Deriving Event Data Sharing in IoT Systems using Formal Modelling and Analysis

Paul Fremantle[a], Benjamin Aziz[b]

[a]*WSO2, London, U.K.*

[b]*School of Computing, University of Portsmouth, Portsmouth, U.K.*

Abstract

The increasing presence and utilisation of IoT systems raises many fundamental security and privacy issues that require robust approaches in understanding the behaviour of IoT systems and tackling those issues. In previous works, we demonstrated how some of the security and privacy questions in IoT systems could be answered by means of using federated identity management and authorisation frameworks, such as OAuth, intelligent gateways and personal cloud systems. In this paper, we take these works into a more fundamental level by formally modelling and analysing the OAuthing personal cloud-based IoT system. We demonstrate that this exercise reveals how data is shared across the system, and therefore how security and privacy guarantees can be established at a fundamental level.

Keywords: CSP, Federated Systems, Formal Modeling, IoT

1. Introduction

The number of Internet of Things (IoT) devices has grown rapidly in recent years, with some recent estimates suggesting that there were 12.5 billion Internet-attached devices in 2010 and a prediction of 50 billion devices by 2020 [1]. This brings with it multiple security challenges:

- The sheer scale and number of predicted devices will create new challenges and require new approaches to security.
- These devices are becoming more central to peoples' lives, including in safety critical systems such as cars. Therefore the security of IoT devices is becoming more important.
- Many IoT devices collect information that may be *fingerprinted* [2] and therefore become personally identifiable. This can lead to privacy concerns.
- Because devices can affect the physical world, there are attacks that can cause physical harm to people and systems.
- These devices, due to size and power limitations, may not support the same level of security that is expected from more traditional Internet-connected systems.

Because of the pervasive nature of IoT, privacy and security are important areas for research. In 2016, more than 100,000 IoT devices were conjoined into a hostile botnet named Mirai that attacked the DNS servers of the east coast of the US [3]. The total attack bandwidth of this system was measured at more than 600Gbps. In fact, the number of devices attacked was a small number compared to the potential: previous research [4] has identified several million devices that are available for attack.

Email addresses: paul@wso2.com (Paul Fremantle), benjamin.aziz@port.ac.uk (Benjamin Aziz)

Therefore there is a strong motivation to find approaches to improve and enhance the security and privacy of IoT systems. In [5, 6, 7], the authors demonstrated how IoT security and privacy can be improved through the use of token-based federated authentication and authorisation approaches (in [5] and [6]), and through personal cloud middleware systems (in [7]). In this paper, we extend the work in [7], which presented a personal middleware system called OAuthing, by building a formal model of the system that aims at achieving two goals. The first is to understand end-to-end properties of the system that disprove unwanted behaviour. For example, to show that certain communications do not occur in order to preserve security and privacy. And the second is to prove that, as a result, data can only be shared in a safe manner based on users' consent.

Since the system is fundamentally a distributed federation of communicating processes, we choose to build its model using the Communicating Sequential Process (CSP) formal language [8]. CSP offers a clear and unambiguous set-theoretic approach for describing such systems as processes that interact using message-passing. For more in-depth review of CSP, we refer the reader to sources such as [9, 10]. CSP is supported by means of a powerful refinement checking tool, called Failures-Divergences Refinement (FDR) [11]. The FDR tool uses trace refinement modelling to validate that a defined process behaves as a specification. This is accomplished by defining two different processes, and showing that the finite traces of one process are a subset of the finite traces of the other. While this does not handle infinite traces, in many cases the finite model checking is sufficient. In this way it is possible to validate that the model meets specifications.

The rest of the paper is structured as follows. In Section 2, we give a brief overview of literature works related to this paper. In Section 3, we discuss the OAuthing model informally using the UML notation. In Section 4, we give a brief overview of the CSP algebra. In Section 5, we present the formal model of the various entities in the OAuthing system using CSP. In Section 6, we analyse the composition of these entities together leading to an event-sharing analysis. In Section 7, we discuss two sets of properties of the system that can be derived from this analysis; end-to-end and data-sharing properties. Finally, we conclude the paper in Section 8 giving directions for future work.

