


Article

Implementation of a Partial-Order Data Security Model for the Internet of Things (IoT) Using Software-Defined Networking (SDN)

Abdelouadoud Stambouli ^{1,*} and Luigi Logrippo ^{1,2} 

¹ Département d'informatique et d'ingénierie, Université du Québec en Outaouais, Gatineau, QC J8X 3X7, Canada; luigi@uqo.ca

² School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, ON K1N 6N5, Canada

* Correspondence: staa16@uqo.ca

Abstract: Data security on the Internet of Things (IoT) is usually implemented through encryption. This paper presents a solution based on routing, in which data are forwarded only to entities that are intended to receive them according to security requirements of secrecy (also called confidentiality), integrity, and conflicts. Our solution is generic in the sense that it can be used in any network, together with encryption as appropriate. We use the fact that, in any network, security requirements generate a partial order of equivalence classes of entities, and each entity can be labeled according to the position of its equivalence class in the partial order. Routing tables among entities can be compiled using the labels. The method is demonstrated in this paper for software-defined networking (SDN) routers and controllers. We propose a centralized IoT architecture with a cloud structure using SDN as networking infrastructure, where storage entities (i.e., cloud servers) are associated with application entities. A small 'hospital' example is shown for illustration. Procedures for network reconfigurations are presented. We also demonstrate the method for the normal case where different partial orders, representing distinct but concurrent security requirements, coexist among a set of entities. The method proposed does not impose an overhead on the normal functioning of SDN networks since it requires calculations only when the network must be reconfigured because of administrative intervention or policies. These occasional updates can be done efficiently and offline.



Citation: Stambouli, A.; Logrippo, L. Implementation of a Partial-Order Data Security Model for the Internet of Things (IoT) Using Software-Defined Networking (SDN). *J. Cybersecur. Priv.* **2024**, *4*, 468–493. <https://doi.org/10.3390/jcp4030023>

Academic Editor: Danda B. Rawat

Received: 13 May 2024

Revised: 17 July 2024

Accepted: 18 July 2024

Published: 20 July 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: Internet of Things (IoT); software-defined networking (SDN); data and information security; data flow control; access control; secrecy-confidentiality-integrity

1. Introduction and Motivation

In this paper, we see the Internet of Things (IoT) as a set of entities that contain data, are interconnected by directional data channels, and can modify their data contents by transferring data among themselves over the channels. Entities and channels can change over time. However, it is always necessary that certain data security requirements be respected. Data security is concerned with the three aspects of *secrecy* (often called *confidentiality*), *integrity*, and *availability* [1]. Secrecy means that only authorized entities can read certain data; integrity means that only certain entities can write them; and availability means that all authorized entities can access them. In this paper, we also consider 'conflict' requirements. These are found where it is required that certain entities should not be allowed to know (i.e., read) certain combinations of data (see the concept of the Chinese Wall in [1]). Conflict requirements are secrecy requirements but are applied to combinations of data rather than to single data items. We present a method to satisfy all these requirements by appropriately programming routers in a software-defined networking and 'cloud' architecture.

Data security, information security, and data privacy are major concerns in the IoT; see Alaba et al. [2], Singh et al. [3], and Qiang et al. [4]. The importance of data flow security for data privacy is discussed by Landwehr in [5].

Encryption is the established method to protect data in the IoT, and it can be expected to remain so. However, routing, with its ability to prevent data from reaching certain entities or to direct data towards certain entities, has received much less attention. This paper focuses on routing and proposes a method for using it, possibly as a complement to encryption, especially in cases where encryption is unfeasible or impractical (e.g., if the network's entities have very limited computing capabilities), in order to further strengthen data security.

Software-defined networking (SDN) is a networking architecture that facilitates the establishment of global routing strategies. In SDN, packets can be forwarded or dropped. However, no research exists on the use of SDN for the satisfaction of the security requirements mentioned above. Most security research related to SDN is concerned with infrastructure vulnerabilities.

Our approach to using SDN for data security is based on data labeling, combined with policies stating that only entities labeled in certain ways can receive data from entities labeled in certain other ways. Data labeling can be used to specify constraints on passing data among entities as a general-purpose data flow control method. Research towards the use of labeling in security exists (see Section 2), but not in connection with the global routing capabilities of SDN. We show that SDN routing tables can be compiled for arbitrary networks by using labels constructed according to an efficient method.

In Section 2, we provide a literature review. In Section 3, we give a short account of previous research on data flow control for security and an introduction to software-defined networking. Section 4 describes our method in principle. Section 5 presents a concrete 'hospital' example. Section 6 discusses the topic of network reconfigurations. Section 7 shows how networks with multiple flows can be implemented in our method. Section 8 presents how our method was simulated and tested. Section 9 deals with efficiency and scalability. Section 10 concludes the paper.

2. Related Work

We have drawn inspiration from the work by Etalle et al. [6], where a function *Tag* is defined that maps subjects or objects to the set of tags assigned to them, and where a security administrator can formulate logic-based authorization policies that define access rights in terms of these tags. This work extends work by Hinrichs et al. [7]. An important difference between our approach and the one of [6,7] is that our labels can be calculated automatically, while their tags are assigned by users. These authors also introduce expressive policy languages that are still under development for our approach. In Singh et al. [3], entities and data are labeled with two labels, one for secrecy and another for integrity, and security policies are defined in such a way that data from entities can only flow into other entities labeled to receive them. In our approach, we generalize this idea by using labels that can be computed in any network, and we take advantage of the observation of Sandhu [8], in which only one label is required to express both the secrecy and integrity of data.

In a recent paper, Burke et al. [9] propose *MLS-Enforcer*, a software-defined networking (SDN) controller that enforces multi-level policies while retaining the ability to securely relabel network nodes under changing configurations and network traffic demands; this is done by using a polynomial-time heuristic relabeling algorithm. From the security point of view, their method is much less general than our method, being restricted to a "Relaxed Bell-La Padula" model, and the labels used are heuristic and more complex than ours. However, they consider traffic demands that are outside of our scope. Future research can deal with combining the ideas of this paper with those of ours, possibly leading to more general results.

Some papers propose the use of different types of access control and data flow control policy models in the IoT; for example, Xie et al. [10] propose the use of provenance-based data flow control (PDFC), defined by complex authorization rules. Our policy model is simpler, covers both access and flow control, has well-defined concepts of secrecy and

integrity, and can be directly applied to SDN. It also expresses provenance to the extent that our labels can express provenance.

Al-Haj and Aziz [11] present a solution to enforce security policies to control the routing configuration in database-defined networks. To achieve this, the authors use row-level security checks and the lattice-based model [8,12] alongside the RAVEL architecture (Wang et al. [13]). Their solution consists of constructing routing tables by using the lattice model, encoding the tables in the database-defined network architecture of RAVEL, and enforcing multi-level security policies using row-level security as an enforcement mechanism. The authors deal separately with secrecy and integrity. To enforce the upward flow of data, the authors propose to define the flow path, in the *Can Flow* table. This path consists of sequences of nodes that data can flow into. Once a path is defined, each node in this path starting from the first one, will be given a security label. Finally, a security policy is defined with respect to a multi-level model, which states that data can only flow upward from a security level into a higher one. The enforcement of downward data flow for integrity is dual. Our work generalizes the work done in this paper in at least two directions, namely that it can be applied to any network rather than lattice-based ones and that our labels encode simultaneously secrecy and integrity requirements. One idea we retain for further research is using a database approach to represent data flow policies.

Fernandes et al. [14] and Celik et al. [15] use data flow tracking within the system to enforce fine-grained security policies using a combination of dynamic and static analysis. We take another approach, which is labeling data. This can be simpler, compatible with existing infrastructure, and efficient, but it can have limited granularity. Assigning labels to data packets is a straightforward process that does not require significant computational resources or expertise. This approach can be compatible with existing IoT systems since many IoT systems already use some form of data labeling, such as metadata or tags, to identify and manage data. It can also be more efficient than tracking the flow of data in systems with limited resources. Labeling data can have limited granularity, and its appropriateness depends on the context. For example, labeling an entire data packet with a single label can be an efficient approach if the packet contains multiple data items of similar sensitivity.

Haotian et al. [16] focus on smart home environments, and their approach may be less applicable to other IoT domains. Our approach can be applied to any IoT domain beyond smart homes and proposes a solution for securely sharing data between different IoT domains. Our approach also provides a comprehensive analysis of the requirements for IoT data flow configuration and presents a detailed architecture for implementation using SDN.

Papers dealing with SDN security usually tackle security factors such as the exchange and deployment of security policies within the network in the case of SDN domains, intrusion detection, security against external attacks, etc. [17,18]. Several of the proposed security solutions use cryptographic algorithms that may be difficult to implement in sensors, and in any case, they can be used in conjunction with our method. As mentioned, our approach does not require cryptography, although cryptography can be used in combination with it. Several of the reviewed papers are short and present only ideas for solutions. Many do not concentrate on data security. Our work is the first to present a data flow control method for security based on SDN routing, and thus a direct comparison of results is not possible.

Surely, some of the techniques proposed may be compatible with our approach and, in combination with it, may lead to efficiency and security improvements; this will be the subject of further research.

3. Preliminaries

3.1. Data Security Concepts

In classical papers by Denning [12] and Sandhu [8], which were preceded by the work of Bell-La Padula [19] and followed by many others concurring with their basic ideas, a concept of *secure information flow* was introduced, in which data are expected to flow

according to the order relations in a lattice of labeled entities. We adapt the basic definitions from this literature as follows: For entities x and y , we say that there is a *Channel* from x to y if entity x has permission to write data on y or entity y has permission to read data from x , and graphically, we represent this as an arrow from x to y . Terms such as *receiving* or *pulling* can also be used instead of *reading* and *sending* or *pushing* instead of *writing*.

The *CanFlow* or *CF* relation is defined as the transitive, reflexive closure of the *Channel* relation and graphically corresponds to directed paths between entities. Any network of entities that are represented as a directed graph in this manner is a *preorder* that can be reduced to a *partial order* of equivalence classes of entities, where two entities x and y belong to the same equivalence class iff $CF(x,y)$ and $CF(y,x)$ [20]. In other words, a set of entities is in the same equivalence class if the data that is in any one of them can reach any other through a directed path; further, the set of the resulting equivalence classes forms a partial order.

