section 5), which allow a legacy device to join an S×C network. These two services are fit for the computing capability of a Fog node, but they might also be hosted by another network device or offered as **software as a service (SaaS)**. As a third optional service, we envision a message broker. As mentioned for DBC and PC service, assigning this task to the Fog node is appropriate, but it is not mandatory. For instance, a network might rely on communication paradigms that do not require publish/subscribe brokers, or the broker might be hosted by another dedicated device. Hosting the broker on the Fog node allows for a tighter monitoring but, at the same time, increases its workload.

Regarding communication protocols, for our proof-of-concept, we opted for **Message Queuing Telemetry Transport (MQTT)**, a lightweight publish/subscribe protocol. MQTT is an open OASIS and ISO standard (ISO/IEC 20922) [9] that, due to its characteristics, is a popular choice for IoT applications [37]. In particular, we used an open source implementation of MQTT, Mosquitto [22]. In our implementation, devices exchange information in the form of Messages, Java objects that are serialised as JSON objects, forwarded, and deserialised back to a Message object. MQTT publish/subscribe protocol allows for one-to-one communication with proper **access control lists (ACLs)**. Clients can access to topics with three different permissions: read (r), write (w), or read and write (r/w). An administrator can restrict devices to read only specific topics and write (without reading) to other selected topics. Setting up a local MQTT broker with an ACL is easy, if required.

For the scope of this article, we used a private local broker. A remote public broker would introduce a high degree of uncertainty, given that traffic and load balancing depend on third parties utilising the same broker. This would make it hard to distinguish the overhead introduced by our solution and the one introduced by regular network fluctuations. Regarding the ACL, we decided to skip such functionality, because we did not have a precise idea about the potential influence on performance. In our proof-of-concept, each device listens to an *[ID]/in* topic, where [ID] is a unique number that identifies an IoT device. To communicate with each other, devices send their messages to the recipients' *[ID]/in* topics. In principle, anyone could subscribe to such topics and read the messages. This would not be desirable in a real application; we advise setting up proper ACLs for production MQTT brokers, as previously discussed. However, to evaluate precisely our proof-of-concept, we prioritised minimal overhead over ideal broker configuration.

7 EXPERIMENTS AND DOMAIN LIMITS

In this section, we show the experiments we ran on the third part of our proof-of-concept, and we provide notable case studies that show how S×C4IoT helps to address default insecure configurations and insufficient security configurability. In addition, we discuss the malicious behaviour that S×C4IoT contributes to prevent and mitigate, and the domain limits within our solution works.

7.1 Unit Tests

First, we wrote extensive unit tests for the classes that compose our Java proof-of-concept. We took into consideration scenarios compatible with S×C4IoT specification, but also scenarios that violate such specifications. This allows us to verify that the framework works as expected. Together with the source code, we also published the unit tests on our repository [24]. In Table 18, we provide

Table 18. S×C4IoT Classes - Unit Tests Overview

Class Tested	Number of Tests	Passed
Device	7	7 ✓
Service	5	5 ✓
Rule	10	10 ✓
Contract	16	16 ✓
Policy	8	8 ✓
FogNode	8	8 ✓
IoTDev	8	8 ✓

an overview on the number of unit tests that we created for each class, while in Appendix D, we give an in-depth insight regarding the scenarios that we evaluated.

For instance, the class Device implements the object used for defining the field D inside of a rule and is the base unit for the object SHARES, which is in fact a list of different devices (as per definition in Section 4). Important unit tests include the assurance that the symbol * is managed correctly as a token for *any*; therefore, we verify that APPLE.PHONE $\subseteq *.*$, but that APPLE.$* \nsubseteq$ APPLE.PHONE.

Regarding the base unit of S×C, the Rule, we implement unit tests for verifying if rules are core and well formed. We feed a malformed rule R_{MA} to the method isWELLFORMED and we verify that, as expected, the method returns False. Similarly, we verify that the method isCORE returns true when we verify a rule against an empty set, we confirm that the consistent contract C_B is composed of core rules, and that the inconsistent contract C_{IB} is not, since it stores R_{B3} that restricts R_{B2}.

We also implemented a virtual Fog Node and we verified that it manages contracts, policies, and dynamic evolution cases, as defined in Section 6. For example, we verified that an inconsistent contract C_{IB} is not added to a Policy P_A (thus, the device carrying C_{IB} does not join the network). We verified that a consistent C_B is correctly added to an empty policy P_{Empty} as well as to P_A, as expected. Additionally, we verified that the same contract is correctly removed from the network, when required, and that any rule belonging to such contract is wiped. We performed these tests with different contracts to ensure that our tests hold against different cases.

7.2 Impact of S×C Contracts

Apart from verifying the correctness of our solution, we also wanted to evaluate its potential impact. Referring to Figure 7 and Figure 8, first, we must identify which phases might have the most relevant impact on network performance.

Assessing if a device has a contract, and if it is verified or extracted, is straightforward. For the reasons mentioned in Section 6.6, we did not implement a proper PoC for our virtual IoT devices, and we substituted this component with a Boolean flag and appropriate stub methods. For example, the function hasValidPoc verifies if an IoT device has a valid PoC or not; in of our proof-of-concept, this equals to a check of a Boolean flag. By definition, a PoC should be hard to create but easy to formally verify; this is a sound simplification.

Analysing again the four flowcharts, all of them envision the possibility that a device does not comply with S×C. In that case, external services PC and DBC might be called for assigning an S×C contract to the device. This invocation could produce an overhead, but we already covered this in Section 5. In this section, we are interested to assess the overhead that S×C4IoT core functions add to a network, excluding optional services such as PC and DBC. Looking at the flowcharts in Figure 7 and Figure 8, there are two common phases left out: the contract transmission and the contract/policy matching. Exchanging contracts and verifying them against a policy is critical

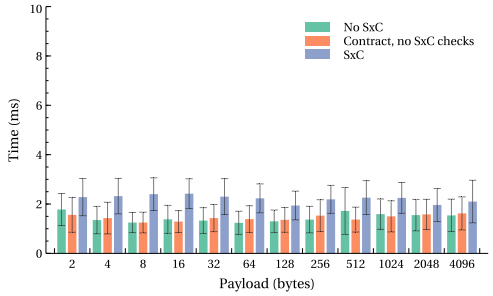

Fig. 11. Time overhead with one rule per contract.

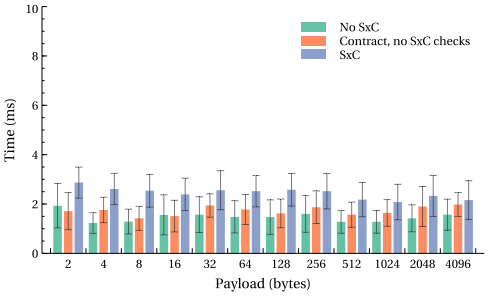

Fig. 12. Time overhead with two rules per contract.

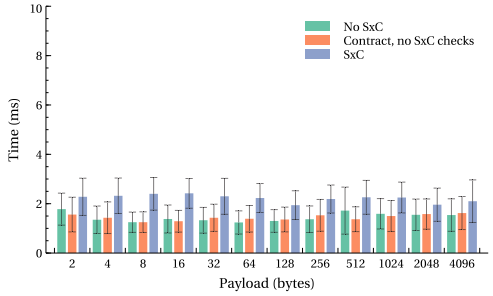

Fig. 13. Time overhead with 4 rules per contract.

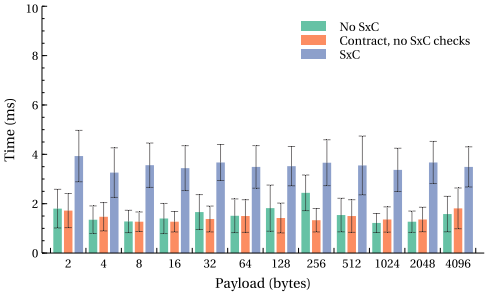

Fig. 14. Time overhead with 6 rules per contract.

and does not come for free; time and packet size overheads are to be expected. We ran some experiments to analyse the extent of such overheads.

It is important to highlight that overheads heavily depend on design choices. For example, we designed our S×C4IoT proof-of-concept in a way that each IoT device within the network has instant access to the policy, as if they locally stored an updated copy. This decreases the total overhead because, instead of querying the Fog node every time that a communication happens, the IoT devices instantly and independently assess the scenario. This is also a fair simplification we introduced in our framework. Even though we envision a policy stored within a Fog node, this does not exclude distributing mechanisms: S×C devices could store a copy of the network policy and, whenever the policy is updated, the Fog node could broadcast it to the devices.

In our experiments, we evaluate two different types of overhead. First, we analyse the overhead due to simply transmitting a contract. Second, we evaluate the overhead that occurs when transmitting and verifying a contract against a policy by means of the contract/policy matching. To do so, we create two virtual IoT devices, a sender and a receiver. The sender creates an MQTT packet with a message of predefined length, attaches its contract, serialises everything, and publishes an MQTT message on the recipient channel. Once the message is received, the recipient deserialises the message and verifies if the sender contract matches with the S×C policy. If the condition holds, then the message is received, stored, and logged. If not, then the message is simply discarded. We want to clarify that this would not be a good option for setting up a proper communication protocol. It would be more efficient to verify the contracts at the beginning of a communication session, instead of adding overhead to every message. However, it is a good option for conducting experiments, because it allows us a proper comparison at the level of single MQTT packets.

7.2.1 Time Overhead. The first thing to evaluate is the impact that contracts have on the time overhead. In particular, contracts with more rules not only entail larger packets for transferring

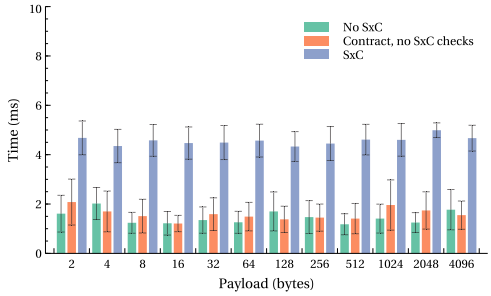

Fig. 15. Time overhead with 8 rules per contract.

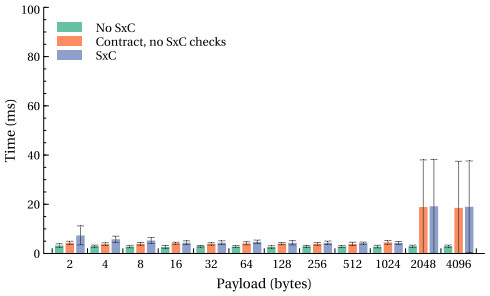

Fig. 16. Time overhead with 16 rules per contract.