2. Related Work

As we mentioned earlier, the work presented here models the OAuthing system developed and presented in [7]. In [7], we demonstrated, at a technical level with the OAuthing system, how IoT users could share data without linking the specific user to a given IoT device, using a system based on the OAuth2 [12] standard. One of the major advantages of systems such as OAuthing is that they promote interoperability by solving issues related to identity management when several IoT components are connected to one another. Such issues have already been highlighted and addressed in several works in literature, e.g. [13, 14, 15], all of which can provide relevant inputs to OAuth-based systems to control access according to the type of context the system operates within. Some of these works use formal approaches in defining and verifying such interoperability, e.g. in [13], graph theory was used to formalise structural controllablility in super node architectures in distributed smart grid systems leading to nodes being able to interoperate in the system with one another. In such theory, security is modelled based on the access control framework defined by the IEC-62351-8 standard [16].

The concept of a *gateway* as one of the important strategies that can facilitate interoperability, was proposed in [14], in the context of Cyber Physical Systems (CPSs). Dependability properties (including reliability and robustness), for which formal specification and analysis techniques allow their establishment, have been highlighted as one of the control and automation requirements for such gateway-based systems. Furthermore, in [15], a reference architecture for Industry 4.0 systems was proposed that establishes interoperability in a

secure manner using Policy Decision Points (PDPs), which allow various access control measures, e.g. RBAC [17, 18] to be deployed when components in a critical system need to coordinate and constrain access to their resources.

Formal analysis and verification have in recent years been applied to IoT and security protocol standards using various approaches. For example, OAuth2 [12] has been formally analysed using the ProVerif static analysis tool [19], in which specific threats were identified. Another example is that of Pai et al. [20], who utilised the Alloy Framework [21] to analyse the security constraints of OAuth2. In [22], a threat model for OAuth2 was defined, which despite showing that threats do exist in OAuth2, it is nonetheless is a reasonably robust protocol. More recently, a formal analysis of the MQTT protocol [23] was conducted in [24] in relation to the quality of service semantics of the protocol, where the authors demonstrated that certain ambiguities existed that could compromise the security of the protocol. Apart from the above attempts, in general, applying formal modelling and analysis techniques to IoT systems and protocols has been limited over the years, despite the critical nature of some of such systems. An early attempt in [25] was made to model formally publish/subscribe protocols to capture their essential properties such as minimality and completeness, however, without any attempt to incorporate hostile environments within which these protocols may run. In [26], a formal model of publish/subscribe protocols, within the domain of Grid computing, was defined using Petri-Nets. The model offers a mechanism for the composition of existing publish/subscribe protocols hence offering a viable approach for the validation of such protocols. In [27], an early attempt was made to discuss the security properties and requirements desirable in a publish/subscribe protocol, in particular within the domain of Internet-based peer-to-peer systems, where such protocols became popular early on. Our work follows in the path of such works, although in our case, the system modelled and verified is of a complex nature, whereas in most of the above cases, only toy examples are considered.

Similar to our approach, several other works, e.g. [28, 29, 30, 31, 32], have adopted model checking as an automated technique for verifying properties of publish-subscribe-based IoT systems related to reliability and correctness as well as various levels of efficiency. The authors in [28, 31] define a general framework for the model-checking of publish-subscribe systems, without focusing on specific systems or properties. The approach of [29] adopts a specific system, *thinkteam*, as the target for their analysis, and in [30], the authors propose a dedicated model checking technique to verify properties of publish/subscribe-based Message Oriented Middleware (MOM) systems. However, unlike our approach, none of these works seeks to establish properties based on the flow of data accross components of the systems analysed, as we propose to do here (Section 7.2). Another related work is that of [32], where probabilistic model checking is used to capture uncertainties inherent in publish-subscribe systems, using a stochastic model. On the same note, probabilistic model checking has also been used in [33] to analyse quality of predictions in service-oriented architectures. Such probabilistic models are more expressive that the standard approach we used in our paper, however, currently we do not model probabilistic or stochastic aspects of the OAuthing system.

Within the domain of sensor network protocols, there is more focus of effort on the formal analysis and verification of IoT protocols. For example, in [34], the authors apply model checking techniques in the verification of a medium access control protocol called LMAC. Similarly, in [35] propose a formal model of flooding and gossiping protocols for analysing their performance probabilisitic properties. More recently, the authors in [36] proposed a formal model and analysis of clock-synchronised protocols in sensor networks based on timed automata. Such works remain at the level of what we call *device*-based systems (Section 5.1), whereas in our case, we also incorporate higher level concepts, such as users.