We define the *label* of x , $Lab(x)$, as the set of the names of all entities that can flow to x . So, if x , y , and z are the names of all entities that can flow to w beside w itself, then $Lab(w) = \{x,y,z,w\}$. Entities in the same equivalence class have the same label. It follows that $CF(x,y)$ iff $Lab(x) \subseteq Lab(y)$, and so labels, with the inclusion relation between them, define the flow relation between entities in the partial order.

Following the usual terminology in data security theory, we call entity names *categories*, and we note that each category also defines a data *provenance*. Thus, the label of an entity represents the data categories that the entity *can know*. It is important to note that, given a set of reading and writing permissions, as they could be represented in an access control matrix or in a graph, the labels can be computed efficiently by using well-known graph component algorithms [21].

Following Sandhu [8], we say that the top entities in a partial order, not having any outgoing data flows, have *top secrecy*, while the bottom entities, not having any incoming data flows, have *top integrity*. Top secrecy corresponds to minimum integrity, and vice versa. By extension, entities can be considered to have different levels of secrecy or integrity according to their position in the partial order. In addition, situations of conflict (such as the ones represented by *Chinese Walls*) can be represented by excluding certain label combinations; e.g., if no entity is supposed to know both data of category x and category y , then labels containing both x and y are forbidden (again, this is consistent with Sandhu [8]). Studies [20,21] have shown that these definitions can be used not only for lattice networks but also for any networks that can be specified by means of access control matrices or permission lists, including the widely implemented role-based access control (RBAC).

Thus, a main difference between the theory presented in Denning [12] and Sandhu [8] and the theory used here can be expressed as follows: while the former states that secure data flows must be built by forming lattices of labeled entities, the latter theory shows that for any preorder of entities, representing an arbitrary data flow in a network at a given time, the equivalence classes of the entities form a partial order, and the entities in the partial order can be considered to have different levels of secrecy or integrity according to the position of their equivalence classes in the partial order. In the former theory, a lattice structure is a precondition for secure data flows, while in the theory we have described, any network (lattice-structured or not) has its own security properties for secrecy, integrity, and conflicts.

3.2. Software-Defined Networking (SDN)

Just like the IoT, SDN is a networking technology introduced at the beginning of this century. The literature on SDN is abundant; we mention some points in this section for completeness.

SDN is an evolution of the classic network model into a network defined by applications. SDN architecture separates the network control (control plane) and forwarding functions (data plane), enabling the network control to become directly programmable and centrally managed. This programming is done via SDN controllers instead of classical

Internet protocols. Centralization allows the controller to maintain a global view of the network and control it through standards such as OpenFlow, which is a protocol defined by the Open Networking Foundation to transfer forwarding rules from the controllers to the routers using APIs. We use in our work the most common way of programming SDN networks, where applications give abstract rules to controllers, which translate them into commands for the network equipment concerned, the SDN router.

To justify our choice of the SDN architecture, we start with the observation that global security solutions are more efficient and effective when they are centralized, as SDN is. Further, SDN is a system designed for efficient networking, so its use for data security will be efficient. Finally, we will see that SDN allows a straightforward translation of our labels into rules for controllers and then routers. Many types of controllers and routers exist in practice, but our approach appears to be feasible for any of them. The use of SDN limits our approach to centrally controlled IoT systems, but the resulting efficiency makes it valuable in such contexts.

Reviews of SDN and its use for security can be found in several papers, e.g., Huang et al. [22], a review paper that focuses on the use of SDN specifically for security in the IoT. Other research that proposes SDN-based security frameworks for the IoT has been mentioned in Section 2. However, the main concerns of this literature are the management and deployment of security policies, identity management, and detection or prevention of vulnerabilities, intrusions, or attacks; these subjects are outside of this paper's scope.

SDN has been adopted in various systems and contexts, both in research and industrial practice. Google uses SDN in its B4 WAN [23] to optimize bandwidth usage and resource allocation between its data centers. Microsoft employs SDN in its Azure cloud platform [24] to manage network resources efficiently, providing scalable and reliable cloud services. GENI [25] is a research initiative that uses SDN to create a flexible and programmable testbed for experimenting with future internet architectures and services. Researchers use GENI to explore new networking paradigms and applications. Several other applications of the SDN concept are found in theory and practice, hence the importance of our research.

4. Our System Design

4.1. Network Configurations and Graphic Representation

Following on the principles presented in Section 3, the main idea of this paper is that the labels of entities, which can be computed automatically and efficiently from the *Channel* relation [20,21], can be used for routing data in SDN networks, from the entities where data originates, their provenance, to all the entities where they can flow. By using SDN configured according to the partial order security model, it is possible to constrain the flow of data in IoT systems to satisfy data security requirements. On this basis, we formulate an SDN architecture where SDN forwarding tables are set by the SDN controllers using the entities' labels. It follows from the theory presented in Section 3 that this method is general in the sense that it can be used for any network for which the data flow relation can be represented as a directed graph of entities. Our method considers a centralized IoT architecture where all data are transferred and stored on cloud platforms and accessed by user applications.

We chose to work on a centralized IoT architecture with a cloud structure using SDN as communication infrastructure. Several papers in the literature propose centralized configurations for IoT security, such as Christos et al. [26], Hany et al. [27], and Roy et al. [28]. Further, our efficient centralized algorithms can reconfigure networks dynamically as necessary; see Section 6. SDN will work very well in closed systems such as hospitals, industrial plants, smart homes, and the like since its architecture is well-conceived for scalability and efficiency.

In the Cloud, a data container and a server can be two distinct entities interconnected via the network. For simplicity, we chose to represent them as a single entity. In centralized IoT systems, all devices are connected through centralized cloud servers, and communication between different devices must be achieved through these servers. This

IoT configuration consists of three main layers: *Sensing layer*, *Networking layer*, and *Application layer*. The *Sensing layer* consists of different types of sensors, RFIDs, and other data-collecting devices. This layer collects data from the environment and sends it to the cloud servers via centralized gateways. Entities requiring high integrity are found in this layer. The *Application layer* involves various IoT applications that use the data collected by sensors in contexts such as healthcare systems, smart cities, etc. High-secrecy entities are found in this layer. The Cloud constitutes the IoT *Networking layer* and all communication passes through it.

The *Networking layer* connects IoT objects to the Internet and contains the servers used to store the data collected from the *Sensing layer*. Several communications technologies and protocols can be used in this layer, such as 3G/4G/5G, Zigbee, Bluetooth, WiFi to transport data from the *Sensing layer* to the *Application layer* on the one hand and inside the *Networking layer* between the servers on the other hand. Our solutions are oriented towards Wi-Fi since, with this technology, every entity or object in the system will have an IPAD (or IP address) that identifies it. This simplifies our presentation, but our approach can be extended.

We adapt the centralized IoT architecture to the SDN architecture; see Figure 1. The Controller will have two routers to take care of: the first router is the Cloud router, which interconnects the servers in the Cloud, implementing the *Networking layer*. The second router is the *Application router*, to which the cloud router connects, and which interconnects the entities in the *Application layer*. We also have an Access point that connects the *Sensing layer* with the *Networking layer*, but we do not program this one, since its function is simply forwarding the data to the cloud router. These are logical devices that can be implemented by several physical devices of different types.

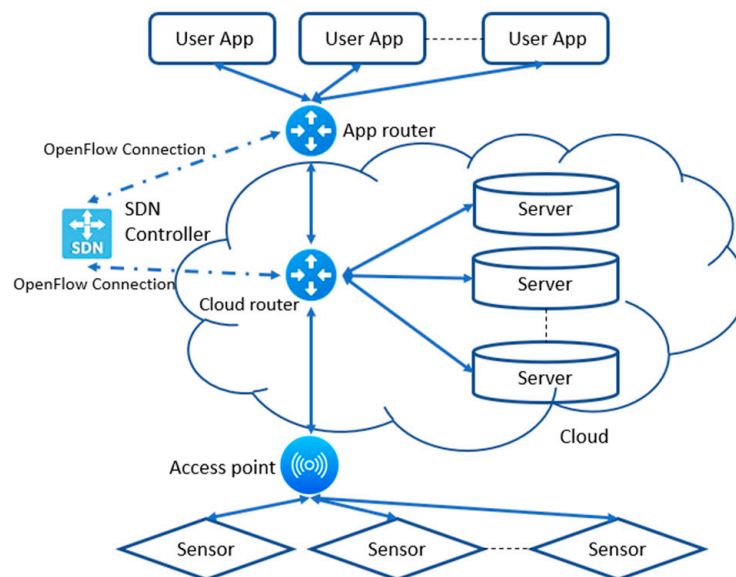


Figure 1. Our SDN implementation configuration.

Many papers in the literature mention a single controller for wide-area SDN. In ElGaroui et al. [29] and Dias et al. [30], the authors use the same controller as us (Ryu controller) to control multiple routers in their wide-area SDN. The constraints on the physical placement of the servers and of the application entities will depend on factors such as the type of controllers and routers used; for example, hierarchical controllers allow a more distributed placement. These are lower-level implementation concerns, not discussed in this paper.

4.2. Labeling Tables, Forwarding Tables, and Data Flow Control Policy Enforcement

Our method starts with a network representing an application layer configuration of directly connected application entities, defining a *Channel* relation. Following the principles presented in Section 3 and in the papers cited there, the equivalence classes of entities in the network are identified, and the resulting reduced graph is a partial order of equivalence classes. Labels are then assigned to the equivalence classes, which become the labels of the entities in each class. These labels can be simply obtained by set union, proceeding from source to sink. For example, if the label of a source equivalence class contains the name *BobPulse*, then all equivalence classes that dominate it will also contain this name in their labels. As mentioned, the necessary calculations are automatic, starting with the *Channel* relation [20].

The network obtained in this way will not be in the form of a *centralized* cloud-based configuration since application entities will be shown as communicating directly and not through the Cloud. So, the next step, in addition to the method described above, is to create the cloud infrastructure. This will be done by assigning at least one *storage entity* (in practice, a server or database) to each equivalence class of entities. Hence, in our architecture, data are sent simultaneously to application entities and their associated storage entities for permanent storage. The partial order of equivalence classes will be unchanged, with storage entities added to equivalence classes. The collection of these storage entities forms the Cloud and implements the Networking Layer of the IoT.