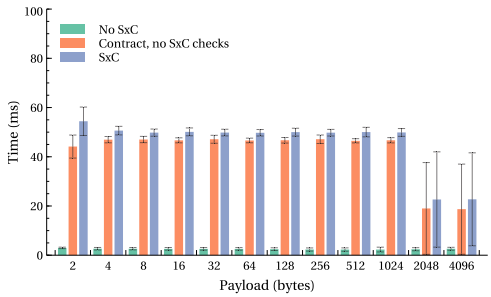

Fig. 17. Time overhead with 32 rules per contract.

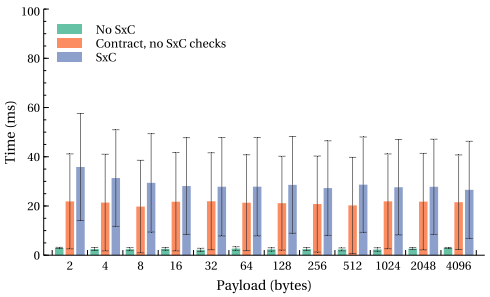

Fig. 18. Time overhead with 64 rules per contract.

the contracts themselves, but also more time to evaluate if any illegal information exchanges exist. This is how we conducted our experiments. First, we create eight different pairs of IoT devices, with a growing number of rules per each (i.e., 1, 2, 4, 6, 8, 16, 32, and 64 rules per contract). Then, we add each pair to 8 different networks, resulting in eight Fog nodes that contain 2, 4, 8, 12, 16, 32, 64, and 128 rules, respectively. Figure 11 to Figure 18 show the results of our experiments for each of these networks. For each network, we create a baseline dataset, sending messages from one device to the other without any S×C routine involved. This means that we do not attach the contracts to the MQTT message, nor do we execute any information flow check upon receiving a message. We call this baseline dataset "No S×C." Regarding the exchanged message, we start with a message of 2 bytes, and we exponentially increase it up to 4,096 bytes. We send each message 100 times, and we calculate the mean to determine the average time it takes to send a message of predefined length.

Once we have obtained the baseline dataset, we add the contracts to the messages and repeat the process; in the graphs, this data is referred as "Contract, no S×C checks." In this second dataset, contracts are not verified against policies with the contract/policy matching algorithm. By doing this, we isolate the overhead that comes only from serialising, communicating, and deserialising a contract. Last, we extend this second set of experiments with a third one, called "S×C" in our graphs, where the contracts are finally verified against the policy. Summing up, each dataset is composed of 9,600 messages, for a total number of 28,800 messages used in our experiment.

In Figure 19, we report the summary of the time overhead, shown in details from Figure 11 to Figure 18. Each bar represents the average time required to send a message, regardless of the payload size. The error bars represent the mean absolute deviation from the mean. As expected and shown in the "No S×C" bar group, the time required for transmitting a message without S×C routines nor contracts, remains fairly stable. The same applies for the case where the contract is

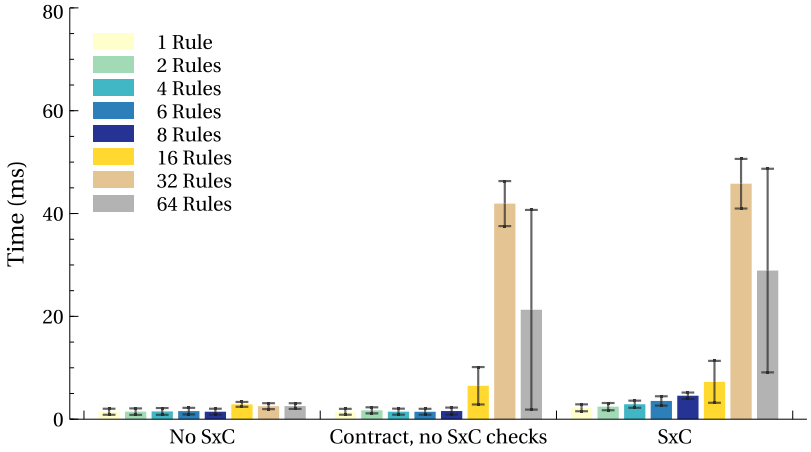

Fig. 19. Average time overhead with growing number of rules per contract.

simply attached, but not evaluated, as shown in the "Contract, no S×C check" bar group, until the contracts are composed of 16 rules or more. With respect to the "No S×C" dataset, a small but consistent overhead should be expected, stemming from the fact that the payload is larger. However, the results show considerable variability, depending on the payload size. In the case of 2 rules per contract and 8 rules per contract, we have a slightly higher delivery time. With 1, 4, and 8 rules per contract, the delivery time is equal or even slightly smaller. This is due to the fact that delivery times are on average about a couple of milliseconds; even small disturbances, such as brief spikes in CPU usage, can have a relevant impact on 1.5 *ms* delivery times.

Focusing on the cases where the rules per contract equal to 16, 32, or 64, the results get considerably worse. The average delivery times, represented by the graph bars, grow up to an average of 40 *ms* for the case with 32 rules, but around 20 *ms* for 64 rules per contract. Interestingly, what stands out in Figure 19 is that the mean absolute deviation with 64 rules is 5-fold bigger than the mean absolute deviation shown with 32 rules. We investigated the matter and we narrowed down the causes to an unintended behaviour of Mosquitto, the MQTT broker we used in our experiments, which introduces a noticeable jitter when the messages size exceed 8 *kB*. As noticeable in Figure 16 and Figure 17, towards the end of the experiment, results show unmotivated variations with respect of the previous bars. Taking Figure 17 as an example, with a growing payload it is reasonable to assume that the result would show a monotonic increasing trend. Instead, delivering a 4,096 bytes payload takes on average half the time than delivering a 1,024 bytes. The mean absolute deviation shows that this is due to the fact that some packets are delivered within 5 *ms*, while others in the same experiment are delivered after more than 40 *ms*, leading to an average of $\approx$20 *ms*. A similar, unexpected behaviour is shown also at the end of the dataset in Figure 16. Last, Figure 18 looks more consistent if the analysis stops at the average delivery time. However, each payload shows the same high variability that we discussed, with some packets that are delivered within 18 *ms* and others that take almost 60 *ms* to reach the destination.

Last, the "S×C" bar group in Figure 19 represents the time overhead that comes from applying S×C as described previously in this section. What we notice is that the overhead grows with the number of rules per contract. This overhead does not happen because of larger size packets, else we would have seen a similar increase in the second dataset. The overhead is mostly due to the fact that more security rules require more comparisons for establishing if a device complies with a network policy or not. Let us assume a contract C_D is compatible with a policy P_F. Intuitively, we

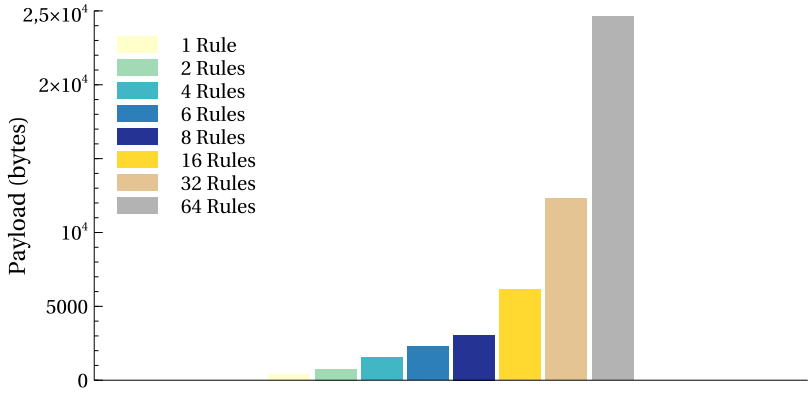

Fig. 20. Packet size overhead with growing number of rules per contract. The rules we introduced are more or less similar to each other, so, as expected, the size overhead grows linearly with the number of rules.

have to check every rule of C_D against every rule in P_F to be sure that C_D would not create any illegal information exchange. In other words, assuming a simple greedy algorithm, if a contract is composed of n rules and a policy of m rules, then the complexity equals to $O(nm)$.

Comparing the average delivery time of the "S×C" case with the "non-S×C" one, shown in Figure 19, the impact of the aforementioned overhead is noticeable but not critical. In the worst case, we evaluated (i.e., 64 rules per contract), our simulations show that the S×C scenario takes approximately 28.9 *ms*, while the non-S×C one requires around 2.5 *ms*. From a relative perspective, this is a 10-fold increase. However, in absolute numbers the overhead is ≈ 25 *ms*, which is unnoticeable delay for a human being, in terms of system responsiveness. It is also important to highlight again that the higher delivery time is likely to be influenced by unexpected behaviour of the MQTT broker. This is supported by the fact that, in the worst-case scenario with 64 rules per contract, the average delivery time of the "S×C" case differs from the average delivery time of the "Contract, no S×C checks" case by just ≈ 7 *ms*. Therefore, S×C operations account only for the 25% of the delay, further supporting that there is an underlying issue with the Mosquitto broker.

7.2.2 Packet Size Overhead. Besides the time overhead, the packet size overhead introduced by S×C contracts is a relevant information. This overhead depends on various factors, such as the data structure used to store the contracts, or the amount of data carried by each contract. Nonetheless, it is still important to estimate the influence that a contract can have on data transmission.

In Figure 20, we show the packet size overhead for different contracts with a growing number of rules. The rules we used are comparable in terms of the number of characters. As expected, the size overhead introduced by a contract grows proportionally with the number of rules per contract. With a single rule, a contract uses only 379 bytes and with 2 rules the payload corresponds to 765 bytes. The required space grows linearly and with 8 rules per contract the required storage space is 3,069 bytes, while with 64 rules per contract it amounts to 24,622 bytes.

7.2.3 Impact on Devices Storage. Modern smartphones are equipped with plenty of computing and storage resources, and it is common to find affordable smartphones with more than 32 GB of storage space. However, IoT devices are fairly limited with respect to available resources. In particular, they do not have access to large storage media as other devices we are used to nowadays. For example, **PIR (Passive InfraRed)** sensors are often piloted by Arduino boards [1, 57], which can have as little as 32 kB of flash memory in their cheapest version [4]. Therefore, any additional component can have a relevant impact on hardware-limited devices.