3. Background: The OAuthing System

We describe in this section, informally, the various concepts and components underlining the model of the OAuthing system using the UML notation [37]. We start by describing what we

mean by a *device*. The concept of a device is restricted to systems that can directly contact the Internet. For example, the prototype uses a WiFi-connected chip, which allows direct connection to Internet systems. This excludes systems that connect via Bluetooth or other non-IP networking. It is possible to think of such Bluetooth connected systems as sensors or actuators connected wirelessly to a "device" (such as a mobile phone), that does participate in this model. For example, a connected car might be seen as a "device" in the terms of this model, and some sensors or actuators might be wirelessly connected over various non-IP protocols to the central processing unit of the car. In addition, this model assumes that devices have at least intermittent connections to the Internet, whereby tokens can be refreshed.

Figure 1 shows the current situation for many IoT systems [7], where the device talks to a single service that manages identity, stores data, provides a user Web interface, etc.

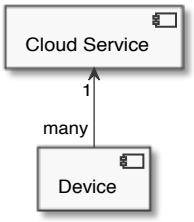

Figure 1: Existing Model of IoT Systems [7]

By comparison, the federated model of OAuthing proposed in [7] and shown in Figure 2, allows different federated parties to provide different services that work together.

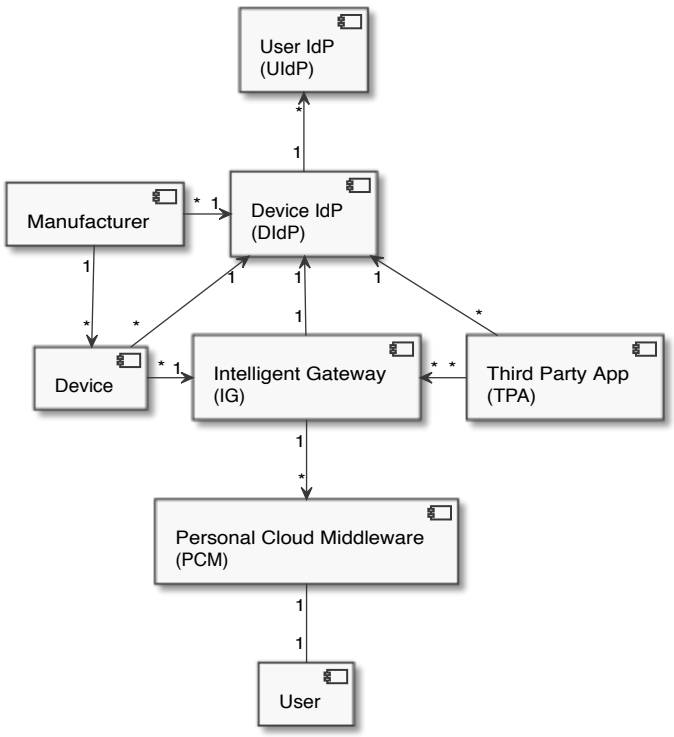

Figure 2: Proposed Model: OAuthing [7]

The participants of the OAuthing model are:

- **The User Identity Provider (UIdP)**: this is an existing login system where Users present their credentials (e.g. Google, Facebook, Github, Twitter or any other OpenID Connect (OIDC) [38] login).
- **The User**: A User may own one or more Devices. A User must have at least one identity with a UIdP.
- **The Device Identity Provider (DIdP)**: this is an *Identity Broker* that first authenticates a User with a UIdP using existing federated identity protocols including OAuth2, OIDC or SAML2. Once the identity is validated, it then creates a secure random anonymous identity which is used in all further processing. This anonymous identity is not shared except with the Intelligent Gateway. Devices and Cloud Services are issued with random tokens that give permission to perform certain actions but do not identify users in any way. Currently, each instance of OAuthing has a single DIdP.
- **Personal Cloud Middleware (PCM)**: this is an isolated broker that shares data between devices and Third Party Applications (TPAs) on behalf of the user. The PCM talks to the Devices and the TPAs. Within the remit of a single OAuthing instance there is one PCM per user. A cloud environment is used to dynamically launch PCM instances on behalf of users as needed.
- **Intelligent Gateway (IG)**: The IG interfaces with the DIdP to validate identities and access authorisation policies and to the cloud infrastructure to instantiate new PCMs. Devices and CSs connect to the IG, and it routes requests to each user's PCM.
- **Third Party Application (TPA)**: A device is an IoT device if and only if it shares or receives data and commands with an Internet service. Users control which TPAs can access their sensor data or control their actuators by explicitly consenting to authorise a TPA. Any third party can provide a TPA. If no TPA is authorised by the user then a Device's data is neither shared nor stored.
- **The Device**: The device consists of one or more sensors and actuators together with a controller. The device is issued with a Client ID at manufacturing time. Once the device is registered with a user, it stores a token that identifies the user, the Client ID and the scopes of access that the user has authorised.
- **The Manufacturer**: The Manufacturer is the logical organisation that creates and markets the Device, irrespective of whether they actually outsource any part of the physical manufacturing to a third party. In this model, the Manufacturer configures each device with a single DIdP.