At this point, we note that names in labels can be replaced by entity names. For example, if a name such as *BobPulse* denotes data originating from an entity (a sensor) named *A*, then each occurrence of *BobPulse* in labels can be replaced by the name *A*. The label-inclusion relationship remains the same. These new labels based on entity names will directly provide the routing information needed to configure the SDN routers. They are compiled in labeling tables in the following way: For entities x and y , we say that $x \in Holds(y)$ iff $Label(x) \subseteq Label(y)$. A labeling table will have a line for each entity y in the network and a column *Holds* containing, for each y , the set of x such that $x \in Holds(y)$. For the controller, $x \in Holds(y)$ means that data in entity x can be forwarded to entity y . The programming of SDN routers is then immediate. Forwarding tables contain the command *forward* if a packet should be forwarded from one entity to another. For each router, we implement a forwarding table that includes the entities that are connected to it.

We assume that we deal with routers with arbitrarily large capacities. Average routers in use today can have up to 250 entities connected to them [31], but this number can be increased by connecting routers sequentially (in cascade). Many modern routers adapt automatically if a port is connected to another router. These technical details are ignored here because they depend on the technology available, which is rapidly evolving.

Of the several columns a forwarding table may have, we need only the columns *Match Rules* and *Action*. Each packet will have a source and a destination header. If in the labeling table $x \in Holds(y)$, then the controller will create in the router a flow entry using the *IPAD* (x) source (*IP src*) and *IPAD* (y) destination (*IP dst*) in match rules and define the forward action for such a pair since this is an authorized flow. When a packet arrives at the router, the router will compare the *IP src* and *IP dst* in the packet headers. If there is a forwarding rule, the router will perform it. Otherwise, the packet will be dropped. If a packet arrives at a router and the destination entity cannot be found connected to this router, the router will forward this packet to the next router in the configuration. This will prevent overloading routers and eliminate unnecessary delays. In this way, the specified partial order of equivalence classes will be implemented. Figure 2 presents a summary of our method.

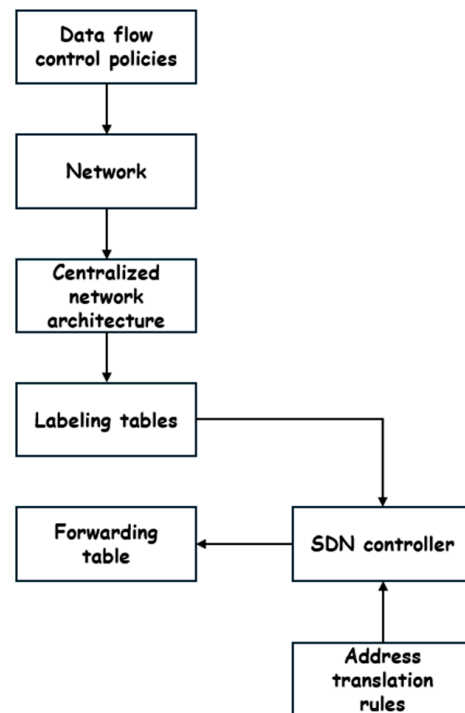


Figure 2. Summary of our method.

5. Example

5.1. The Basic Configuration and Its Implementation

As an example, we consider a very small health system. In generic terms, its configuration is as follows: there are *sensors* for patients' blood pressure and pulse. There are *wards*, each of which has *doctors* and *nurses*, and patients are assigned to wards. There is also a *Reanimation* department and a *Chief of Medicine* department, each with a workstation. Entities other than sensors are application entities. There are the following data categories: *Pressure* and *Pulse* data for each patient, and *Stat* (statistics) data for each ward. The security policies or requirements to be implemented are as follows:

- The sensors should have the highest integrity but also low secrecy since their *Pressure* and *Pulse* data are needed by all other entities. Thus, they must be at the bottom of the partial order [20], and this is where they will end up given that their labels contain only one data category.
- The *Chief of Medicine* department will have the lowest integrity since it uses data collected from all other entities, but also the highest secrecy, since it contains highly sensitive data for all patients and *Wards*. It must be at the top of the partial order [20], and this is where it will end up given that its label contains all data categories.
- The *Wards* and *Reanimation* departments take data from the *sensors*, process them, and forward the results to the *Chief of Medicine* department, they should have intermediate levels of integrity and secrecy.
- Conflicts: (a) Patient data should be known only in each patient's own *Ward* and in the *Reanimation* and the *Chief of Medicine* departments. In addition, (b) Each *Ward* keeps its own statistics that should be known only to it and to the *Chief of Medicine*. These conflicts are represented by forbidding labels containing conflicting data categories, such as, in our example, *SamPress* in *Ward2*.

We limit ourselves to an instance of this type of network where there are two *Wards* and three patients, *Sam*, *Bob*, and *Sally*, each using a sensor. It is shown in Figure 3. In the figure, rectangles represent *sensors* (three in the bottom layer) or *application entities* (six in the layers above) and include an upper-case letter for a short name of the entity, a longer descriptive name, and the label (a set of data categories) in braces. For readability, labels

have been simplified with respect to the theory presented in Section 2, but still show the data categories allowed in each entity, e.g., the workstation of *Doctor2* can contain the data for Sally and some local statistics. As in Section 3, arrows represent directional channels for receiving (reading, pulling) and sending (writing, pushing), so for example in Figure 2 entity *C* can send data to *K*, or equivalently *K* can receive data from *C*. There is an arrow, or a path of arrows (a data flow), between an entity and another iff the label of the first entity is included in the one of the second.

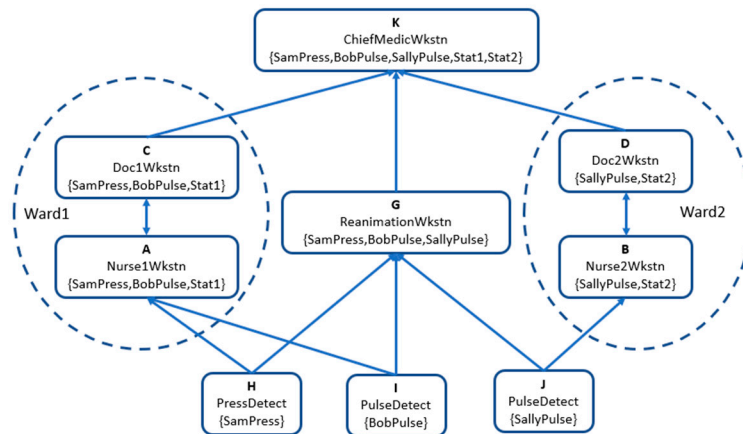


Figure 3. Hospital example.

According to IoT terminology, the three sensors are the *Sensing layer* and the rest is the *Application layer*. Figure 3 shows a ‘direct’ *Channel* configuration without the *Cloud* or the *Networking layer*. It can be checked that the security policies above are implemented by the choice of label sets, which was done by the security administrator at the time the network was configured, explicitly or implicitly by defining the *Channel* relation. For example, the blood pressure of *Sam* can only be known in *Ward1*, in the *Reanimation* or *Chief of Medicine* departments.

Figure 3 represents the network architecture that results from the data flow policies. The resulting partial order of equivalence classes in this architecture is shown in Figure 4, using letters for the names of the entities in Figure 3. Note the non-trivial equivalence classes $\{A,C\}$ and $\{B,D\}$, since the entities in Wards have symmetric channels and thus can know the same data. The other equivalence classes are singletons. The partial order of equivalence classes is shown by the arrows, e.g., $\{H\} < \{A,C\} < \{K\}$, also $\{H\} < \{G\} < \{K\}$, etc. This partial order serves as a reference point for representing the order relationships in our design. In Figure 5, we present the labeling tables for the App router in this configuration, obtained by the mentioned rule $x \in Holds(y)$ iff $Label(x) \subseteq Label(y)$ (Section 4.2). As defined in Section 3.1, $Lab(x)$ is the set of the names of all entities that can flow to x . For example, the name of entity *H* is included in the labels of all entities that can receive the data from *H*. A router programmed in this way implements all and only the flows in Figure 3, thus ensuring that the data security constraints are satisfied. For example, entity *B* (the workstation of Nurse 2) can only receive data from its equivalent entity *D* (Doctor’s 2 workstation) or entity *J*, Sally’s pulse detector, or from itself, *B* (which can be removed if desired). On the contrary, *B* cannot receive data from any of *A*, *C*, *G*, *H*, *I*, and *K* that are not in its *Holds*. *D* has the same label and thus the same *Hold* as *B* and therefore can receive from the same sources only. The SDN implementation configuration of this example is shown in Figure 6. We see in this figure that the sensors are connected to the access point, and the application entities are connected to the application router. The App Router in this figure has the routing table shown in Figure 5, and so the bidirectional arrows do not mean unconditional transfer of data in both directions; they mean transfer of data from and to the router, which will decide where to forward according to its routing table. When labeling tables are compiled into routing tables, an entity name such as *A* will be translated to $IPAD(A)$. To recapitulate, we have seen here an application of our method where the

Channel relation was used to calculate labels, the labels were used to compile the labeling table, and the labeling table was used to program the App Router, after the replacement of entity names with IP addresses.

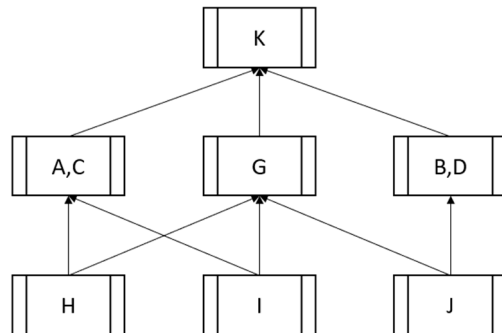


Figure 4. Partial order of equivalence classes for Figure 3.

Sensor's Labeling table	
Entity	Holds
H	H
I	I
J	J
Labeling table of the App router	
Entity	Holds
G	G, J, I, H
D	B, D, J
B	B, D, J
C	A, C, I, H
A	A, C, I, H
K	All

Figure 5. Labeling tables for the configuration of Figure 3.