Table 19. Storage Capacity of IoT Devices Presented

Device	Flash Memory (kB)	Expandable
Arduino Uno	32	Yes
Arduino Micro	32	Yes
Arduino Mega	256	Yes
Philips Hue Motion	512	No
Philips Hue White	512	No
D-Link 933L	8,192	No
OORT Plug	Unknown	Unknown
OORT Lock	Unknown	Unknown

Table 20. Impact of Different Contracts on IoT Devices Storage

Device	1 Rule (B)/ Used Memory (%)	2 Rules (B)/ Used Memory (%)	4 Rules (B)/ Used Memory (%)	6 Rules (B)/ Used Memory (%)	8 Rules (B)/ Used Memory (%)
Arduino Uno	379 (B)/ 1,157%	765 (B)/ 2,335%	1,531 (B)/ 4,672%	2,300 (B)/ 7,019%	3,069 (B)/ 9,366%
Arduino Micro	379 (B)/ 1,157%	765 (B)/ 2,335%	1,531 (B)/ 4,672%	2,300 (B)/ 7,019%	3,069 (B)/ 9,366%
Arduino Mega	379 (B)/ 0,145%	765 (B)/ 0,292%	1,531 (B)/ 0,584%	2,300 (B)/ 0,877%	3,069 (B)/ 1,171%
Philips Hue Motion	379 (B)/ 0,072%	765 (B)/ 0,146%	1,531 (B)/ 0,292%	2,300 (B)/ 0,439%	3,069 (B)/ 0,585%
Philips Hue White	379 (B)/ 0,072%	765 (B)/ 0,146%	1,531 (B)/ 0,292%	2,300 (B)/ 0,439%	3,069 (B)/ 0,585%
D-Link 933L	379 (B)/ 0,005%	765 (B)/ 0,009%	1,531 (B)/ 0,018%	2,300 (B)/ 0,027%	3,069 (B)/ 0,037%

First, we analysed the storage space available in the devices we used throughout this article. As shown in Table 19, Arduino boards exist in different forms, and their flash memory goes from 32 kB up to 256 kB. While this is not a lot, it is also worth mentioning that Arduino boards can be expanded with an SD board/shield that allows them to use an external SD card as storage. However, Philips devices and the D-Link camera have plenty of space in their flash memory: 512 kB and 8 MB, respectively. For OORT devices, we could not find any technical specification.

After finding out the available space, we analysed what would be the relative impact of different contracts on these devices. As previously seen in Figure 20, the contracts size grows linearly with the number of rules they contain. As shown in Table 20, a single rule contract uses 379 bytes, while the smallest Arduino board has 32 kB of storage space. To put it in perspective, a single rule contract requires 1,157% of an Arduino Uno flash space. If the rules grow up to 8, then the respective contract requires 3,069 bytes, which means that this contract would use 9,366% of the available space in Arduino. While this is an impactful footprint, this shows that it is technically possible to store a contract in a cheap Arduino Uno without resorting to an external SD card. Moreover, a PIR sensor is a device that has very specific tasks and is unlikely to participate in complex interactions. Therefore, it is improbable that they would be equipped with complex contracts. At the other end of the spectrum, a D-Link 933L camera is equipped with 8 MB of storage space. If we store a contract composed of 8 rules in this camera, then we use only 0,037% of its available storage space. This shows that a contract has a relatively modest footprint on most of modern IoT devices.

Table 21. Security Contract $C_{Smart.Lamp}$ for a Smart Lamp that Allows Other Devices to Turn It On/off Inside the LAN, and Access to the Internet Only for Checking Its Updates Status

	Rule R_{Lamp_1}	**Rule R_{Lamp_2}**
D	Smart.Lamp	Smart.Lamp
Dom	LAN	Internet
Shares	*	-
Provides	OnOff	-
Invokes	-	Smart.Server.CheckUpdates

7.3 S×C4IoT for Security Configurations and Configurability

As previously discussed, achieving secure default configurations and sufficient security configurability requires the capability of describing acceptable security-relevant behaviours. In this section, we provide examples that show how S×C4IoT addresses these two problems.

7.3.1 Insecure Default Configuration. One of the best examples for insecure default configurations can be found in different contemporary IoT devices that, to grant availability, offer their services to any device within the same network and request Internet access without specifying which domains they require to contact. As a direct consequence of this bad practice, DDoS malware that recruit IoT devices for their botnets grew extremely popular in recent years.

Assume a smart lamp that should offer the service OnOff inside the LAN, and download new updates using the service CheckUpdates offered by an Internet server Smart.Server. This legacy lamp has no way to specify its security-relevant behaviour, hence it is simply granted access to the Internet, as any other LAN device. If a DDoS-enabling malware infects this IoT device and uses it to send a large amount of packets to a third-party victim, then the attack occurs. Within the network, there are no mechanisms for labelling this behaviour as unintended, nor for stopping it.

If this smart lamp were S×C-compliant, then it would exhibit a contract similar to the one shown in Table 21, which states that communication over the Internet takes place only to invoke the service CheckUpdates. If the infected smart lamp starts to flood another service Victim.ImportantService, then the Fog node would be able to detect it and prevent it, for example, by removing the smart lamp from the network. In this case, an S×C4IoT framework can be an efficient bulwark against DDoS attacks performed against third-party targets.

7.3.2 Insufficient Security Configurability. The second issue identified by Zhou et al. [62] that we address in this article, is the insufficient security configurability problem. Manufacturers that produce S×C-compliant devices are enforced to formalize how their devices interact with other devices, in terms of provided and invoked services. This is extremely valuable for improving the security configurability of an IoT network, especially when dealing with S×C-noncompliant devices.

Assume a smart plug that provides a simple service, SetTimer, to any device within the network, as shown in the simple example in Table 22. This is important for two reasons. First, it provides knowledge about the behaviour of the devices. Second, this knowledge is formalized, thus tractable in terms of network policies. An S×C network administrator can setup a network policy that regulates which S×C-noncompliant devices can interact with the smart plug, in fact shaping the behaviour of legacy devices. For example, as shown in Table 23, a network administrator can write a rule in the network policy, allowing generic smart buttons to invoke the SetTimer.

Table 22. Security Rule $R_{SmartPlug}$ that Describes a Smart Plug, which Allows Any Device within the LAN to Set a Sleep Timer

	Rule $R_{SmartPlug}$
D	Smart.Plug
Dom	LAN
Shares	*.*
Provides	SetTimer
Invokes	-

Table 23. Security Rule $R_{GenericButton}$ that Allows a Generic Smart Button to Invoke the Service Generic.SmartPlug.SetTimer

	Rule R_{Plug}
D	Smart.Plug
Dom	LAN
Shares	-
Provides	SetTimer
Invokes	Generic.SmartPlug.SetTimer

Table 24. Security Policy $P_{FogNode}$

	Rule R_{933L}	**Rule $R_{HueWhite}$**
D	D-Link.933L	Philips.HueWhite
Dom	*	LAN
Shares	Apple.LukePhone	Philips.*
Provides	OnOff	On, Bri, Hue
Invokes	-	Philips.HueMotion.Presence

7.4 Prevention and Mitigation of Malicious Behaviours with S×C4IoT

In this section, we provide an overview of the malicious behaviours that can be counteracted with S×C4IoT. Some behaviours can be fully prevented, such as Fog-actionable behaviours; other events, even though Fog non-actionable, can be partially mitigated.

7.4.1 Manipulation Captured by Contracts. The main utility of S×C contracts lies in the fact that they can allow for detection and prevention of Fog-actionable malicious events. Assume a Philips HueWhite device, which is supposed to invoke only service Philips.HueMotion.Presence. D-Link.933L carries a contract, composed of only one rule, that states that it provides service OnOff only to Apple.LukePhone, on any domain. These two rules are shown in Table 24.

Let us also assume that Philips.HueWhite is infected with a malware. This malware attempts to access to the service D-Link.933L.OnOff to shut down the D-Link933L camera. However, rule $R_{HueWhite}$ as stored in the Fog node policy $P_{FogNode}$, states that Philips.HueWhite is expected to invoke only one service, namely, Philips.HueMotion.Presence. This is a Fog-actionable event, as it can be observed by the Fog node monitoring the traffic or can be signalled by D-Link 933L when it receives the unexpected request. In turn, this behaviour can trigger different sanity-checks on Philips HueWhite, such as remote attestation routines or remote malware scans.

7.4.2 Code Manipulation Not Captured by Contracts. One of the upsides of including a PoC that is digitally signed by the manufacturer is that it could prevent some malicious activities that are not covered directly by the S×C contract and the PoC. Let us assume that we are dealing with a smartlock that unlocks when the service OpenLock is invoked by any device that has access to the LAN and can reach the lock service. However, the device is programmed to open only after 8pm, when its owner comes back home from work. While it is possible to envision a contract that specifies time constraints, we keep as a reference contracts that do not include this specification, as described in Section 4 and detailed in Appendix A.3. The resulting contract would be simple, similar to what is shown in Table 25.

Let us assume now that a malicious attacker wants to break into the victim's house while he is at his workplace. The attacker attempts to tamper with the smartlock software, aiming to change the

Table 25. A Simple Contract for a Smartlock that Opens when OpenLock Service Is Invoked

	Rule $R_{Lock\_Extracted}$
D	SmartLock
Dom	LAN
Shares	**
Provides	OpenLock
Invokes	-

time constraint from 8pm to 4pm, so she can trigger the OpenLock service before 8pm and enter the house. However, to be successful, the attacker would have to be able to produce a valid PoC that reflects the new software, as well as a valid manufacturer signature that binds all the components together. As previously described in Section 3, we rely on the fair assumption that a malicious attacker cannot get hold of the private key used by the manufacturer to sign the triplet *<Software, Contract, PoC>*. The software update triggers the Fog-actionable event "Update Software" described in Section 6, Figure 8, leading the Fog node to verify the new triplet validity and discovering that the device has been tampered with.

In conclusion, the act of signing a triplet *<Software, Contract, PoC>*as a single entity has the potentiality (but not the guarantee) to compensate for missing details in contracts, with positive implications in terms of security. In this specific case, even though the contract was not detailed enough for specifying the relevant time constraint, the S×C4IoT framework expectation of a valid triplet allowed to limit the damages of this malicious manipulation. However, more expressive contracts would be desirable. Not only would they prevent similar threats in a more efficient and controlled way, but they would also allow for *a priori* fine-grained reasoning over devices' behaviour and interactions.