Figure 3 shows the UML sequence diagram of a runtime interaction between a device and a third-party application [7]. This model utilises the OAuth2 model as a basis for the identity and ownership of devices. One concern with IoT is that hardware devices can be compromised and secrets read from them. It is therefore important that each device has its own credentials. Each device is to be a unique OAuth2 Client, and the system uses the OAuth2 Client ID as a secure device ID that is only ever shared with the DIdP. Ownership of a device is defined by the user authorising the issuance a security token to the device giving it permission to act on the user's behalf.

3.1. Lifecycle

The UML lifecycle diagram corresponding to the OAuthing model is shown in Figure 4. Once the device is initially flashed it is connected to a manufacturing server. The manufacturer then uses the DCR API into the DIdP to request a Client ID and Secret. These are configured into the device by the manufacturing server. At the same time, the DIdP returns a unique User Registration URI (URU), that is printed onto the device (usually as a Quick Response (QR) code) by the manufacturer.

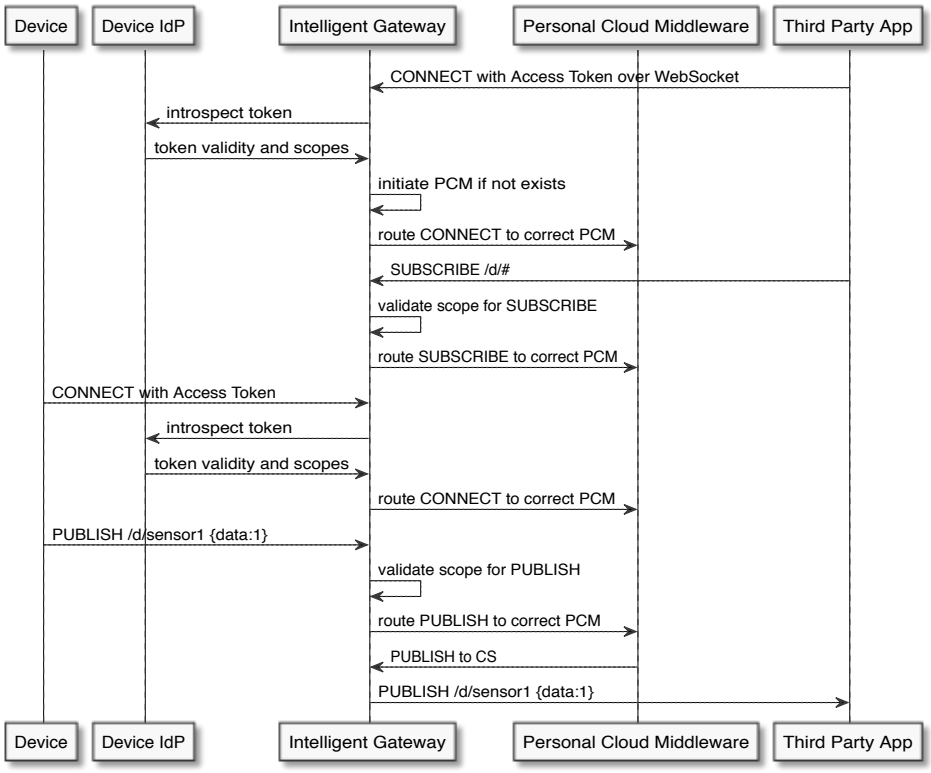

Figure 3: Device Publishing Data to App [7]

When the user buys the device, they scan the QR code or otherwise access the URU. This directs the user to the DIdP which presents a choice of UIdPs to the user. Once the user is authorised with their existing UIdP, they are asked in turn to authorise the device. The resulting OAuth2 refresh token is then stored on the device, and represents the logical ownership of the device. If at some future point the user sells the device they can revoke the OAuth2 token - either by resetting the device back to its initial state or by using a web interface at the DIdP.