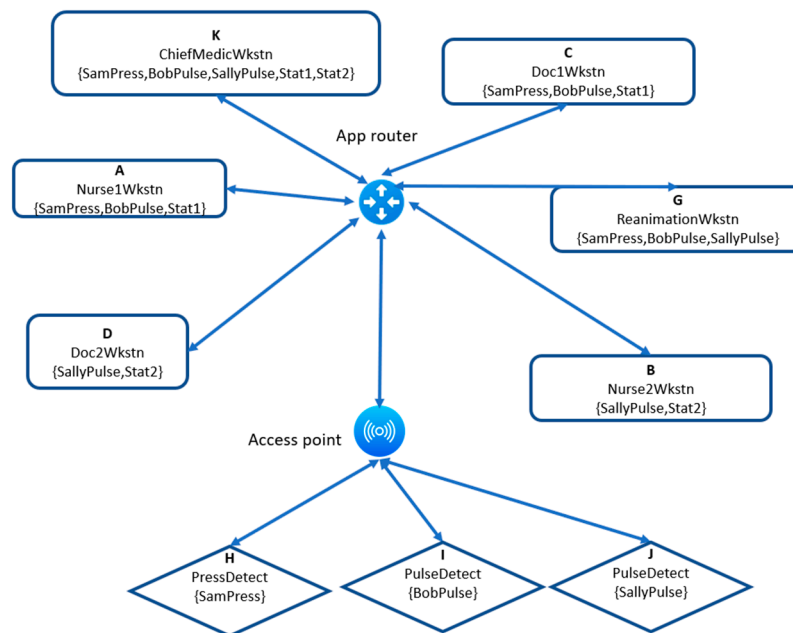


Figure 6. Implementation configuration of Figure 3.

5.2. Introducing the Networking Layer

To the configuration of Figure 3, we now add the *Cloud* and the *Networking layer* to fully implement the configuration of Figure 1. Flows between application entities must pass through the Networking layer, and so storage entities (such as databases, servers, etc.) have been added to the Cloud; hence, at least an equivalent storage entity (i.e., with the same label) is associated with each equivalence class of application entities. Storage entities are identified with primes. For example, we have added an entity G' that allows the *ReanimationWkstn*, entity G , to retrieve the data received from the sensors. We have also deleted any entity-to-entity channels that are not transited by a storage entity. Note that the required data flows between application entities (and no others) are still obtained by transitivity.

In Figure 7, we see that databases A' , B' , G' , and K' were added to function as storage entities. The partial order of equivalence classes implemented by this configuration is shown in Figure 8.

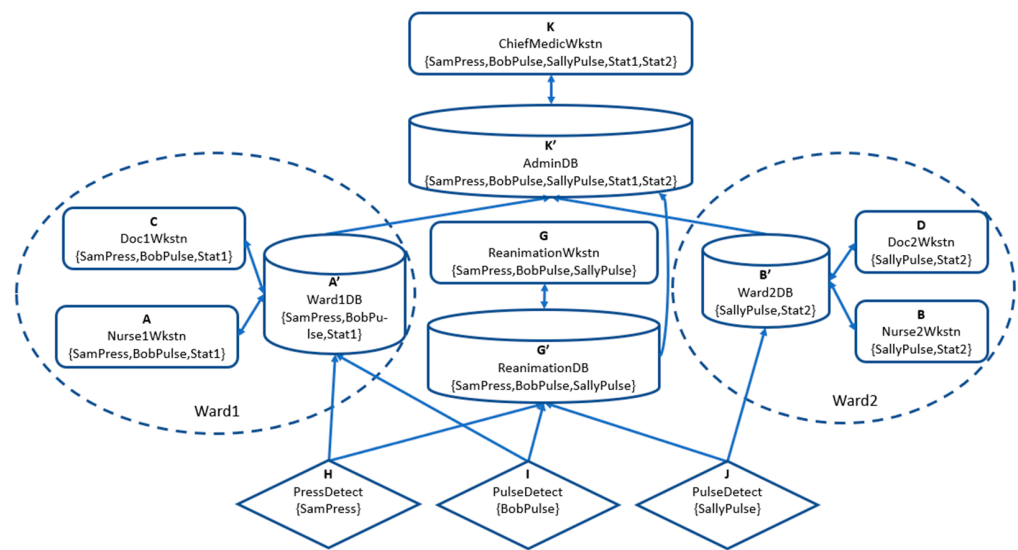


Figure 7. Cloud configuration for Figure 3.

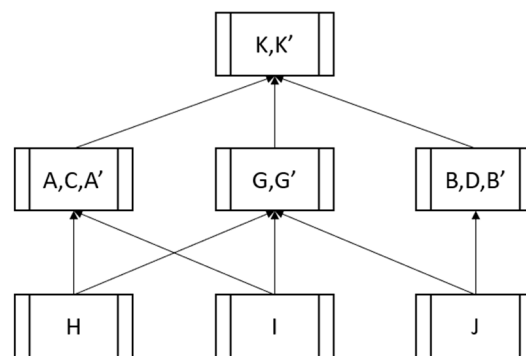


Figure 8. Partial order for the centralized architecture.

For the implementation configuration of Figure 9, the sensors are connected to access points that transfer their data to first-level cloud routers. These cloud routers forward the data to the storage entities. Finally, second-level routers are configured to connect the user endpoints to the first level of cloud routers. By adding the required routers, we obtain the configuration shown.

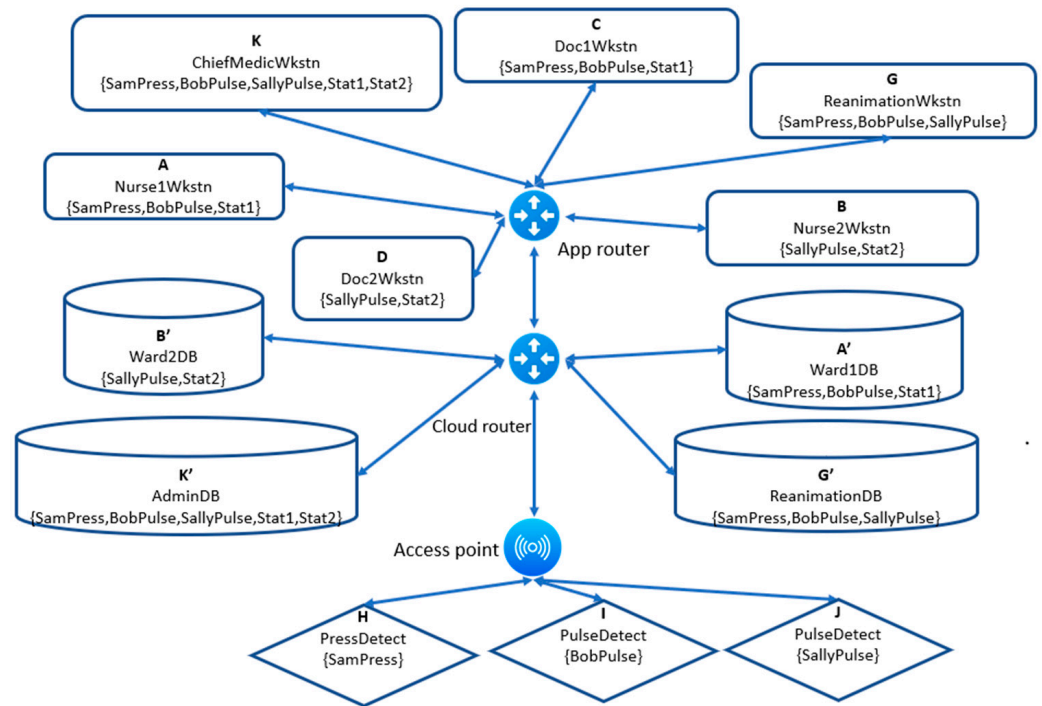


Figure 9. An implementation configuration for Figure 4.

The configuration of Figure 9 is an extension of the configuration of Figure 6, obtained by adding the mentioned new storage entities or databases and a router among them. As required, all the storage entities are connected to the *Cloud Router*, the application entities are connected to the *App Router*, and the sensors are connected to an *Access point*, which in turn is connected to the *Cloud router*, just as in Figure 7. No direct communication between application entities occurs, and all data passes through the *Cloud Router*. However, communication between storage entities is allowed to permit data flows to higher levels in partial order.

This having been done, we must configure our routers; again, we do this by constructing the labeling tables.

The *Cloud router* will have the function of allowing application entities and sensors (right column) to send data to the storage entities. So, for example, data sent from sensor *J* that detects *SallyPulse* will arrive at the *Cloud router* through the *Access point*. The router will find the rows containing *J*, which are the ones for storage entities *B'* and *G'*, and forward the data to these entities. If the destination is not found in any row of the first router, the latter will send the data to the second router. Note, for example, that $Label(J) = \{SallyPulse\}$ and $Label(B) = Label(B') = Lab(D) = \{SallyPulse, Stat2\}$. So by label inclusion, each of $\{J, B, D, B'\}$ can flow to *B'*, as specified in the labeling table. These are all and only entities whose labels are included in the label of *B'*, and so they are all and only entities whose data should be allowed to flow to *B'*, as shown in Figure 10. The table in Figure 10 can be easily constructed from the partial order of Figure 8, using the same formula $x \in Holds(y)$ iff $Label(x) \subseteq Label(y)$ that was introduced in Section 4.2.

Once the data reaches the *App Router* the same treatment is done: we check which row of the labeling table applies according to the provenance of the data, we send the data to each designated entity, and we drop the rest. In this table, the data sent to *B'* will also be available to *J, B, D, K,* and *K'*, and the data sent to *G'* will also be available to *G, K,* and *K'*. On the other hand, it is easy to check that no new flows have been added.

Sensor's Labeling table	
Entity	Holds
H	H
I	I
J	J
Labeling table of the cloud router	
Entity	Holds
B'	J, B, D, B'
A'	H, I, A, C, A'
G'	I, J, H, G, G'
K'	All
Labeling table of the App router	
Entity	Holds
G	I, J, H, G, G'
D	J, B, D, B'
B	J, B, D, B'
C	H, I, A, C, A'
A	H, I, A, C, A'
K	All

Figure 10. Labeling table for Figure 6.

Clearly, these transformations can be implemented by simple algorithms.

The final configuration of Figure 9 seems to have no relation to the partial order of equivalence classes we started from (Figure 8), the only similarity being the fact that the sensors are at the bottom layer in both. However, according to the contents of the routing tables, the data flows between entities are the same, although not along the same paths. This means the first-given policies of secrecy, integrity, and conflict are properly implemented. For example:

1. In Figure 3, we have a flow $H \rightarrow A \rightarrow C \rightarrow K$. By looking at Figures 7–9, we see that the data of H can go to A , C , and K through the *Cloud Router* and server A' .
2. Similarly, in Figure 3, we have the flow $J \rightarrow G \rightarrow K$. In the cloud configuration data can go from J to G through G' and from J to K through B' and G' .
3. On the other hand, unwanted flows are clearly impossible. For example, the reader can check easily that there is no way for data in H to end up in D or data in C to end up in D . Hence, conflict requirements are satisfied.

We see that our solution uses multicasting, through which the same data can arrive at its destination through multiple paths. Multicasting is a well-known technique in networks, and methods exist to eliminate duplicate packets received from several sources. It makes our solution fault-tolerant to some extent, e.g., in the second example, the failure of one of B' or G' will not affect the path $J \rightarrow K$.