7.4.3 Malware Operating within Contracts Boundaries. As previously stated in Section 3.2, in this article, we split every possible event within an S×C network in two groups: Fog-actionable and non-actionable events. In this article, we cover the first category and we do not consider the latter, meaning that there are cases that we cannot address with our solution.

Assume two S×C devices. One is a camera IP.Camera that allows any device inside the LAN to take a snapshot of the current frame, invoking the service TakeSnapshot. The other device is a button, Smart.Button, which can use the service IP.Camera.TakeSnapshot and take snapshots, but is not allowed to share this information with other devices outside the LAN. The same button provides a service UpdateStatus, which allows an external storage Remote.Storage to download raw data from the button. However, this raw data is supposed to contain only information about the operational status of the button, nothing stored in the Smart.Button internal storage. Let us also assume that the onboard software ensures total isolation between the two functions, so the second function can be provided without violating the camera requirement of communicating only inside the LAN. In other terms, the button cannot upload any internal storage information (i.e., the snapshots). Two hypothetical contracts for these two devices are shown in Table 26.

At this point, a malware infects the button. This malware performs an attack similar to **Return Oriented Programming (ROP)**, reusing the existing contracts to perform malicious actions. In particular, the malware circumvents the aforementioned isolation, extracting a snapshot from the internal storage, converting it to raw data, and attaching this information to the operational status raw data. Eventually, when Remote.Storage invokes the Smart.Button.UpdateStatus service,

Table 26. Two Contracts for Two Devices, IP.Camera and Smart.Button

	Rule R_{Camera}	**Rule R_{Button_1}**	**Rule R_{Button_2}**
D	IP.Camera	Smart.Button	Smart.Button
Dom	LAN	LAN	*
Shares	**	-	Remote.Storage
Provides	TakeSnapshot	-	UpdateStatus
Invokes	-	IP.Camera.TakeSnapshot	-

Due to the current degree of expressiveness, a malware could potentially perform malicious actions that fall within the scope of legitimate contracts.

the snapshot leaks out of the LAN. In fact, even though the button acts within the boundaries defined by its own contract, it violates the intended behaviour. This is a case of a malicious behaviour that is not detectable by an S×C4IoT framework in its current state, falling within the set of Fog non-actionable events, as described in Section 3.2. A solution for mitigating these issues lies in the expressiveness of S×C contracts. For example, improving the expressiveness of the contracts and including a field that restricts the data size and content would prevent unintended use of services.

Another approach would be to integrate our framework with other strategies that specifically aim to detect and solve Fog non-actionable events. Dushku et al. [18] proposed SARA, a remote attestation protocol that verifies the trustworthiness of IoT devices and their exchanged data. Koeberl et al. [34] proposed a security architecture for tiny embedded devices called TrustLite. TrustLite envisions hardware-enforced isolation of software modules that provides trusted computing execution. Furthermore, Agarwal et al. [2] proposed an approach that relies on context-sensitive features for improving the detection of abnormal programs executions. These techniques and S×C4IoT are not mutually exclusive and they can be federated for achieving a secure IoT framework.

7.5 Network Architecture Considerations

In our work, we assume an IoT network architecture that is Fog-based and relatively centralized, but different IoT network architectures exist. In this section, we analyse the implications of our assumptions, highlighting benefits and drawbacks with respect to other network types.

7.5.1 Impact of Fog Computing versus Cloud Computing. The Fog node is an important component of S×C4IoT. It has multiple roles, from evaluating whether an IoT device contract is compliant with the network policy to safely storing the policy itself. As previously discussed in our threat model in Section 3.2, in this work, we assume that the Fog node is trustworthy. However, it is necessary to discuss what are the implications about choosing Fog computing over Cloud computing as a reference paradigm. Conducting experiments to unravel the different impact of the two paradigms is quite complex and would require a number of in-depth analyses that goes beyond the scope of our work. For example, Cloud architectures and services can vary substantially from each other, in terms of implementation and design choices. The same applies to Fog computing: Fog nodes can be implemented on different architectures, multiple Fog nodes can be deployed, and computing loads balanced on specific network requirements.

That being said, the literature has shown that Fog computing can bring considerable benefits over Cloud computing, from the point of view of scalability and execution speed. Veeramanikandan and Sankaranarayanan [38] proposed a software-defined multi-tier Fog computational model for IoT with a remote MQTT broker at the Fog layer and a main MQTT broker at the Cloud layer. Their experiments, conducted with 50 IoT devices and up to 10,000 MQTT clients, showed that the addition of a Fog node improved the performance of their MQTT setup and did not introduce

any bottleneck. In particular, their multi-tiered MQTT solution reduced the service latency up to 32,4%.

Ferrández et al. [20] deployed a Fog-based solution on a pre-existing residential building, aiming to achieve a fully functional smart-building. Power management, as well as control and monitoring services were designed and implemented on a single Fog node, a simple RaspberryPi with a single 1,400 MHz CPU and 1 GB memory. On the same Fog node, an MQTT broker and a simple Machine Learning algorithm (i.e., KNN) also ran, showing that even a fairly limited Fog node is capable to deal with relatively complex tasks without introducing relevant overhead nor scalability issues.

Regarding the implementation of a Fog node, it is worth noting that the architecture chosen can play a relevant role as well. Maleki et al. [41] showed that Spark-based Fog nodes can considerably improve Fog nodes scalability and reduce their power consumption, with respect to traditional Fog nodes implementations. It is also interesting to highlight that Spark brings noticeable benefits when dealing with tasks that can be parallelized, as noted by the authors. The policy-matching algorithm proposed in this article is a good example of a task that is extremely easy to parallelize: Every contract rule can be independently verified against every policy rule, as each comparison does not rely on the output of any other comparison.

Thanks to the Fog computing capability of scaling depending on the offload, S×C4IoT has no theoretical upper-bound of IoT devices that it can manage, provided enough computational power and sufficient optimization at the Fog node level (e.g., parallelization of contract-policy matching algorithms). Besides, while throughout this article we used smarthome examples, any scenario that utilizes a Fog node and requires to manage IoT devices based on their security relevant behaviours can adopt S×C4IoT. Bhattcharya and De [6] showed that offloading computational tasks to larger edge devices can provide better performance than offloading the same tasks to a Cloud server, around a 10% decrease time. Since the authors conducted their experiments on a regular laptop, it is safe to assume that a dedicated Fog node would achieve even better results.

7.5.2 Centralization and Applicability to Mesh Networks. In this article, one relevant assumption is that we consider a network that incorporates Fog computing in the form of—at least— one Fog node. Fog computing is gaining more and more attention among IoT and **industrial IoT (IIoT)** applications, for its capability of overcoming the downsides of pure Cloud-based solutions, such as high latency and intermittent connectivity that prevent timeliness [51].

At first glance, the requirement for a centralized node can create friction with IoT networks deployed with mesh protocols, such as Zigbee. However, that is not strictly the case. As we previously mentioned in Section 3.2, considering a Fog node that monitors and detects Fog-actionable actions is a helpful simplification, but it is not a strict requirement. Similar results can be achieved with a polling strategy, where the Fog node randomly challenges the IoT devices for verifying their integrity. Alternatively, IoT devices themselves can play an active role in ensuring network integrity. The simplest example is a D_B that receives a request from D_A to invoke D_B.Service, even though D_B does not allow it in its contract. Something suspicious might be happening, and D_B triggers a warning to the Fog node, which polls the suspicious device.

Research shows that it is possible to efficiently combine mesh networks with TCP/IP WiFi networks and that Fog computing can be the key enabler. In 2018, Froiz-Míguez et al. proposed ZiWi [23], a distributed home automation system based on Fog computing, which allows seamless communication between Zigbee and WiFi over MQTT. In their work, the authors highlight that ZiWi decouples hardware and software from the Cloud, remarkably reducing latency for real-time operations. They also point out that Fog nodes alleviate the Cloud from undertaking every task and that, by keeping sensitive data within the network, they increase security and data privacy. The

same rationale applies to our network policy that, without a Fog node, would have to be managed at the Cloud level.

Regarding the single point of failure problem, it is worth noting that ZiWi was designed and conceived for being deployed on a Fog computing architecture; the authors stress that this makes easy to scale up by merely adding Fog nodes, which provide service redundancy by-design. A similar remark was done by Butun et al. [8], who explain that the single point of failure problem can be avoided by means of replication, easily achievable with Fog nodes that are intentionally designed for scaling-up. The authors also mention that Fog computing can improve IoT networks security, simply by the fact that a layered approach such as Edge/Fog/Cloud allows to split security tasks, avoiding centralization. At the same time, Butun et al. also mention that centralizing too many access control features on a single Fog node can potentially increase security risks for the IoT network, which is a risk not to underestimate.

8 LIMITATIONS

One limitation of this study is the PoC implementation. As mentioned in Section 6.6, in our proof-of-concept, we did not implement a functional PoC and we opted for stub methods that represent its functionalities. This is a fair simplification, given that enriching execution codes with formal proofs has been proved possible in the past (as discussed in Section 3). However, this also leaves some open challenges. In Section 7.2.3, we discussed the impact that an S×C contract has on different IoT devices, but we neglected how much storage space would be necessary for storing the PoC. As shown in Table 20, on embedded devices like Arduino Uno, a contract composed of eight rules uses 10% of the device storage or more, if the rules are more complex. If we add the storage space required by the PoC, then S×C might not be feasible for tiny embedded devices. Similarly, transmitting the entire triplet <*Software, Contract, PoC*>might have considerable communication cost.

Another aspect that we neglected in this study is the energy consumption that comes from introducing S×C in limited embedded devices and the potential attack surfaces for energy-consuming attacks. For example, a malicious agent could use a device to forward a warning message to the Fog node, tricking the latter into challenging repetitively a third victim device. This would force the victim to reply to the Fog node challenges, leading to consuming its own power reserve to a faster rate than expected. These and other open challenges will require further investigation, which we will undertake in our future work.

9 RELATED WORK

IoT security is a critical topic that has been thoroughly discussed in recent years. In this section, we do not provide a detailed discussion of IoT security state-of-the-art, which would require an article of its own. Our goal is to give an overview on the most relevant works with respect to IoT security configurations and configurability. In Table 27, we provide an overview of the capabilities of S×C4IoT with respect to the other proposals discussed in this section.