3.2. Personal Cloud Middleware

A key part of the model is the concept of a personal hub: where each user's data is routed to its' own hub, protecting the data from multi-tenant attacks. Each hub is run in its own virtualised Cloud environment. When a request comes in from a device or CS, the pseudonym associated with the bearer token is used to route the request to an instance that is specific to that user. If there is no cloud server available, the routing system makes a call to the cloud management system to instantiate a new PCM "on-demand", and then waits until the instance is running before routing the request to the PCM. In the model the PCM supports routing, distribution of data and commands, as well as summarisation and filtering of data. These capabilities have an important role in protecting users privacy: firstly, the runtime does not inherently share data such as IP addresses or MAC addresses that can be used to identify devices or users. Secondly, by filtering or summarising data, the PCM can avoid many fingerprinting attacks on devices [2]. The PCM can also provide protocol mapping and device shadow capabilities, meaning that it is simpler for TPAs to connect to devices.

3.3. Scopes

The DIdP implements consent-based authorisation policies called *scopes*. The concept of scopes comes from the OAuth2 specification but also features in other frameworks, e.g. the

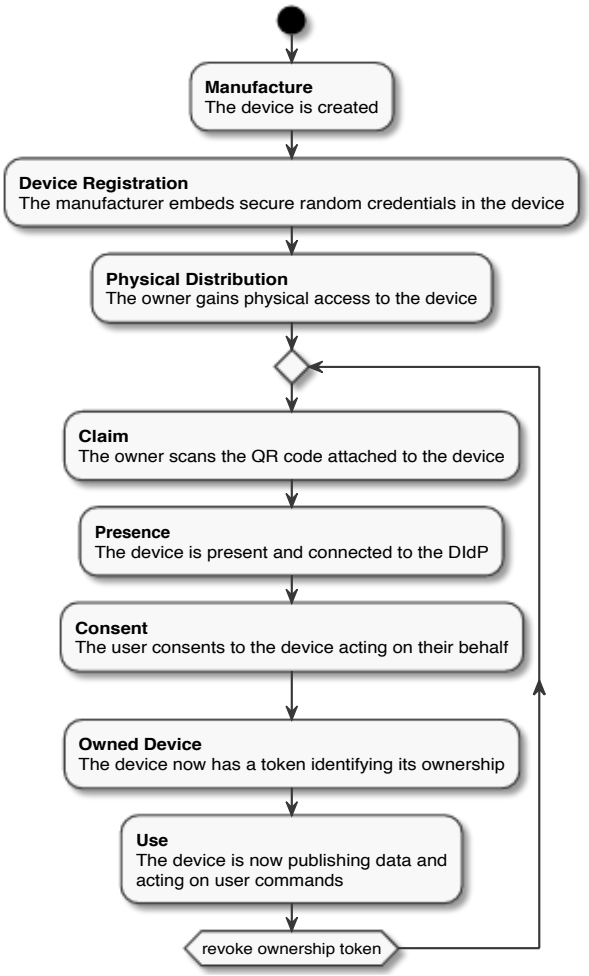

Figure 4: Lifecycle of a Device

IEC/TS 62351-8 [16] role-based access control framework for power systems management. Each scope controls access to a set of APIs. These APIs may be implemented in multiple protocols. Users may consent to a third-party to have access to a specific scope, which is captured in a token. One of the outcomes of defining scopes as part of this model is that there is a clean mapping between the different roles in the system and the scopes which each role requires access to, which is shown in Table 1.