Clearly, this example can be scaled up by introducing many more sensors, many more wards, many more workstations, etc. To make this possible, the entities would have to be parameterized or indexed, such as *PulseDetect1*, *PulseDetect2*, etc. Evidently, real-life networks will not be able to be shown in the simple graphic format used here. Graphic interfaces, showing high-level representations that can be manipulated by administrators, should be the subject of future research.

6. Network Reconfigurations

In the IoT, the network topology may be continuously transformed or reconfigured with the creation or removal of entities and communications channels. This can occur for many reasons, notably by the intervention of system administrators or by the effect of policies. For example, in some systems, some reconfigurations are determined by policies expressed in terms of time, such as that at certain times, certain entities may change their permissions (i.e., labels) or disappear altogether, while others may be created. The

routing tables must be updated at each such event, but the implementation configuration will remain the one in Figure 1 or Figure 9. It is a useful property of the partial order model that a partial order exists for any network, and so our method can be used to construct new routing tables after each network reconfiguration (note that, on the contrary, if a lattice-structured network is reconfigured, the result may not be lattice-structured). Reconfigurations may affect only some routing tables.

In our partial order model, the reconfigurations that matter are the changes in the domination relation, because these are the only ones that can change the data flows. Reconfigurations that do not change this relation are adding or removing channels that are implied by transitivity. Consequently, we consider three types of reconfigurations: introduction or removal of entities, or label changes. The label of an entity can be specified in one of two ways:

- Explicitly: in this case, the label indicates the intended contents of the entity and, by inference, its *Channel* and *CF* relations
- Implicitly, in this case, the *Channel* or *CF* relations of the entity are given, along with, by inference, its intended contents and label.

For implementation efficiency, it should be considered that different networks will have different update needs. For example, some networks may have very frequent label changes but much less frequent additions or removals. In this case, the algorithms and data structures will have to be optimized for quick label changes, and it may not matter if they perform less well for the other operations. Adding backward links in the labeling tables will help speed up certain searches but will also increase the amount of memory required for the tables. Further, the labeling tables may have to be kept sorted according to some criteria to speed up searches. Such decisions should be left to the designers of specific systems. Standard data structure theory proposes methods that can be used for optimizing the reconfiguration methods, and we leave this to further research.

- *Addition of new entities*: Three types of entities can be introduced: sensors, storage entities, and application entities. In each case, we assume that the new entity comes with a label, see above.
 - Adding a sensor: The sensor's labels contain only the names of the sensors themselves, along with the names of other equivalent sensors, if any. In the implementation configuration, the new sensor must be attached to the appropriate access point. In the labeling tables, a line must be added for the new entity, containing in the *Holds* column the name of the equivalent sensors. Further, the name of the new sensor must be added to the *Holds* lists of all entities that should receive data from it.
 - Adding a storage entity: If it is decided to add a new storage entity to the cloud layer, the change to the implementation configuration is the appearance of this entity attached to the *cloud router*. Concerning the labeling table, this new entity will have to belong to one of the already existing equivalence classes. This one already must have at least one storage entity (otherwise it will be disconnected from the other entities). Then the new entity must be added to the labeling table with the same *Holds* list as the other entities in its equivalence class; it must also be included in the *Holds* lists of all the entities in its equivalence class.
 - Adding an application entity: Two main cases arise, according to whether the new entity belongs to an existing equivalence class or whether instead, it will be in a new equivalence class (in other words, whether it has an existing label or a new one).
 - A. The first case is easily treated. For the implementation configuration, the new entity will be connected to the *App Router*. The new entity will access the same data entities as the other entities in its class. For the labeling table, a new entry must be created for the new entity, its name must be added to the *Holds* lists of all entities in its equivalence class, and the *Holds* list

of the new entity must be the same as the *Holds* lists of these entities. The name of the new entity should be added to the *Holds* lists of all entities that dominate it in the partial order (that should receive data from it).

- B. The second case is the case of the addition of an application entity with a new label, which creates a new equivalence class. In this scenario, we must add at least a corresponding storage entity for this new entity with the same label. For the implementation configuration, the new entity must be connected to the *App Router* and the new storage entity must be connected to the *Cloud Router*. For the labeling tables, new entries must be created for each of the two new entities. The *Holds* lists of these two entities must be identical and must contain the names of all entities from which they should receive data. The names of these two entities must be added to the *Holds* lists of the entities where they should send data.

The following pseudocode summarizes the steps for adding entities to our configurations (Algorithm 1).

Algorithm 1: Entity Addition

Input: Initial implementation configuration

Output: Updated implementation configuration

```

1: Procedure AddEntity(Entity E)
2:   If E is a sensor, then
3:     AddSensorEntity
4:   Else if E is a storage entity (database), then
5:     AddStorageEntity(E)
6:   Else if E is a workstation, then
7:     AddWorkstationEntity(E)
8:   End if
9: End procedure

10: Procedure AddSensorEntity(Entity E)
11:   Entity(E)
12:   AddConnectionToAccessPoint(E)
13:   UpdateRoutingTablesForSensor(E)
14: End procedure

15: Procedure AddStorageEntity(Entity E)
16:   AddEntity(E)
17:   AddConnectionToCloudRouter(E)
18:   UpdateCloudRouterLabelingTable(E)
19:   UpdateAppRouterLabelingTable(E)
20: End procedure

21: Procedure AddWorkstationEntity(Entity E)
22:   If E has an equivalent storage entity, Then
23:     AddEntity(E)
24:     AddConnectionToAppRouter(E)
25:     UpdateAppRouterLabelingTable(E)
26:     UpdateCloudRouterLabelingTableForAuthorizedEntities(E)
27:   Else
28:     AddEntity(E)
29:     AddEquivalentStorageEntity(E)
30:   Repeat AddWorkstationEntity(E) // Recursive call
31:   End if
32: End procedure

```

```

33: Function UpdateCloudRouterLabelingTable(Entity E)
34:     // Add a line in the cloud router's labeling table for the new entity E
35: End function

36: Function UpdateAppRouterLabelingTable(Entit' E)
37:     // Add a line in AppRouter's labeling table for the new entity E
38: End function

39: Function UpdateCloudRouterLabelingTableForAuthorizedEntities(Entity E)
40:     // Add IP address of E into labeling table of cloud router for authorized entities only
41: End function

42: Function AddEntity(Entity E)
43:     // Add entity E to the implementation configuration
44: End function

45: Function AddConnectionToAccessPoint(Entity E)
46:     // Add connection between entity E and the Access Point
47: End function

48: Function AddConnectionToCloudRouter(Entity E)
49:     // Add connection between entity E and the cloud router
50: End function

51: Function AddConnectionToAppRouter(Entity E)
52:     // Add connection between entity E and AppRouter
53: End function

54: Input: Initial implementation configuration
55: Output: Updated implementation configuration

56: For each Entity E in InitialConfiguration do
57:     AddEntity(E)
58: End for

59: Return UpdatedConfiguration

```

-
- *Entity removal and entity failure:* This will change the implementation configuration since the removed entity will not be included in the new implementation configuration. It should be kept in mind that the removal of an entity does not make it necessary to find alternate paths in a network, since labeling tables contain the *IPADs* of all potential receivers of a data item. In fact, the system will keep working properly even if nothing is done in the case of entity removal: simply, data will continue to be sent to a non-existent entity. In this sense, we claim that our system is tolerant of entity failure, an important property of the IoT.
 - Removing a sensor: The sensor must be removed from the implementation configuration. All occurrences of the name of the sensor must be removed from the labeling tables.
 - Removing a storage entity: The fact that every equivalence class of entities must have a storage entity implies that a storage entity can be removed if and only if there remains at least one storage entity in its equivalence class. The name of the storage entity must be removed from the labeling tables; however, these tables should already contain references to other equivalent entities, so nothing else needs to be changed. In practice, the different storage entities in an equivalence class may have different contents, and if so, some contents may have to be copied, but we leave this as an implementation issue.

- Removing an application entity: In our example, this would be removing a *workstation*. In this scenario, we need also to check the equivalence classes. We have two cases:
 - A. If the equivalence class that contains the entity to remove has at least another application entity in it, we only remove the intended entity and we leave the corresponding storage entities for the other application entities. The name of the entity must be removed from the labeling tables.
 - B. Otherwise, we remove the intended entity and all the equivalent storage entities since none of them is required anymore. The names of all such entities must be removed from the labeling tables.

The following pseudocode summarizes the steps for removing entities from our configuration Algorithm 2.

Algorithm 2: Entity Removal

Input: Initial implementation configuration

Output: Updated implementation configuration

```

1: Procedure RemoveEntity(Entity E)
2:   If E is a sensor, then
3:     RemoveSensorEntity(E)
4:   Else
5:     If E is a database (storage entity), then
6:       RemoveStorageEntity(E)
7:     Else If E is a workstation, then
8:       RemoveWorkstationEntity(E)
9:     End if
10:  End if
11: End procedure

12: Procedure RemoveSensorEntity(Entity E)
13:   RemoveEntityFromConfiguration(E)
14:   RemoveEntriesFromLabelingTables(E)
15: End procedure

16: Procedure RemoveStorageEntity(Entity E)
17:   RemoveEntityFromConfiguration(E)
18:   RemoveEntriesFromLabelingTables(E)
19: End procedure

20: Procedure RemoveWorkstationEntity(Entity E)
21:   If EquivalenceClassContainsOtherEntities(E) then
22:     RemoveEntityFromConfiguration(E)
23:     RemoveEntriesFromLabelingTables(E)
24:   Else
25:     RemoveEntityFromConfiguration(E)
26:     RemoveCorrespondingStorageEntity(E)
27:     RemoveEntriesFromLabelingTables(E, CorrespondingStorageEntity)
28:   End if
29: End procedure

30: Function EquivalenceClassContainsOtherEntities(Entity E)
31:   // Check if the equivalence class of E contains at least one other application layer entity
32: End function

33: Function RemoveEntityFromConfiguration(Entity E)
34:   // Remove entity E from the implementation configuration
35: End function

```

```

36: Function RemoveEntriesFromLabelingTables(Entity E, Optional Entity F)
37:     // Remove all entries of entity E from the labeling tables
38:     If F is provided then
39:         // Also remove entries of entity F from the labeling tables
40:     End if
41: End function

42: Input: Initial implementation configuration
43: Output: Updated implementation configuration

44: For each Entity E to remove in ImplementationConfiguration do
45:     RemoveEntity(E)
46: End for

47: Return UpdatedConfiguration

```

- *Label changes*: Changing the label of an entity is equivalent to removing the entity and then adding it with the new label, so it can be done by combining the two procedures. This change has no effect on the implementation configuration: the entity remains in its place. The labeling tables will have to be consistent with the new labels.