Matheu et al. [43] highlighted that manufacturers should be included in the loop for creating more resilient devices. The authors proposed 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 the contracts useful information about compliance with 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. [35] proposed to strengthen IoT security through a **network edge device (NED)**, a secure device that stores the user-defined policies and enforces them on

Table 27. S×C4IoT Comparison with Related Work

	S×C4IoT	MUD [36]	W3C TD [33]	Kuusijärvi et al. [35]	Matheu et al. [42]	BACnet [29]	Hamza et al. [30]
Feasibility	Relevant effort for producer ~	Existing implementations ✓	Relevant effort for producer ~	Relevant effort for producer ~	Based on original MUD feasibility ✓	Already deployed for HVACs ✓	Based on original MUD feasibility ✓
Admin experience	Intuitive semantics ✓	Intuitive semantics ✓	Complex semantics ✗	Relevant effort from end-users ~	Intuitive semantics ✓	Somewhat intuitive semantics ~	Intuitive semantics ✓
End-user experience	Transparent ✓	Transparent ✓	Transparent ✓	Transparent ✓	Transparent ✓	Transparent ✓	Transparent ✓
Compatibility with legacy devices	Partial, without PoC ~	No ✗	No ✗	Yes ✓	No ✗	No ✗	Yes ✓
Required additional server or gateway	Fog node, not entirely dedicated ~	MUD Controller ✗	Unspecified -	NED, not entirely dedicated ~	MUD Controller ✗	BACnet server ✗	MUD Controller ✗
Human-readability	JSON ✓	JSON ✓	JSON ✓	No ✗	JSON ✓	Clear-text specification ✓	JSON ✓
Behavioural description	Contract ✓	MUD file ✓	TD file ✓	No ✗	MUD file ✓	BACnet file ✓	MUD file✓
Binding of description and behaviour	PoC ✓	No ✗	No ✗	No ✗	No ✗	No ✗	No ✗
Detection of privacy leaks	Holistic focus ✓	End-to-end focus ~	End-to-end focus ~	End-to-end focus ~	End-to-end focus ~	End-to-end focus ~	End-to-end focus ~
Detailed security properties	No, but contracts are expandable ~	No ✗	Security-Scheme metadata ✓	No ✗	Security-Scheme metadata ✓	No ✗	No ✗
Reasoning on sets of devices	Rules based on manufacturers	No ✗	No ✗	No ✗	No ✗	No ✗	No ✗

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 to 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 [3] and providing secure resource orchestration in Fog computing [16]. Similar to our work, they share the necessity of enforcing policies at the Fog layer over devices that might not be compliant by design.

MUD [36] is an IETF specification (RFC8520), in which manufacturers specify which hosts and ports their devices need to operate correctly. MUD 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) that defines appropriate ACLs. However, MUD-compliant devices do not carry the contract themselves. A MUD device stores 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. Besides, it is unclear how the contracts should be parsed, verified, and treated with respect to a security policy. The key S×C concepts of contract/policy matching and policy enforcement seem to be missing. Moreover, a MUD file is pretty restrictive, as it barely describes basic information such as allowed protocols and reachable hosts. Other researchers pointed out that, although MUD is a relevant contribution to the IoT security state-of-the-art, it has some shortcomings from the point of view of expressiveness.

Matheu et al. [42] proposed an extension to the MUD model, going beyond communication restrictions at the network level and taking into account other factors, such as cryptographic algorithms and keys size. Other researchers built on MUD. Hamza et al. [30] 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 the intended ports, then MUD alone cannot be enough to identify the ongoing attack. Another work by Hamza et al. [31] 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.

BACnet is a communication protocol for automated building control systems, such as **heating, ventilating, and air-conditioning control (HVAC)** [29]. BACnet devices carry *objects* that describe the services they offer in terms of control and relevant data for other devices. Recently, BACnet was extended with **BACnet Secure Connect (BACnet/SC)** [21], a virtual datalink that provides BACnet devices secure TLS communication and authentication capabilities.

The **World Wide Web Consortium (W3C)** noted that IoT devices lack from the point of view of behavioural description, and proposed **Web of Things (WoT) Thing Description (TD)** [33]. A device equipped with TD provides textual metadata about itself, a set of Interaction Affordances that describe how other devices can interact with it, data schemas for describing data formats and improving machine understandability, and Web links for expressing any relation to other devices or documents on the Web. TD approach is URI-based, as it is MUD approach, but it is more expressive and easier to extend. Even though TD does not express interactions in terms of sets and subsets of devices, it allows to add contextual definitions that expand its semantics. Therefore, it seems possible to create fields that correspond to S×C fields, such as “manufacturer” and “device”). In a similar way, TD does not explicitly express which services are necessary for a device to operate correctly. However, TD provides generic *links* fields that allow defining relationships with other devices; for example, a smart lamp could allow a motion sensor to turn it on/off, by means of a *controlledBy* statement. It is worth noting that TD provides the *form* class, a hypermedia control for manipulating devices’ state. Within a form, there is an *op* field that expresses the operation name, the *href* that points to the correct URI for accessing the service, and other fields that define the parameters. Instead of looking at S×C and TD as competing solutions, we can think of TD as an enabling middle layer. By means of an *op* field, TD could bind S×C high-level concepts (e.g., a service) to URI-based resources (e.g, a web service accessible with a HTTP POST request).

Regarding our proposal, in previous works, we discussed how S×C can be combined with the Fog paradigm, and we described the main pillars of S×C, as well as the main framework life-cycles [26]. Then, we formally defined the framework components, the policy/matching algorithm, and other critical concepts, such as the allowed information flow and illegal information exchange [25]. In this article, we summarise those contributions for the sake of clarity, and we extend the contribution by analysing how our S×C framework can cope with legacy devices and devices’ dynamic evolution.

As discussed throughout this article and summarised in Table 27, S×C4IoT envisions a framework where devices exhibit a description of their behaviour and carry a formal proof that confirms the description. Moreover, while other proposals are IP-based and focused on end-to-end security rules, S×C4IoT allows to evaluate the network in a holistic way and identify potential data leakage. In a heterogeneous network composed of dozens of different devices, this is an invaluable characteristic. In its current form, S×C4IoT does not allow to specify specific security properties, such as available cipher suites or minimum key size. However, contracts can be expanded with new

fields that could easily contain such information. Another interesting characteristic of S×C4IoT is that it allows to reason in terms of sets of devices. Not only rules can apply to every device or a specific device, they can also apply to sets of devices that have been manufactured by the same company. This provides to the manufacturers the flexibility of defining more permissive rules, on the premise that manufacturers trust their own products more than third-party devices.

An upside shared by many proposals, analysed in Table 27, is that the syntax for defining the security relevant actions is intuitive, which makes the semantics of the behavioural descriptions intuitive as well. Intuitive syntax and semantics simplify the tasks of network administrators that need to define their own network policies, reducing the probability that errors are introduced in such policies. Similarly, all the solutions we took into consideration envision a transparent system for end-users, which simply necessitate to connect their IoT devices to the network.

Among the drawbacks of S×C4IoT is that it requires a relevant effort from the manufacturers to compute the triplet <*Software, Contract, PoC*>, which also has an impact over the IoT devices' storage. This requirement is higher than MUD, for example, which only stores a simple URI in the devices. Most solutions do not envision a mechanism to guarantee retrocompatibility with legacy devices. The exception is Hamza et al. [31], which, assuming the profiling process is successful, allows to achieve almost perfect compatibility with regular MUD devices. S×C4IoT provides routines for retrocompatibility, but it cannot create a valid PoC for such devices. While the result is that legacy devices can fully cooperate in an S×C environment, they are still not fully comparable to S×C-compliant devices that provide a full-fledged triplet. Moreover, all the solutions analysed in Table 27 require an additional node/server/gateway that perform verification operations. Kuusijärvi et al. [35] and our solution have a small benefit over the others: The gateway is not a dedicated node, unlike the MUD controller used by MUD-based solutions, for example. This reduces costs and network complexity to a certain extent.

10 CONCLUSION

In this article, we aim to address two main challenges for nowadays IoT devices: default insecure configurations and insufficient security configurability. Tackling these issues is a challenging task. Current IoT devices rely upon the Cloud for offloading computing tasks and, in general, for managing their configurations. Taking into consideration the growth rate of IoT devices on the market, it is necessary to rethink this paradigm for enabling IoT devices to adapt to the environment and self-configure themselves. This can happen only if every device within the same domain shares a common way of describing its behaviour. Learning firsthand about the environment is computationally expensive, so it is not a feasible strategy for resource-restricted IoT devices.

In this work, we proposed S×C4IoT, a framework that combines the Security-by-Contract (S×C) paradigm with the Fog paradigm. We gave an intuitive description of the foundations of S×C4IoT, followed by formal and thorough definitions. We showed how to describe behaviours by means of S×C contracts and how to express acceptable network behaviours through S×C policies.

We discussed two different approaches for dealing with legacy devices that are not S×C-compliant by-design. The first approach envisions a parser that extracts the relevant information from the IoT device and complements it with additional information extracted from the Fingerbank API. The second approach leverages machine learning algorithms to analyse the device, discover which kind of device it is, and assign it an appropriate predefined contract for the category.

Furthermore, we claim that it is critical to take into consideration that IoT devices evolve according to their specific life-cycles. Such evolutions can trigger many dynamic changes within a network, evolutions that must be taken into account in a complex security framework. In this article, we identified five different types of dynamic evolutions, and we defined five precise workflows to manage such evolutions in a consistent and predictable way.

We evaluated the impact of S×C4IoT in different case studies, defining which threats can be prevented and mitigated with S×C4IoT and which ones cannot. We provided notable examples that show how S×C4IoT helps to address the insecure default configuration and the insufficient security configurability problems. We also covered benefits and drawbacks of introducing the Fog computing paradigm in an IoT environment, such as reduced latency and centralization issues.

We implemented S×C4IoT as a proof-of-concept, and we evaluated the impact that it might have when deployed in a real scenario. Our main goal was to have a clear depiction of the impact that the main phases can have on the network performance. Therefore, we isolated and separately evaluated the most impactful components. First, we confirmed the correctness of our implementation by means of in-depth unit tests. Second, we proved that it is possible to transmit a contract as a JSON file included in an MQTT message, showing that the number of rules per contract has a linear impact on the message payload, as predictable. Third, we showed that increasing the number of rules per contract has an impact on the contract-policy matching algorithm, whose complexity grows with the number of input rules. However, the overhead proved to be limited to $\approx 3ms$ with a contract and a policy composed of 8 and 16 rules, respectively. The performance degrades more significantly when a message sent over MQTT exceeds $8kB$, leading to an average overhead of $\approx 40ms$. While this degradation is considerable, in terms of relative values, it is worth noting that $40ms$ is a hardly noticeable delay for a human being. Besides, we have discussed how this delay is not due to S×C4IoT but to Mosquitto, the MQTT broker that we used in our experiments.