Role	Scopes	Description of roles and scopes
UIdP	N/A	This IdP is the primary source of identity to the Device IdP and does not have any OAuth2 scope permissions
DIdP	openid ***(or UIdP Specific)***	The Device IdP is the "source" of scopes to the other roles. It requires access to the third-party IdPs, which may define their own scopes.
Manufacturer	dcr	Dynamic Client Registration (DCR): allows caller to create new ClientIDs using the DCR API
Intelligent Gateway	intro	Introspection: allows the IG to ask the DIdP for the pseudonym and scopes for a given Bearer Token
TPA	Rd, Pc	Read/Subscribe to Data (Rd) and Publish Commands (Rc) The TPA may be allowed one or other or both
Device	Pd, Rc	Publish Data (Pd). Read/Subscribe to Commands (Rc).

Table 1: Mapping of Roles to Scopes

4. A Brief Overview of CSP

We start by first giving a brief overview of CSP. CSP is an algebraic approach for reasoning about components that communicate via messages. In CSP, components are called *processes*. CSP allows reasoning about individual processes and also communicating groups of processes. Each process communicates using *events*, which may pass messages.

A process P accepts an event e, and then ($\rightarrow$) stops (accepts no further events):

$$P = e \rightarrow STOP$$

$STOP$ is a well-defined process that cannot accept any further events. $STOP$ effectively captures deadlock, so to distinguish *successful* termination, the event ✓, (pronounced "success") identifies successful termination. The process $SKIP$ is defined as a process that does nothing but terminate successfully.

The process Q accepts event e and then continues as Q, showing recursion. In effect, Q can accept any number of events e:

$$Q = e \rightarrow Q$$

A process can be parameterised, leading to a set of processes:

$$Q(d) = e.d \rightarrow Q(d)$$

In this process, $e.d$ indicates that event e transfers data d.

In addition, CSP supports *pattern matching*, which is a common approach in functional programming languages. For example, the following three lines of CSP syntax define the function $f(x)$ for all values of x:

$$f(0) = True$$
$$f(1) = False$$
$$f(\_) = Error$$

An equivalent definition of the same function as a bijection would be:

$$f = \left\{ (0, True), (1, False), (x, Error) \;\middle|\; \forall\, x \notin \{0, 1\} \right\}$$

Pattern matching is also applicable to processes:

$$P(0) = e.A \rightarrow P(1)$$
$$P(1) = e.B \rightarrow P(0)$$
$$P(\_) = e.C \rightarrow P(0)$$

This defines a set of interlinked processes. An event can have associated data. For example, we said that $e.d$ was defined as an event e parameterised with data d. In addition to parameterised events, it is also possible to indicate the sending or receiving of data with events; sending data is written $e!d$, whereas receiving data as $e?d$. $e\$d$ indicates that d may be any arbitrary non-deterministic choice of data from the range of allowed values of d. This range can be restricted; if D is a set, then $e.d : D$ allows $e.d$ where $d \in D$. The same is applicable with $e\$d : D$ and $e!d : D$.

If P, Q are processes, $P \,\Box\, Q$ is the choice between P and Q where the environment chooses which process continues based on the initial event received. This is called *external choice*. For example, $(a \rightarrow P) \,\Box\, (b \rightarrow Q)$ will behave as P after an event a, or Q after an event b. Hence, the "incoming event" chooses the path that the process takes.

CSP also supports general external choice across a set of processes. Therefore:

$$\Box_{x \in S} P_x$$

is equivalent to:

$$P_a \,\Box\, P_b \,\Box\, \ldots \,\Box\, P_m$$

where $a, b, \ldots, m \in S$.

$P \,|||\, Q$ is the process where P and Q *interleave*: that is, they operate independently with no synchronisation of events. On the other hand, $P \parallel Q$ is the process formed by running P and Q in parallel and *synchronising* on all events. In other words, they must both accept the same event at the same time.

The $\parallel$-step law captures this most clearly:

$$(?x : A \rightarrow P) \parallel (?y : B \rightarrow Q) = ?z : A \cap B \rightarrow (P \parallel Q)$$

One can say that P and Q only synchronise on events in set E using the following construct:

$$P \underset{E}{\parallel} Q$$

In many cases, there is the need to define the combination of two processes that each offer certain events, whilst synchronising on yet others. Therefore, the construct:

$$P \; {}_X\|_Y \; Q$$

indicates that P handles events in X except those in Y (notated $X \setminus Y$), Q handles events in $Y \setminus X$ and the processes synchronise on events in $X \cap Y$.