It should be stressed that these are not manual operations, and in the above, we have described the algorithms to perform them.

7. Networks with Multiple Data Flows

In the example of Section 5, we have only considered the existence of a single data flow in the network. Usually, however, several separate data flows are present in networks. Each one of these flows will have different security requirements and will need to be controlled separately, hence it will have its partial order. We modify our hospital example to add a downward flow that we call *Diagnostic*, from the *Chief of medicine* towards the patients. For this new flow, the secrecy-integrity requirements are reversed, and labels denoting combinations of patients' diagnostic data are allowed only for certain equivalence classes of entities.

We say that the example of Section 5 deals with *Consultation* data that flow from patients towards the medical staff as we have seen. We add to this *Diagnostic* data that travel in the opposite direction and have their requirements in terms of secrecy, which leads to a different partial order. The network with the representation of the two different flows is shown in Figure 11. In this figure, it is possible to find all flows considered in Section 5, plus the new ones, among others the flow $K \rightarrow D \rightarrow E$ by which *Sally* receives her medical results.

We have two sets of labels, one with the flow identifier *Consultation*, and the other with the flow identifier *Diagnostic*. There are also some new entities: *BobWkstn*, *SamWkstn*, and *SallyWkstn* respectively *L*, *F*, and *E*, which represent the patient applications that will allow them to consult the *Diagnostic* data flow. So some entities will have two labels. For example, the labels of *ChiefMedicWkstn* are as follows: *Consultation* (*SamPress*, *BobPulse*, *SallyPulse*, *Stat1*, *Stat2*) and *Diagnostic* (*SamDiagnos*, *SallyDiagnos*, *BobDiagnos*). This means that *ChiefMedicWkstn* participates in the two flows, and that for each flow, *ChiefMedicWkstn* has access to data on the corresponding labels.

This example shows an application of the concept of *trusted entities* that can access data belonging to different flows but are trusted to deliver the right data to the rightful entities only. One such entity is the *ChiefMedicWkstn*. This entity knows both *Sam's* and *Bob's* data and sends data to both, but it is expected not to send *Sam's* data to *Bob* or vice-versa. The concept of a trusted entity is well established in security theory and is present in the *Bell-La Padula model* [19], where trusted subjects are described as "guaranteed not to consummate a security-breaching information transfer even if it is possible". Trusted entities are very common in security; for example, a bank employer will hold separate in her mind the two

conversations she may be having with her manager and with her client on two different phone lines. Trusted entities can be thought of as split into different parts, one for each flow to which they belong, with controlled internal communication between the parts. Each part will be governed by the label associated with its flow. In the example below, the Chief Medical Workstation and its database (K, K') are trusted entities that are supposed to keep separate the data of the three users, and similarly, the doctor and nurse workstations and databases (C, A, A') are supposed to keep separate the data of *Bob* and *Sam*. This could be shown in greater detail but at the cost of complicating the presentation.

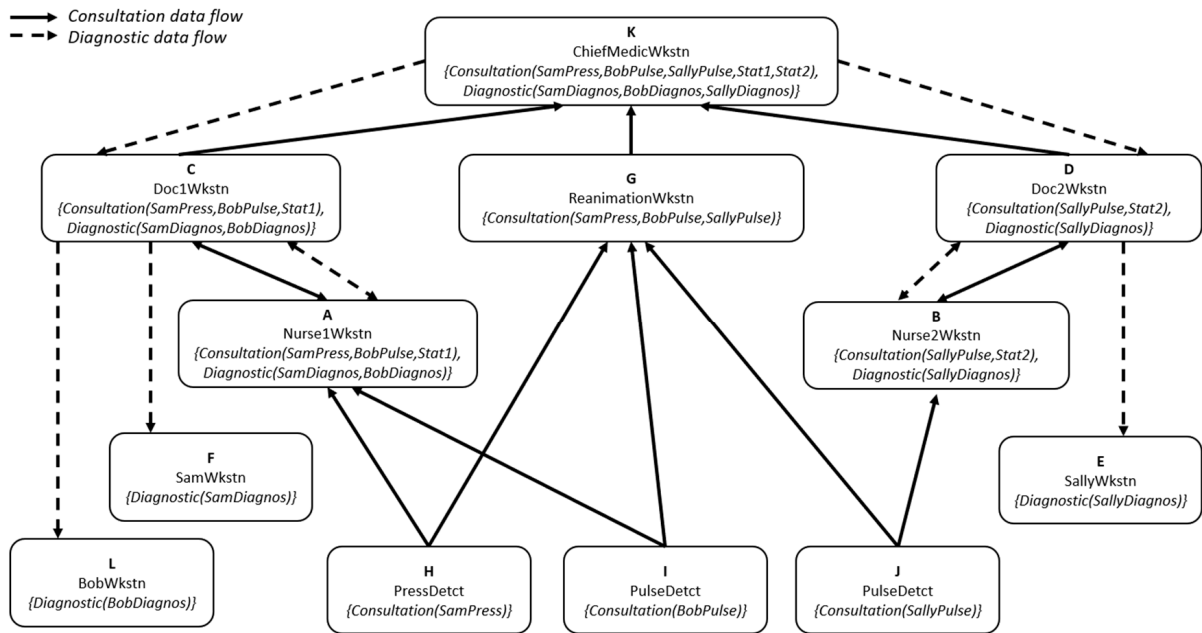


Figure 11. Two-flow network for the hospital example.

In order to implement this model, we need to create a network where all the data are saved in the Cloud. For this purpose, we add storage entities to the newly created entities for the patients. These can be small storage spaces allocated through the patient’s account created during registration on the hospital servers.

Figure 12 represents the resulting network. We have two sets of labels, one set for each flow, respectively, named *Consultation* and *Diagnostic*. For each flow, the labels associated with that flow are used.

The data flow $K \rightarrow D \rightarrow E$ noted in Figure 11 becomes in Figure 11 $K \rightarrow K' \rightarrow B' \rightarrow D \rightarrow E' \rightarrow E$ since B' is a Network layer database shared by D and B .

The main difference with respect to the one-flow example is that the controller will have two forwarding tables, one for each data flow. In the case of *Consultation* data flow, the labeling tables for the two routers will be the same as the one for the one-flow example.

The new implementation configuration is as described earlier; we have two routers that connect the network entities: one to connect the storage entities and one to connect the workstations.

Each router will have a forwarding table for each flow; each incoming packet will contain its flow identifier together with its label; and the controller will use the flow identifier to switch to the appropriate forwarding table for each packet.

The partial order for the new *Diagnostic* flow is shown in Figure 13. We clearly see in this figure that the diagnostics start at entities K and K' , which are the *ChiefMedicWkstn* and its database, and end at the patients and their databases. We also see that the dataflows do not allow sending one patient’s data to other patients. The labeling table is still obtained automatically by the rule of Section 4.2, as shown in Figure 14.

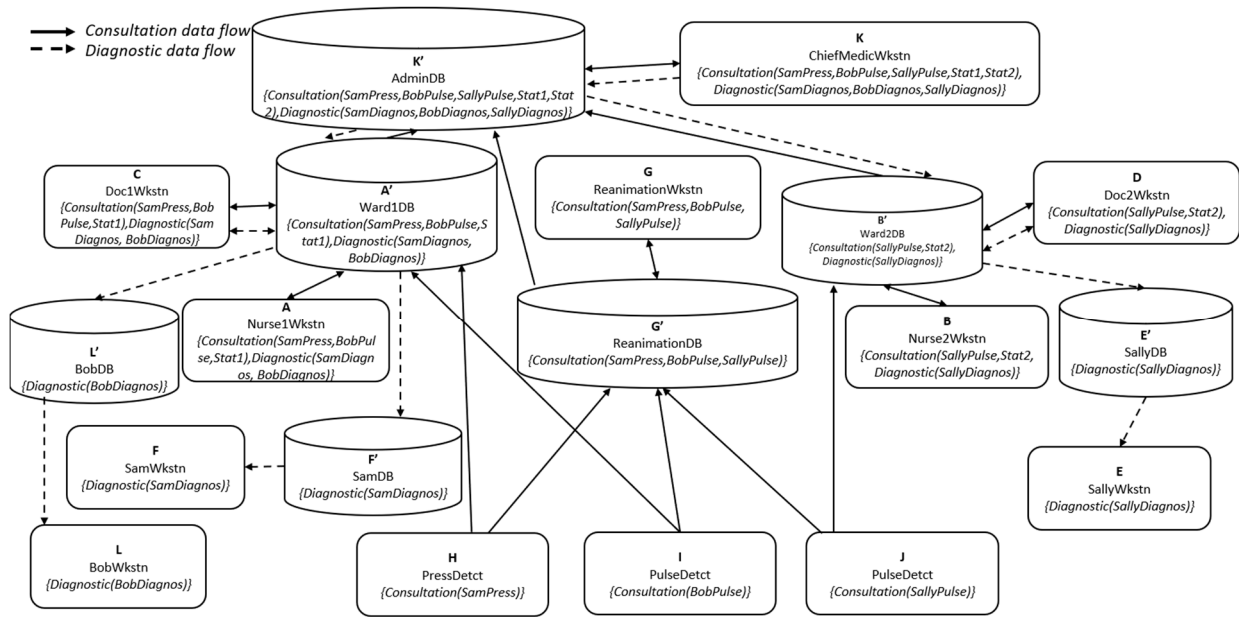


Figure 12. Cloud configuration of Figure 11.

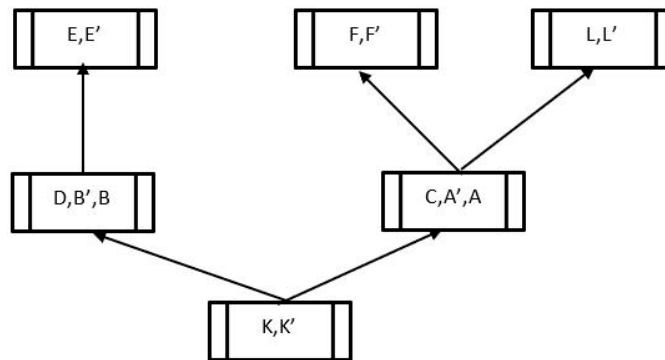


Figure 13. Partial order for the Diagnostic Flow.