Moreover, we evaluated the impact of S×C4IoT on the storage space of IoT devices. We collected information about the available storage space for the devices we used as case studies in this article, and we analysed how much space different contracts would require in terms of absolute and relative numbers. We conclude that such overhead is acceptable for many IoT devices and that our experiments proved that it is possible to achieve an effective S×C4IoT framework for IoT devices.

APPENDICES

A BACKGROUND

In this section, we summarise the key pillars that allow us to manage S×C-compliant devices. This section and the formal definitions have been adopted from our previous work [25]. While these are not new contributions, a summary of formal definitions is necessary for making this article self-contained. For a full treatment of this topic, we refer the reader to the original paper [25]. As shown in Figure 1, this part covers the management of S×C-compliant devices, depicted in Stages 2A, 3, 4, and 5, grouped in the green dashed box A.

A.1 S×C Security Rules

Both security contracts and policies are a collection of different security rules. Therefore, before we can formally describe contracts and policies, we need to define what a security rule is. Security rules are composed of devices, domains, and services. These three entities are defined as follows:

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.

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.

By composing these three blocks, we obtain a formal definition of an S×C security rule:

Table 28. 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).
Invokes	List of services s_1, ..., s_m, $m \geq 0$ that the IoT device D might invoke to function.

Definition 4 (Security rule). A security rule (or simply, a rule) R is a 5-tuple represented by the fields listed in Table 28.

From now on, we use the notation R[D] to denote the device D of a security rule R. Analogously, R[Dom], R[Shares], R[Provides], and R[Invokes] denote the related fields of a rule R. With respect to our previous work, the field we previously called Requires has been replaced by Invokes to better depict the fact that the services listed in Invokesmight, or might not, be critical.

A.2 Well-formed and Core Security Rules

To avoid malformed security rules, we introduce the notion of well-formed security rule. Intuitively, a security rule is well formed if it states which IoT devices can use the services specified in R[Provides]. Since these devices are listed in R[Shares], we have that R[Shares] must not be empty in case D provides any service in R (i.e., R[Provides] is not empty).

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.

Ensuring that a security rule is well formed is necessary for achieving consistency, but it is not sufficient. Intuitively, two rules can be well formed but also contradict each other. No rule concerning a device should restrict another rule concerning the same device. With restricts, we mean that the rule provides less or the same number of services to the same set, or a subset, of devices.

Definition 6 (Core rule). Given a set of security rules Set = $\{R_1, \ldots, R_m\}$, m>0, a rule R ∈ Set is said to be core with respect to Set, if and only if the following condition holds:

$$R \in \text{Set is } \textit{core} \text{ iff } \nexists\, R_\star \in \text{Set}: (R[D] = R_\star[D]) \wedge (R[\text{Dom}] = R_\star[\text{Dom}]) \wedge R_\star[\text{Shares}] \neq \emptyset \wedge (R_\star[\text{Shares}] \subseteq R[\text{Shares}]) \wedge (R_\star[\text{Provides}] \subseteq R[\text{Provides}]).$$

A.3 S×C Security Contract

After we have defined a security rule, we can give a formal definition for an S×C security contract. In Figure 1, Stage 2A, we show that the first step for a manufacturer for achieving an S×C-compliant device is to create an S×C contract. 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 C_D of an IoT device D is a non-empty and non-ordered set of security rules concerning the device D, such that:

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

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

After the contract, the manufacturer must issue and sign a new PoC. In turn, as depicted by Stages 3 and 4 in Figure 1, this creates a valid S×C device. If, due to a device update, the software or the contract change, then the manufacturer must issue and sign a new PoC, renewing the triplet *<Software, Contract, PoC>*.

To be acceptable within an S×C environment, a contract must be consistent. A contract is said to be consistent if two conditions hold. First, all the rules in a consistent contract have to be well formed. Second, all the rules in the contract are core, which ensures 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. Formally:

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

A.4 S×C Security Policy

Similar to what we have done for security contracts, now we define 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 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 $C_{D_1}, \ldots, C_{D_n}$ of the IoT devices the Fog node F is responsible for, and a number of additional contracts $C^A_{D_1}, \ldots, C^A_{D_p}$ defined by the administrator A, which apply to the respective devices $D_1, \ldots, D_p$. 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:

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

A.5 Communication Flows

In this section, we describe how we manage the communication flows within the S×C framework. In particular, we provide formal definitions for *direct communication*, *information flow*, and *illegal information exchange*. These definitions are fundamental for the contract/policy matching defined in the next subsection.

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

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

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 $\forall\ R_\star \in C_{D_1}, R_\circ \in C_{D_2}$: $R_\star[\textsc{Invokes}] \cap R_\circ[\textsc{Provides}] \neq \emptyset$, we have $R_\star[\textsc{Shares}] \subseteq R_\circ[\textsc{Shares}]$.

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 $\exists\ R_\star \in C_{D_1}, R_\circ \in C_{D_2}$: $R_\star[\textsc{Invokes}] \cap R_\circ[\textsc{Provides}] \neq \emptyset \wedge R_\star[\textsc{Shares}] \not\subseteq R_\circ[\textsc{Shares}]$.

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 $(D_1 \overrightarrow{no} D_2) \vee (D_2 \overrightarrow{no} D_1)$.

A.6 S×C Contract/Policy Matching

The last concept we have to define concerns the matching between a device contract and a Fog node policy. Before giving a formal definition for contract/policy matching, we need to introduce the notion of consistent security policy.

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

(1) $\forall\ R \in P_F$, R is well formed;
(2) $\forall\ R \in P_F$, R is core in P_F;
(3) $\nexists\ C_D \in P_F, C_D \nRightarrow P_F$;
(4) $\nexists\ C_D^A \in P_F, C_D^A \nRightarrow P_F$.

Finally, we can formally define the core idea behind the S×C approach, the matching. A device is allowed to join a network if, and only if, the contract of the device matches the policy stored in the Fog node. The matching phase is depicted as Stage 5 in Figure 1 and is hereby defined.

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.

B PARSED CONTRACTS - TEST RESULTS

MUD File	S×C Device	LAN		Internet		# Dom.
amazonEcho	Amazon.Echo	5,353	O	33,434	I/O	20
		1,900	O	443	I/O	
		67	I/O	123	I/O	
		53	I/O	89	I/O	
augustdoorbellcam	August.DoorBellCamera	67	I/O	443	I/O	19
		53	I/O			
		547	I/O			
awairAirQuality	Awair.R2	67	I/O	8,883	I/O	3
		53	I/O	443	I/O	

MUD File	S×C Device	LAN		Internet		# Dom.
belkincamera	Belkin.NetCam	5,104	I/O	8,899	I/O	8
		3,478	I/O	8,443	I/O	
		1,900	O	5,104	I/O	
		67	I/O	3,475	I/O	
		53	I/O	443	I/O	
				123	I/O	
blipcareBPmeter	BLIP.Systems	67	I/O	8,777	I/O	1
		53	I/O			
canaryCamera	Canary.All-in-One	67	I/O	443	I/O	8
		53	I/O	80	I/O	
chromecastUltra	Google.ChromecastUltra	5,353	O	5,228	I/O	37
		1,900	O	443	I/O	
		67	I/O	123	I/O	
		53	I/O	80	I/O	
dropcam	Nest.Camera	67	I/O	443	I/O	4
		53	I/O	123	I/O	
hellobarbie	Nabi.BarbieTablet	67	I/O	443	I/O	3
		53	I/O			
hpprinter	HP.Printer	5,355	O	5,223	I/O	3
		5,353	O	5,222	I/O	
		547	I/O	443	I/O	
		137	O	80	I/O	
		67	I/O			
		53	I/O			
HueBulb	Philips.PhilipsHueSmartlighting	5,353	O	443	I/O	12
		1,900	O	123	I/O	
		67	I/O	80	I/O	
		53	I/O			
ihomepowerplug	iHome.SmartPlug	5,353	O	443	I/O	2
		67	I/O	80	I/O	
		53	I/O			
lifxbulb	LIFX.lighting	56,700	O	56,700	I/O	2
		67	I/O	123	I/O	
		53	I/O			
nestsmokesensor	Nest.Smoke+COAlarm	67	I/O	11,095	I/O	46
		53	I/O			
NetatmoCamera	Netatmo.Camera	67	I/O	4,500	I/O	12
		53	I/O	500	I/O	
				443	I/O	
				123	I/O	
				80	I/O	
NetatmoWeatherStation	Netatmo.PersonalWeatherStations	67	I/O	25,050	I/O	1
		53	I/O			
pixstarphotoframe	Pix-Star.WiFiFrame	138	O	443	I/O	2
		137	O	80	I/O	
		67	I/O			
		53	I/O			

MUD File	S×C Device	LAN		Internet		# Dom.
ringdoorbell	Ring.Doorbell	67	I/O	9,998	I/O	4
		53	I/O	443	I/O	
				123	I/O	
				80	I/O	
samsungsmartcam	Samsung.IPCamera	5,353	O	5,222	I/O	5
		1,900	O	443	I/O	
		67	I/O	123	I/O	
		53	I/O			
SmartThings	Samsung.SmartThings	1,900	O	443	I/O	3
		67	I/O	123	I/O	
		53	I/O			
tplinkcamera	TP-Link.IPCamera	5,353	O	3,478	I/O	6
		67	I/O	443	I/O	
		53	I/O	123	I/O	
				80	I/O	
tplinkplug	TP-Link.HS100	67	I/O	50,443	I/O	11
		53	I/O	123	I/O	
tribyspeaker	Invoxia.SmartPortableSpeaker	5,353	O	10,003	O	14
		67	I/O	10,002	I/O	
		53	I/O	8,090	I/O	
				5,228	I/O	
				443	I/O	
				123	I/O	
				80	I/O	
wemomotion	Belkin.WeMo	1,900	O	8,899	I/O	3
		123	I/O	8,443	I/O	
		67	I/O	3,478	I/O	
		53	I/O			
wemoswitch	Belkin.SmartHome	1,900	O	8,443	I/O	2
		3,478	I/O	3,475	I/O	
		123	I/O			
		67	I/O			
		53	I/O			
withingsbabymonitor	Withings.SBM	5,353	O	1,935	I/O	7
		67	I/O	80	I/O	
		53	I/O			
withingscardio	Nokia.-WithingsIoT	67	I/O	443	I/O	1
		53	I/O			
withingssleepsensor	Withings.AURA	5,353	O	443	I/O	1
		67	I/O	80	I/O	
		53	I/O			