In some cases, one needs to rename events:

$$Q = P[\![^{to}/_{from}]\!]$$

to indicate that the process Q acts like P with event $from$ renamed to event to. CSP also allows the *hiding* of events. For example:

$$(P \underset{E}{\|} Q) \setminus E$$

is the process where P and Q synchronise on events in set E, however, only events that are not in E are visible outside the process. The opposite of hiding is *projection*:

$$P \upharpoonright E$$

which represents process P but where only events in E are visible.

Hiding events is the most obvious cause of *non-determinism*. For example, if there is a process:

$$(a \rightarrow b \rightarrow P \; \Box \; c \rightarrow d \rightarrow Q) \setminus \{a, c\}$$

then the external observer is not aware of whether internal (i.e., hidden) events are taking the process down the a path or the c path. This gives rise to the concept of *internal choice* ($R \sqcap S$), where there is non-deterministical choice between processes R and S. Therefore:

$$(a \rightarrow b \rightarrow P \; \Box \; c \rightarrow d \rightarrow Q) \setminus \{a, c\} = b \rightarrow P \sqcap d \rightarrow Q$$

The concept of *linked parallel* processes encapsulates renaming, hiding and parallel processes into a single concise definition:

$$R = P[c \leftrightarrow d, e \leftrightarrow f]Q$$

indicating that the process R behaves like P interleaved with Q, except that event c from P is renamed as d for Q and vice-versa. Similarly, P sees event f from Q as e and vice-versa. The overall process synchronises on the events (c/d, e/f), which are hidden in R.

Effectively, if f is a fresh unused name:

$$P[c \leftrightarrow d]Q = (P[\![^{f}/_{c}]\!] \underset{f}{\|} Q[\![^{f}/_{d}]\!]) \setminus \{f\}$$

Another useful notation is:

$$R = P \, \Theta_E \, Q$$

which indicates that R behaves like P until an *exception event* from the set of events E occurs, after which it behaves like Q.

CSP defines the concept of *refinement* ($\sqsubseteq$), based on *traces*. Traces are possible patterns of visible events that a process will accept. One can say that P is *trace-refined* by Q (written $P \sqsubseteq_{\mathrm{T}} Q$) meaning that every finite trace of Q is also a finite trace of P:

$$P \sqsubseteq_{\mathrm{T}} Q \Leftrightarrow traces(P) \supseteq traces(Q)$$

This model offers an operational semantics for CSP based on the concept of *Labelled Transition Systems* [39]. The CSP tool FDR [11] allows one to evaluate these trace refinements. The *trace* refinement model does not however, fully show that the implementation process properly implements the specification. To say that all the traces of the implementation are also traces of the specification is not enough. One also needs to show that the implementation has the same failures as the specification (i.e., that it refuses the same events that the specification does). The failures of a process P are formally defined as the set of pairs (s, X), such that the process can follow the sequence of events s (written P/s) and then refuse event X:

$$failures(P) = \{(s, X) \mid s \in traces(P) \wedge X \in refusals(P/s)\}$$

$$P \sqsubseteq_{\mathrm{F}} Q \Leftrightarrow failures(P) \supseteq failures(Q) \wedge traces(P) \supseteq traces(Q)$$

There is one more assertion one can make. A process can *diverge* if it follows a finite trace and then ends up in a state where it can perform an infinite number of *internal events*. As well as the visible events of a process, it may have internal events that are hidden from the external world. In CSP, these events all have the same name τ since they are indistinguishable from one another. A process P is *failures-divergence refined* by Q ($P \sqsubseteq_{\mathrm{FD}} Q$) if the failures and divergences of Q are also failures and divergences of P.

Refinement can be used in two slightly different ways. Firstly, one can define a higher-level model that is refined into a more detailed model. This allows one to show an abstraction away from the details and ensure that the system meets that abstraction. For example, this work defines a device abstractly and then shows that the more complex device lifecycle is a refinement of the simpler concept. Secondly, one can specifically define behaviours that either the model implements or does not implement. The FDR trace refinement analysis can prove that the model either refines or does not refine these behaviour specifications.

5. The OAuthing Formal Model in CSP

Each of the different participants — Device, Manufacturer, Device Identity Provider, Gateway, Personal Cloud Middleware, User, and Application — are modeled in CSP [8]. The whole system is modeled as a composition of these components. The model does not address discovery of the DIdP, UIdP or IG. These could be addressed as further work. Similarly, there is no modeling of change of ownership of a device, which could also be addressed in further work. Moreover, we assume that all distributed communications channels are encrypted, except the PCM-to-PCM communications where the model treats a PCM as CSP processes that communicate internally. Since this PCM-to-PCM communication is internal, it is not encrypted.