Labeling table of the cloud router	
Entity	Holds
K'	K, K'
B'	K, B, D, B', K'
A'	K, A, C, A', K'
E'	K, K', B, B', D, E, E'
F'	F, F', C, A, A', K', K
L'	L, L', C, A, A', K', K
Labeling table of the App router	
Entity	Holds
K	K, K'
C	K, A, C, A', K'
A	K, A, C, A', K'
D	K, B, D, B', K'
B	K, B, D, B', K'
E	K, K', B, B', D, E, E'
F	F, F', C, A, A', K', K
L	L, L', C, A, A', K', K

Figure 14. Labeling table for Figure 9.

The data flow $K \rightarrow K' \rightarrow B' \rightarrow D \rightarrow E' \rightarrow E$, discussed above, can be traced in the labeling table by checking that the name of each entity in the sequence is included in the Holds of the following entity. It can also be checked that unwanted flows are impossible, e.g., Bob's workstation L cannot receive Sally's diagnostics because it does not have E in its Holds. The routing tables, derived from the labeling table, will not enable this forwarding.

8. Simulation and Implementation of the Controller

The SDN implementation of our hospital example has been tested using the Mininet network emulator. Mininet is a network emulator that runs a collection of end hosts, switches, routers, and links on a single Linux kernel. It uses virtualization to make the system look like a complete network, running the same kernel, system, and user code. Mininet hosts behave just like real machines that can run arbitrary programs. The programs can send packets through what appears to be a real Ethernet interface, with a given link speed and delay. Packets get processed by what look like real Ethernet switches or routers, as in our case.

In summary, Mininet's virtual components (hosts, switches, links, and controllers) are created using software rather than hardware, and their overall behavior mimics to the one of discrete hardware elements. It is usually possible to create a Mininet network that represents a hardware network, or a hardware network that represents a Mininet network, and to run the same binary code and applications on either platform. Mininet is particularly adapted to simulate SDN networks, and also is efficient and easy to use.

For the choice of controller, several reasons led us to use the Ryu controller of Asadollahi et al. [32,33]. First, we considered the comparison study documented by Ola et al. [34]. Second, there is the fact that Ryu provides software components with well-defined APIs that make it easy for developers to create new network management and control applications. Third, Ryu is the most suitable controller to use in a Mininet environment since it supports OpenFlow 1.0, 1.2, 1.3, and 1.4. Fourth, because Ryu is Python-based, it is easier for Ryu to develop new network management and control applications in comparison with other controllers. And finally, Asadollahi et al. [32] and Islam and Refat [35] have reported on testing the performance of the Ryu controller in many simulation scenarios and have concluded that the controller is very suitable for prototyping and experimentation for research, experimentation, and demonstrations.

To create our implementation configuration, we have used the Python API to write a configuration Python script. First, we had to create an empty network and add nodes or entities to it. To create this empty network, we manually created a default controller called *Controller c0*. This default controller was replaced later with our Ryu controller.

Figure 15, generated by Mininet, shows our example's topology created in the simulation environment. Note that in the figure, *Router2* refers to the *AppRouter*. and *r1* refers to the *AccessPoint*.

The table of Figure 16 gives the meaning of the hostnames presented in the topology of Figure 15.

The simulations that were done aimed to test the integrity and secrecy requirements; in other words, it was tested that by using our labeling tables and derived routing tables, data flows would only arrive at authorized entities. The case of multiple flows was also tested. Parameters such as the performance of the controller, scalability, etc. have been tested in other SDN-related work already mentioned, see Asadollahi et al. [32], Islam, and Refat [35].

After implementation, the data flow in the network was simulated. We use the *ping* command to see if the results match the security requirements expressed by our partial order. Figures 17 and 18 show examples of the results obtained.

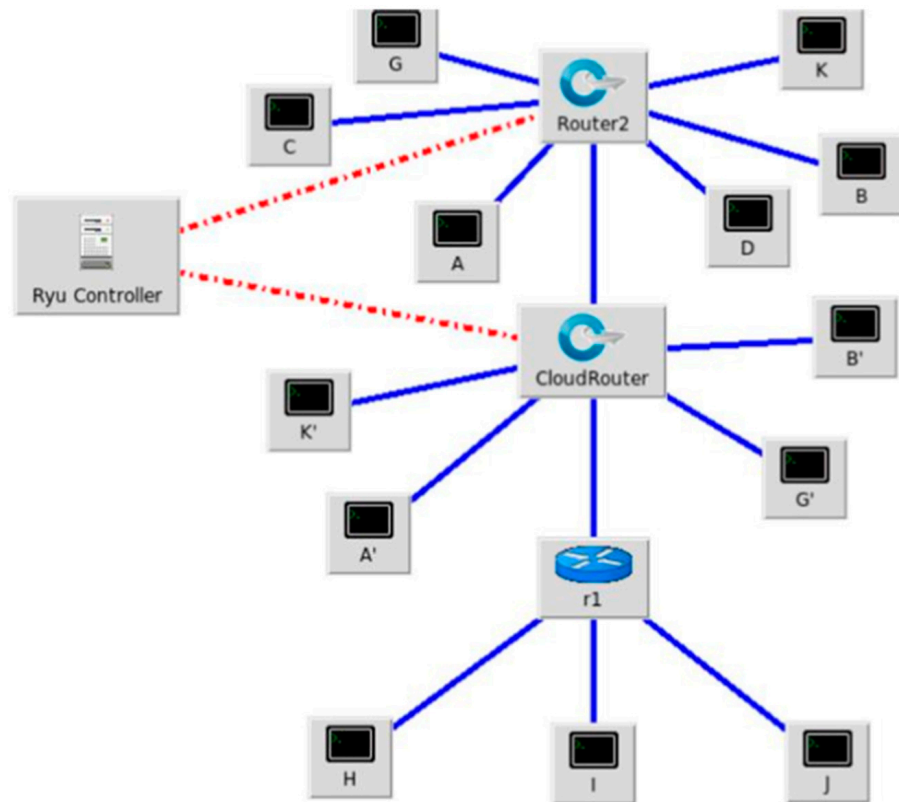


Figure 15. The simulated topology generated by Mininet.

Host ID	Definition
A	Nurse1wkstn
B	Nurse2Wkstn
C	Doc1Wkstn
D	Doc2Wkstn
J	PulseDetect (Sally)
H	PressDetect (Sam)
I	PulseDetect (Bob)
G	ReanimationWkstn
K	ChiefMedicWkstn
A'	Ward1DB
B'	Ward2DB
G'	ReanimationDB
K'	AdminDB

Figure 16. Entities correspondence in the simulation.

```

mininet> J ping B
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=17.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=10.9 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=5.81 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=3.22 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=7.20 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=4.15 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=7.28 ms
^C
--- 10.0.0.2 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6008ms

```

Figure 17. Simulation result of an authorized flow.

```

mininet> J ping A
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
11 packets transmitted, 0 received, 100% packet loss, time 10223ms

```

Figure 18. Simulation result of an unauthorized flow.

In Figure 17, we can see some results for a defined scenario. We have a connection established between entities in the case of authorized data flow J to B , where J represents *PulseDetect(Sally)* and B represents *Nurse2Wkstn*.

Another scenario, presented in Figure 18, shows an unauthorized flow where there is no data flow from J to A , where J represents *PulseDetect (Sally)* and A represents *Nurse1wkstn*.

The multi-flow solution was also tested according to this method.

9. Efficiency and Scalability

For efficiency and scalability evaluation, it is important to note that the overhead imposed by our method will occur only when the routing tables have to be updated; this means at network initialization and whenever events such as administrative decisions or event-driven policies cause network reconfigurations; otherwise, for normal operation, the network will run as any SDN network.

As we have seen, labels can be assigned by administrators or may be calculated from *Channel* or *CF* relations, which may mean capability lists or access control matrices. In [21], Stambouli and Logrippo presented a method for calculating labels based on such information. They showed that a worst-case estimate of label calculation time is for an algorithmic complexity that is cubic on the number of entities in the network (thus excluding exponential complexity). MATLAB simulations yielding estimates were also given in that paper. It was shown in those simulations that for a network of 10,000 entities, the partial order can be found and the labels calculated in about 1.5 min, raising to about 10 min for 20,000 entities, and after that rising rapidly to 1.75 h for 100,000 entities.

These times can improve with more efficient programs and faster computers. Research on efficient graph computations is continuously progressing. Consider also that many IoT networks can be partitioned into partially independent *slices*, as they are called in 5G or *domains*. In practice, many slices or domains can be smaller than the mentioned 10,000 entities, and reconfigurations may affect only some of them.

If, on the other hand, a *CF* relation must be calculated from labels, this also can be done efficiently by setting membership tests, using the relation $CF(x,y) \text{ iff } Lab(x) \subseteq Lab(y)$ [20]. Finally and most importantly, in many practical cases, policies and configurations are set up in such a way that global recalculations are unnecessary since only limited and already planned local changes will occur, with minimal overhead. Due to the many different contexts in which our method can be used, more detailed efficiency considerations, as well

as the adaptation of the method to each context, are left to future research. As a practical example, we may think of a network with 10,000 entities that need to be updated once a day, leading to an overhead of 1.5 min a day.

10. Conclusions

We have shown that it is feasible to use SDN routing in IoT contexts for implementing data security requirements of secrecy, integrity, and conflicts, as we have defined them in Section 3. In the implementation method we propose, data are automatically labeled according to their source, and SDN routing tables will be constructed so that they will be forwarded only to entities meant to receive them; other data will be dropped. We have not explicitly mentioned the property of *availability* in this paper; however, availability follows from the fact that every entity that is allowed to receive some data will receive it by the effect of SDN routing.

Previous research [21,36] has shown that, for any network, it is possible to efficiently calculate labels for the entities in such a way that data flows are determined by the label inclusion relationship. In this paper, it was shown how the labels can be used to construct forwarding tables for SDN controllers that will control data transfers, accordingly, thus ensuring data security. We have proposed a network organization based on cloud concepts with application entities and data entities (or servers, databases). We have demonstrated our method by using a simple ‘hospital’ example that was simulated using standard SDN simulation tools. We envisage future systems where security administrators will be able to configure secure data flows by composing graphs such as the ones presented in our figures at various levels of abstraction or granularity. Reconfigurations can also be done on such graphs, and labels and routing tables can be computed and updated automatically using our method.