C FINAL RESULTS OBTAINED WITH OUR IDENTIFICATION TECHNIQUE

Protocol	Device Name	Type	Predicted Type	Result
MUD	Amazon Echo	Speaker	Speaker	Pass ✓
MUD	August Doorbell	Doorbell	Doorbell	Pass ✓
MUD	Awair Air Quality	House Environment Monitor	House Environment Monitor	Pass ✓
MUD	Belkin Camera	Camera	Camera	Pass ✓
MUD	Blipcare BP Meter	Health Monitor	No Classification	Fail ✗
MUD	Canary Camera	Camera	Camera	Pass ✓
MUD	Chromecast Ultra	Speaker	Speaker	Pass ✓
MUD	Dropcam	Camera	House Environment Monitor	Fail ✗
MUD	Philips Hue Bulb	Light	Light	Pass ✓
MUD	iHome Power Plug	Electric Control	Electric Control	Pass ✓
MUD	Lifx Bulb	Light	Light	Pass ✓
MUD	Nest Smoke Sensor	House Environment Monitor	House Environment Monitor	Pass ✓
MUD	Netatmo Camera	Camera	Camera	Pass ✓
MUD	Netatmo Weather	House Environment Monitor	House Environment Monitor	Pass ✓
MUD	Ring Doorbell	Doorbell	Doorbell	Pass ✓
MUD	Samsung Smart Cam	Camera	Camera	Pass ✓
MUD	TP Link Camera	Camera	Camera	Pass ✓
MUD	TP Link Plug	Electric Control	Electric Control	Pass ✓
MUD	Triby Speaker	Speaker	Speaker	Pass ✓
MUD	Wemo Motion	Motion Sensor	Motion Sensor	Pass ✓
MUD	Wemo Switch	Electric Control	Electric Control	Pass ✓
MUD	Withings Baby Monitor	House Environment Monitor	Health Monitor	Fail ✗
MUD	Withings Cardio	Health Monitor	Health Monitor	Pass ✓
MUD	Withings Sleep Sensor	Health Monitor	Health Monitor	Pass ✓
BACnet	AROB Universal Room Controller	Industry Controller	Industry Controller	Pass ✓
BACnet	BACnet Dewpoint Transmitter	Industry Environment Sensor	Industry Environment Sensor	Pass ✓
BACnet	BAC-RI Room Interface Module	Industry Controller	Industry Controller	Pass ✓
BACnet	BACnet Room Pressure Monitor	Industry Environment Sensor	Health Monitor	Fail ✗
BACnet	SRI-70 Analogue Room Interfaces	Industry Controller	Industry Controller	Pass ✓
BACnet	CDD3 CO2 Detector	Industry Environment Sensor	Industry Environment Sensor	Pass ✓
BACnet	CDR-BAC Room CO2/Temp Sensor	Industry Environment Sensor	Industry Environment Sensor	Pass ✓
BACnet	SRC-100 Series Zone Controllers	Industry Controller	Industry Controller	Pass ✓
BACnet	Touchplate Light Controller	Industry Controller	Industry Controller	Pass ✓

D UNIT TESTS PERFORMED

Device Tests	Passed
PhilipsHuewhiteInSet	✓
AppleLukePhoneInSet	✓
ApplePhoneInSetWithAppleStar	✓
ApplePhoneInSetWithStarStar	✓
StarStarInSetWithAppleLukePhone	✓
StarStarInSetWithAppleStar	✓
DeviceIsInEmptySet	✓

Rule Tests	Passed
RuleCoreVsRuleWithNoShares	✓
isB1SharesSubsetOfB2	✓
isB2SharesSubsetOfB1	✓
isB1ProvidesSubsetOfB	✓
isB2ProvidesSubsetOfB1	✓
RuleMAIsMalformed	✓
OneRuleIsCoreRule	✓
ContractIBAreCoreRules	✓
ContractBAreCoreRules	✓

Policy Tests	Passed
PAIsConsistent	✓
PAContainsContractCB	✓
PAContainsContractCD1	✓
RemoveContractFromPolicyPA	✓
RemoveContractFromPolicyEmpty	✓
PBIsConsistent	✓
PIBIsConsistent	✓

IoTDev Tests	Passed
EmptyHasContract	✓
HasContract	✓
PolicyPFNotInserted	✓
UpdateSoftware	✓
UpdateContract	✓
UpdateContractWithoutFogNode	✓
UpdateSoftwareWithInconsistentContract	✓
UpdateContractWithInconsistentContract	✓

Service Tests	Passed
IsServOnInEmptySet	✓
IsServOnInRuleB1Provides	✓
IsServOnInRuleA2Requires	✓
IsServFakeInRuleB1Provides	✓

Contract Tests	Passed
ContractBIsConsistent	✓
ContractIBIsInconsistent	✓
ContractBMalformedRuleConsistent	✓
IllegalInfoExchangeD1ToD2	✓
IllegalInfoExchangeD2ToD1	✓
LegalInfoExchangeRuleA3ToD2	✓
LegalInfoExchangeRuleD2ToA3	✓

FogNode Tests	Passed
RefuseContractIBWithPolicyPA	✓
AcceptContractCBWithPolicyPA	✓
AcceptContractCBWithPolicyEmpty	✓
AcceptContractB3WithPolicyPA	✓
AcceptContractD4WithPolicyPA	✓
RemoveContractCBFromPolicyPA	✓
RemoveContractA1FromPolicyPB	✓
UpdatePolicy	✓

REFERENCES

[1] Andi Adriansyah and Akhmad W. Dani. 2014. Design of small smart home system based on Arduino. In *Electrical Power, Electronics, Communications, Control and Informatics Seminar (EECCIS)*. 121–125. DOI:https://doi.org/10.1109/EECCIS.2014.7003731

[2] Akash Agarwal, Samuel Dawson, Derrick McKee, Patrick Eugster, Matthew Tancreti, and Vinaitheerthan Sundaram. 2017. Poster abstract: Detecting abnormalities in IoT program executions through control-flow-based features. In *IEEE/ACM 2nd International Conference on Internet-of-Things Design and Implementation (IoTDI)*. 339–340.

[3] Abduljaleel Al-Hasnawi, Ihab Mohammed, and Ahmed Al-Gburi. 2018. Performance evaluation of the policy enforcement fog module for protecting privacy of IoT data. In *IEEE International Conference on Electro/Information Technology (EIT)*. 0951–0957. DOI:https://doi.org/10.1109/EIT.2018.8500157

[4] Arduino. 2018. Memory | Arduino. Retrieved from https://www.arduino.cc/en/Tutorial/Foundations/Memory.

[5] Arjun P. Athreya, Bruce DeBruhl, and Patrick Tague. 2013. Designing for self-configuration and self-adaptation in the internet of things. In *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 585–592. DOI:https://doi.org/10.4108/icst.collaboratecom.2013.254091

[6] Arani Bhattcharya and Pradipta De. 2016. Computation offloading from mobile devices: Can edge devices perform better than the cloud? In *3rd International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC'16)*. Association for Computing Machinery, New York, NY, 1–6. DOI:https://doi.org/10.1145/2962564.2962569

[7] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. 2014. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, Nik Bessis and Ciprian Dobre (Eds.). Springer International Publishing, Cham, 169–186. DOI:https://doi.org/10.1007/978-3-319-05029-4_7

[8] Ismail Butun, Alparslan Sari, and Patrik Österberg. 2019. Security implications of fog computing on the internet of things. In *IEEE International Conference on Consumer Electronics (ICCE)*. 1–6. DOI:https://doi.org/10.1109/ICCE.2019.8661909

[9] ISO/IEC JTC 1 Technical Committee. 2016. ISO/IEC 20922:2016 Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1. Retrieved from https://www.iso.org/standard/69466.html.

[10] ISO/IEC JTC 1/SC 22 Technical Committee. 2017. ISO/IEC 21778:2017 Information technology – The JSON data interchange syntax. Retrieved from https://www.iso.org/standard/71616.html.

[11] Mauro Conti, Ali Dehghantanha, Katrin Franke, and Steve Watson. 2018. Internet of things security and forensics: Challenges and opportunities. *Fut. Gen. Comput. Syst.* 78 (2018), 544–546. DOI:https://doi.org/10.1016/j.future.2017.07.060

[12] Nicola Dragoni, Olga Gadyatskaya, and Fabio Massacci. 2010. Supporting applications' evolution in multi-application smart cards by security-by-contract. In *4th Workshop in Information Security Theory and Practices (WISTP'10)*. Springer LNCS.

[13] Nicola Dragoni, Alberto Giaretta, and Manuel Mazzara. 2017. The internet of hackable things. In *5th International Conference in Software Engineering for Defence Applications*, Paolo Ciancarini, Stanislav Litvinov, Angelo Messina, Alberto Sillitti, and Giancarlo Succi (Eds.). Springer, 129–140.

[14] Nicola Dragoni, Fabio Massacci, Katsiaryna Naliuka, and Ida Siahaan. 2007. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Public Key Infrastructure Conference (PKI'07)*, Javier Lopez, Pierangela Samarati, and Josep L. Ferrer (Eds.). Springer, 297–312.

[15] Nicola Dragoni, Fabio Massacci, Thomas Walter, and Christian Schaefer. 2009. What the heck is this application doing? A security-by-contract architecture for pervasive services. *Comput. Secur.* 28, 7 (Oct. 2009), 566–577. DOI:https://doi.org/10.1016/j.cose.2009.06.005

[16] Clinton Dsouza, Gail-Joon Ahn, and Marthony Taguinod. 2014. Policy-driven security management for fog computing: Preliminary framework and a case study. In *IEEE 15th International Conference on Information Reuse and Integration (IRI'14)*. 16–23. DOI:https://doi.org/10.1109/IRI.2014.7051866

[17] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. 2015. A search engine backed by internet-wide scanning. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. Association for Computing Machinery, New York, NY, 542–553. DOI:https://doi.org/10.1145/2810103.2813703

[18] Edlira Dushku, Masoom Rabbani, Mauro Conti, Luigi V. Mancini, and Silvio Ranise. 2020. SARA: Secure asynchronous remote attestation for IoT systems. *IEEE Trans. Inf. Forens. Secur.* 15 (2020), 3123–3136. DOI:https://doi.org/10.1109/TIFS.2020.2983282

[19] Xuan Feng, Qiang Li, Haining Wang, and Limin Sun. 2018. Acquisitional rule-based engine for discovering internet-of-things devices. In *27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 327–341. Retrieved from https://www.usenix.org/conference/usenixsecurity18/presentation/feng.

[20] Francisco Javier Ferrández-Pastor, Higinio Mora, Antonio Jimeno-Morenilla, and Bruno Volckaert. 2018. Deployment of IoT edge and fog computing technologies to develop smart building services. *Sustainability* 10, 11 (2018). DOI:https://doi.org/10.3390/su10113832

[21] David Fisher, Bernhard Isler, and Michael Osborne. 2019. BACnet Secure Connect: A secure infrastructure for building automation. *AHRAE BACnet whitepaper* 21 (2019). Retrieved from https://www.ashrae.org/File20Library/Technical20Resources/Bookstore/BACnet-SC-Whitepaper-v10_Final_20180710.pdf.

[22] Eclipse Foundation. 2020. Eclipse Mosquitto—An open source MQTT broker. Retrieved from https://github.com/eclipse/mosquitto.

[23] Iván Froiz-Míguez, Tiago M. Fernández-Caramés, Paula Fraga-Lamas, and Luis Castedo. 2018. Design, implementation and practical evaluation of an IoT home automation system for fog computing applications based on MQTT and ZigBee-WiFi sensor nodes. *Sensors* 18, 8 (2018). DOI:https://doi.org/10.3390/s18082660

[24] Alberto Giaretta. 2020. S×C4IoT. Retrieved from https://github.com/albertogiaretta/SxC4IoT.

[25] Alberto Giaretta, Nicola Dragoni, and Fabio Massacci. 2019. IoT security configurability with security-by-contract. *Sensors* 19, 19 (Sept. 2019), 4121.

[26] Alberto Giaretta, Nicola Dragoni, and Fabio Massacci. 2019. Protecting the internet of things with security-by-contract and fog computing. In *IEEE 5th World Forum on Internet of Things (WF-IoT)*. 1–6.

[27] Laurence Goasduff. 2019. Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io.

[28] Google. 2020. A Java serialization/deserialization library to convert Java Objects into JSON and back. Retrieved from https://github.com/google/gson.

[29] Larry K. Haakenstad. 1999. The open protocol standard for computerized building systems: BACnet. In *IEEE International Conference on Control Applications*, Vol. 2. IEEE, New York, NY, 1585–1590.

[30] Ayyoob Hamza, Hassan Habibi Gharakheili, and Vijay Sivaraman. 2018. Combining MUD policies with SDN for IoT intrusion detection. In *Workshop on IoT Security and Privacy*. 1–7. DOI:https://doi.org/10.1145/3229565.3229571

[31] Ayyoob Hamza, Dinesha Ranathunga, Hassan Habibi Gharakheili, Matthew Roughan, and Vijay Sivaraman. 2018. Clear as MUD: Generating, validating and applying IoT behavioral profiles. In *Workshop on IoT Security and Privacy*. 8–14. DOI:https://doi.org/10.1145/3229565.3229566

[32] Josh Juneau, Jim Baker, Victor Ng, Leo Soto, and Frank Wierzbicki. 2010. The Definitive Guide to Jython. Retrieved from https://jython.readthedocs.io/en/latest/.

[33] Sebastian Kaebisch, Takuki Kamiya, Michael McCool, Victor Charpenay, and Matthias Kovatsch. 2020. Web of Things (WoT) Thing Description. Retrieved from https://www.w3.org/TR/wot-thing-description/.

[34] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A security architecture for tiny embedded devices. In *9th European Conference on Computer Systems (EuroSys'14)*. Association for Computing Machinery, New York, NY. DOI:https://doi.org/10.1145/2592798.2592824

[35] Jarkko Kuusijärvi, Reijo Savola, Pekka Savolainen, and Antti Evesti. 2016. Mitigating IoT security threats with a trusted Network element. In *11th International Conference for Internet Technology and Secured Transactions (ICITST)*. 260–265. DOI:https://doi.org/10.1109/ICITST.2016.7856708

[36] Eliot Lear and Brian Weis. 2016. Slinging MUD: Manufacturer usage descriptions: How the network can protect things. In *International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT)*. 1–6.

[37] Roger Light. 2017. Mosquitto: Server and client implementation of the MQTT protocol. *J. Open Source Softw.* 2, 13 (May 2017), 265.

[38] M. Veeramanikandan and Suresh Sankaranarayanan. 2019. Publish/subscribe based multi-tier edge computational model in Internet of Things for latency reduction. *J. Parallel Distrib. Comput.* 127 (2019), 18–27. DOI:https://doi.org/10.1016/j.jpdc.2019.01.004

[39] Zaigham Mahmood. 2018. *Fog Computing: Concepts, Frameworks and Technologies*. Springer International Publishing, Cham.

[40] Rwan Mahmoud, Tasneem Yousuf, Fadi A. Aloul, and Imran A. Zualkernan. 2015. Internet of things (IoT) security: Current status, challenges and prospective measures. In *10th International Conference for Internet Technology and Secured Transactions (ICITST)*. 336–341.

[41] Neda Maleki, Mohammad Loni, Masoud Daneshtalab, Mauro Conti, and Hossein Fotouhi. 2019. SoFA: A spark-oriented fog architecture. In *45th Annual Conference of the IEEE Industrial Electronics Society*, Vol. 1. 2792–2799. DOI:https://doi.org/10.1109/IECON.2019.8927065

[42] Sara N. Matheu, José L. Hernández-Ramos, Salvador Pérez, and Antonio F. Skarmeta. 2019. Extending MUD profiles through an automated IoT security testing methodology. *IEEE Access* 7 (Oct. 2019), 149444–149463.

[43] Sara N. Matheu-García, José L. Hernández-Ramos, Antonio F. Skarmeta, and Gianmarco Baldini. 2019. Risk-based automated assessment and testing for the cybersecurity certification and labelling of IoT devices. *Comput. Stand. Interf.* 62 (Feb. 2019), 64–83. DOI:https://doi.org/10.1016/j.csi.2018.08.003

[44] Guðni Matthìasson, Alberto Giaretta, and Nicola Dragoni. 2020. IoT device profiling: From MUD files to S×C contracts. In *Open Identity Summit*. 143–154. DOI:https://doi.org/10.18420/ois2020_12

[45] Argyro Mavrogiorgou, Athanasios Kiourtis, and Dimosthenis Kyriazis. 2017. A comparative study of classification techniques for managing IoT devices of common specifications. In *Economics of Grids, Clouds, Systems, and Services*, Congduc Pham, Jörn Altmann, and José Ángel Bañares (Eds.). Springer International Publishing, Cham, 67–77.

[46] George C. Necula. 1997. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. 106–119. DOI:https://doi.org/10.1145/263699.263712

[47] George C. Necula. 2000. Translation validation for an optimizing compiler. *SIGPLAN Not.* 35, 5 (May 2000), 83–94. DOI:https://doi.org/10.1145/358438.349314

[48] George C. Necula and Peter Lee. 1996. Safe kernel extensions without run-time checking. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. 229–243. DOI:https://doi.org/10.1145/238721.238781

[49] George C. Necula and Peter Lee. 2004. The design and implementation of a certifying compiler. *SIGPLAN Not.* 39, 4 (Apr. 2004), 612–625. DOI:https://doi.org/10.1145/989393.989454

[50] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake

Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* 12, 85 (Oct. 2011), 2825–2830.

[51] Paul Pop, Bahram Zarrin, Mohammadreza Barzegaran, Stefan Schulte, Sasikumar Punnekkat, Jan Ruh, and Wilfried Steiner. 2021. The FORA fog computing platform for industrial IoT. *Inf. Syst.* 98 (2021), 101727. DOI:https://doi.org/10.1016/j.is.2021.101727

[52] Ammar Rayes and Samer Salam. 2019. *Internet of Things from Hype to Reality: The Road to Digitization.* Springer International Publishing, Cham.

[53] Ramachandran Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and Scott A. Smolka. 2001. Model-carrying code (MCC): A new paradigm for mobile-code security. In *Workshop on New Security Paradigms (NSPW'01).* 23–30. DOI:https://doi.org/10.1145/508171.508175

[54] Ramachandran Sekar, Venkat N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. 2003. Model-carrying code: A practical approach for safe execution of untrusted applications. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 15–28. DOI:https://doi.org/10.1145/1165389.945448

[55] Arbia Riahi Sfar, Enrico Natalizio, Yacine Challal, and Zied Chtourou. 2018. A roadmap for security challenges in the internet of things. *Dig. Commun. Netw.* 4, 2 (2018), 118–137. DOI:https://doi.org/10.1016/j.dcan.2017.04.003

[56] Kewei Sha, Wei Wei, T. Andrew Yang, Zhiwei Wang, and Weisong Shi. 2018. On security challenges and open issues in internet of things. *Fut. Gen. Comput. Syst.* 83 (2018), 326–337. DOI:https://doi.org/10.1016/j.future.2018.01.059

[57] Nico Surantha and Wingky R. Wicaksono. 2018. Design of smart home security system using object recognition and PIR sensor. *Procedia Comput. Sci.* 135 (2018), 465–472. DOI:https://doi.org/10.1016/j.procs.2018.08.198

[58] Mathias Dahl Thomsen. 2019. *Device-b Access Control.* Master's thesis. Danmarks Tekniske Universitet, Denmark. Retrieved from https://findit.dtu.dk/en/catalog/2452038023.

[59] Mathias Dahl Thomsen, Alberto Giaretta, and Nicola Dragoni. 2020. Smart lamp or security camera? Automatic identification of IoT devices. In *12th International Network Conference (INC'20).*

[60] Yiji Zhang and Lenore D. Zuck. 2018. Formal verification of optimizing compilers. In *Distributed Computing and Internet Technology (LNCS).* 50–65.

[61] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. 2014. IoT Security: Ongoing challenges and research opportunities. In *IEEE 7th International Conference on Service-oriented Computing and Applications.* 230–234.

[62] Wei Zhou, Yan Jia, Anni Peng, Yuqing Zhang, and Peng Liu. 2019. The effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. *IEEE Internet Things J.* 6, 2 (Apr. 2019), 1606–1616. DOI:https://doi.org/10.1109/JIOT.2018.2847733

Received November 2020; revised July 2021; accepted August 2021