Routing is taken care of by external SDN routers that will not burden the IoT devices. The use of our method will not affect the normal execution of SDN, except for the mentioned recalculation of labels and routing tables when reconfigurations are done.

Using our approach, it is possible to significantly enhance technologies like Google B4 WAN, Microsoft Azure, and the GENI testbed cited in Section 3.2 for implementing security requirements.

With respect to the literature reviewed in Section 10, we note the following contributions to our work: Instead of using the lattice model, we use the partial order model, applicable to any network; we represent secrecy and integrity policies with a single mechanism, based on the use of a simple labeling method; we develop a generic SDN framework; we show how different data flows can be defined in a single network; we have methods for network reconfiguration; and finally, we have done an implementation and a simulation of our sample SDN-enabled network.

It should be pointed out, however, that our approach is feasible only for networks where the centralized planning we have envisaged is possible. In more decentralized systems, SDN may not be feasible, and it may be necessary to use established methods based on encryption agreed upon among entities. And even in the case of centralized control, encryption may be necessary to protect from covert channels, since data are transmitted clearly. The combined use of our method with encryption will be the appropriate solution in most cases, depending on the organization of the network, its applications, and the expected security threats and risks. We leave these topics for future research.

Furthermore, we have provided a generic method only, and we have shown how the general data flow could be organized. For the proposed method to become practical, it will require the creation of a suitable administrative model; this is the subject of our ongoing research. IoT networks can be very complex, and their design must take into consideration many different requirements, including security requirements that have not been considered here.

Author Contributions: Conceptualization, A.S. and L.L.; Methodology, A.S. and L.L.; Validation, A.S.; Formal analysis, A.S. and L.L.; Writing—original draft, A.S.; Writing—review & editing, A.S. and L.L.; Supervision, L.L.; Project administration, L.L.; Funding acquisition, L.L. All authors have read and agreed to the published version of the manuscript.

Funding: Discovery grant of the Natural Sciences and Engineering Research Council of Canada—No grant number.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: We thank Ahmed Karmouch for introducing us to the possibilities of SDN for network security and Yvon Andrianirina for technical information on SDN tools. We also thank the anonymous referees for comments that have led to improvements in our explanations. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bishop, M. *Computer security, Art and Science*, 2nd ed.; Pearson Addison-Wesley: Boston, MA, USA, 2019.
2. Alaba, F.A.; Othman, M.; Hashem, I.A.T.; Alotaibi, F. Internet of Things security: A survey. *J. Netw. Comput. Appl.* **2017**, *88*, 10–28. [[CrossRef](#)]
3. Singh, J.; Pasquier, T.; Bacon, J. Securing tags to control information flows within the Internet of Things. In Proceedings of the International Conference on Recent Advances in Internet of Things, RIoT 2015, Singapore, 7–9 April 2015; pp. 1–6.
4. Qiang, G.; Quan, G.; Yu, B.; Yang, L. Research on security issues of the Internet of Things. *Int. J. Future Commun. Netw.* **2013**, *6*, 1–10. [[CrossRef](#)]
5. Landwehr, E. Privacy research directions. *Comm. ACM* **2016**, *59*, 29–31. [[CrossRef](#)]
6. Etalle, S.; Hinrichs, T.L.; Lee, A.J.; Trivellato, D.; Zannone, N. Policy Administration in Tag-Based Authorization. In *Foundations and Practice of Security, Proceedings of the 5th International Symposium, FPS 2012, Montreal, QC, Canada, 25–26 October 2012*; Springer: Berlin/Heidelberg, Germany, 2013; Springer LNCS; pp. 162–179.
7. Hinrichs, T.; Garrison, W., III; Lee, A.; Saunders, S.; Mitchell, J. TBA: A Hybrid of Logic and Extensional Access Control Systems. In *International Workshop on Formal Aspects in Security and Trust*; Springer: Berlin/Heidelberg, Germany, 2011; Volume 7140, pp. 198–213.
8. Sandhu, R. Lattice-based access control models. *IEEE Computer* **1993**, *26*, 9–19. [[CrossRef](#)]
9. Burke, Q.; Mehmeti, F.; George, R.; Ostrowski, K.; Jaeger, T.; La Porta, T.F.; McDaniel, P. Enforcing Multilevel Security Policies in Unstable Networks. *IEEE Trans. Netw. Serv. Manag.* **2022**, *19*, 2349–2365. [[CrossRef](#)]
10. Xie, R.-N.; Li, H.; Shi, G.-Z.; Guo, Y.-C.; Niu, B.; Su, M. Provenance-based data flow control mechanism for Internet of things. *Trans. Emerg. Telecommun. Technol.* **2021**, *32*, e3934.
11. Al-Haj, A.; Aziz, B. Enforcing Multilevel Security Policies in Database-Defined Networks using Row-Level Security. In Proceedings of the 2019 International Conference on Networked Systems (NetSys 2019), Marrakech, Morocco, 18–21 March 2019; pp. 1–6.
12. Denning, D.E. A lattice model of secure information flow. *Commun. ACM* **1976**, *19*, 236–243. [[CrossRef](#)]
13. Wang, A.; Mei, X.; Croft, J.; Caesar, M.; Godfrey, B. Ravel: A database-defined network. In Proceedings of the Symposium on SDN Research, Santa Clara, CA, USA, 14–15 March 2016; pp. 1–7.
14. Fernandes, E.; Paupore, J.; Rahmati, A.; Simionato, D.; Conti, M.; Prakash, A. Flowfence: Practical data protection for emerging IOT application frameworks. In Proceedings of the USENIX Security Symposium, Austin, TX, USA, 10–12 August 2016; pp. 531–548.
15. Celik, B.; Babun, L.; Kumar, A.; Aksu, H.; Tan, G.; McDaniel, P.; Uluagac, A. Sensitive information tracking in commodity IoT. In Proceedings of the 27th {USENIX}. Security Symposium ({USENIX} Security 18, 2018), Baltimore, MD, USA, 15–17 August 2018; pp. 1687–1704.
16. Chi, H.; Zeng, Q.; Du, X.; Luo, L. PFirewall: Semantics-Aware Customizable Data Flow Control for Smart Home Privacy Protection. *arXiv* **2021**, arXiv:2101.10522.
17. Khan, M.A. A survey of security issues for cloud computing. *J. Netw. Comput. Appl.* **2016**, *71*, 11–29. [[CrossRef](#)]
18. Kalkan, K.; Zeadally, S. Securing Internet of Things with Software Defined Networking. *IEEE Commun. Mag.* **2017**, *56*, 186–192. [[CrossRef](#)]
19. Bell, D.E.; La Padula, L. Secure Computer Systems: Unified Exposition and Multics Interpretation. Mitre Corp. Report MTR-2997 Rev. 1, March 1976. Available online: https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_811 (accessed on 23 January 2021).
20. Logrippo, L. Multi-level models for data security in networks and in the Internet of things. *J. Inf. Secur. Appl.* **2021**, *58*, 102778. [[CrossRef](#)]
21. Stambouli, A.; Logrippo, L. Data flow analysis from capability lists, with application to RBAC. *Inf. Process. Lett.* **2019**, *141*, 30–40. [[CrossRef](#)]

22. Huang, D.; Chowdhary, A.; Pisharody, S. *Software-Defined Networking and Security. From Theory to Practice*; CRC Press: Boca Raton, FL, USA, 2019.
23. Sushant, J.; Alok, K.; Subhasree, M.; Bogdan, A.; Mat, F.; Paul, S.; Jennifer, Z. B4: Experience with a Globally Deployed Software Defined WAN. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 3–14.
24. Minlan, Y.; Jennifer, R.; Michael, F.; Jia, W. Software Defined Networking for Cloud Developers. In Proceedings of the USENIX Conference on Networked Systems Design and Implementation, San Jose, CA, USA, 28–30 April 2010.
25. Mark, B.; Jeffrey, C.; Lawrence, L.; Akihiro, N.; Max, O.; Charles, S.; Josph, Y. GENI: A Federated Testbed for Innovative Network Experiments. *Comput. Netw.* **2014**, *61*, 5–23.
26. Stergiou, C.; Psannis, K.E.; Kim, B.G.; Gupta, B. Secure Integration of Internet-of-Things and Cloud Computing. *Future Generation Comput. Syst.* **2013**, *78*, 964–975. [[CrossRef](#)]
27. Atlam, H.F.; Wills, G.B. Intersections between IoT and distributed ledger. *Adv. Comput.* **2019**, *115*, 73–113.
28. Roy, W.; Bill, S.; Scott, J. Enabling the Internet of Things. *Computer* **2014**, *48*, 28–35.
29. El-Garoui, L.; Pierre, S.; Chamberland, S. A New SDN-Based Routing Protocol for Improving Delay in Smart City Environments. *Smart Cities* **2020**, *3*, 1004–1021. [[CrossRef](#)]
30. de Assunção, M.D.; Carpa, R.; Lefèvre, L.; Glück, O.; Boryło, P.; Lasoń, A.; Szymański, A.; Rzepka, M. Designing and building SDN testbeds for energy-aware traffic engineering services. *Photonic Netw. Commun.* **2017**, *34*, 396–410. [[CrossRef](#)]
31. Available online: <https://www.lifewire.com/how-many-devices-can-share-a-wifi-network-818298> (accessed on 10 November 2023).
32. Asadollahi, S.; Goswami, B.; Sameer, M. Ryu controller’s scalability experiment on software defined networks. In Proceedings of the 2018 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC 2018), Bangalore, India, 1–2 February 2018; pp. 1–5.
33. RYU Project Team. *RYU SDN Framework-RYU Project Team, 2014*. Available online: <https://osrg.github.io/ryu-book/en/html/preface.html> (accessed on 15 June 2021).
34. Ola, S.; Imad, E.; Ayman, K.; Ali, C. SDN controllers: A comparative study. In Proceedings of the 18th Mediterranean Electrotechnical Conference (MELECON 2016), Lemesos, Cyprus, 18–20 April 2016; pp. 1–6.
35. Islam, M.T.; Islam, N.; Refat, M.A. Node to Node Performance Evaluation through RYU SDN Controller. *Wireless Pers. Commun.* **2020**, *112*, 555–570. [[CrossRef](#)]
36. Logrippo, L.; Stambouli, A. Configuring data flows in the Internet of Things for security and privacy requirements. In Proceedings of the 11th International Symposium on Foundations and Practice of Security, Montreal, QC, Canada, 13–15 November 2018; Springer LNCS 11358. pp. 115–130.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.