5.1. Devices

An IoT *device* is a system that contains either *sensors* or *actuators* or both and supports connections to the Internet either directly or via some intermediary. A *sensor* is a system that can emit an event containing *data* about the world. A sensor can be defined as a process that senses the world and then emits sensor data events. Once it has done this, it can once again sense.

Definition 5.1. Sensor

$SENSOR = sense\$d \rightarrow sd!d \rightarrow SENSOR$

$sense\$d$ indicates that this process may *internally choose*, non-deterministically, among all values of data. The notation $sd!d$ indicates that the event sd sends the value d. Non-deterministic choice correctly models a sensor since real-world events that prompt the sensor data are hidden from the model and so appear "internally" within the sensor.

Similarly, an *actuator* is defined as a system that receives a *command* and then acts on it.

Definition 5.2. Actuator

$ACTUATOR = rc?c \rightarrow act!c \rightarrow ACTUATOR$

The notation $rc?c$ indicates that this event is receiving the value c. The sensor and actuator have no need to synchronise and can operate without recourse to each other. Therefore they are *interleaved*. Next, we define a device as follows.

Definition 5.3. Device

$DEVICE = SENSOR \,|||\, ACTUATOR$

From a set-theoretic point of view, there can be multiple sensors and actuators per device. The CSP view of the device only defines that it can produce data or consume commands. Each sensor or actuator belongs to exactly one device. This is modeled as a function $device$ that maps sensors and actuators to the device they belong to:

$\forall\, s \in Sensors, \exists\, d \in Devices : device(s) = d$

$\forall\, a \in Actuators, \exists\, d \in Devices : device(a) = d$

The function $device$ is surjective:

$\forall\, d \in Devices, \exists\, x \in Sensors \cup Actuators : device(x) = d$

In other words, a device must have at least one sensor or actuator, as shown in Figure 5.

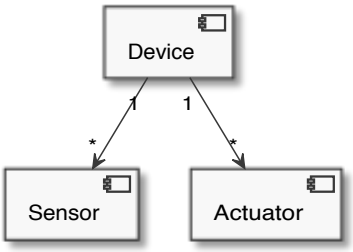

Figure 5: UML Model of a Device

The model is built up in stages. This creates a natural lifecycle for a device (as depicted earlier also in Figure 4). At the high level, the device lifecycle is:

$claim \rightarrow consent \rightarrow connect \rightarrow DEVICE$

In other words, a device is claimed by a user, who consents to it operating on their behalf. The device then connects to a data-sharing system to share data and accept commands. In this model, the specifics of the OAuth2 claim, login and consent flows, as well as the data-sharing model are deliberately expressed in detail. This refinement is important because the specifics of the device's lifecycle and the usability of the system when devices have no user interface or input mechanism are key requirements of the system.

A device is initially a *Fresh Device* (FD). This means that it has just been manufactured and does not have a well-defined identity. This model does not rely on any implicit identity such as a Media Access Control (MAC) address. Instead, the system injects an identity and credential into the device at manufacturing time. A *Registered Device* (RD), on the other hand, is a device that has a credential injected into it during the manufacturing process. Because aspects of this model are closely based on existing OAuth2 model, OAuth2 *ClientIDs* and *ClientSecrets* are explicitly modeled as such credentials. Each device is modeled as a unique OAuth2 Client.

As discussed above, this mapping of OAuth2 to the IoT device world was chosen because it is important that every device has a unique credential.

Definition 5.4. Device Registration

$FD = manufacture \rightarrow updatedevice?cid.cs \rightarrow RD(cid, cs)$

A registered device must then be "owned". This process consists of a User consenting to allow the device to publish data and/or receive commands on the user's behalf. In this system, the OAuth2 approach is extended to support this. The extension is based on the requirement to support devices with almost no User Interface (UI). The only hard requirement on a UI is that the system must have the ability for the user to see if the device is switched on. When the device is switched on, it notifies the DIdP that it is *present*. The user then initiates a *claim* process for the device. This involves the *ClientID* of the device. For example, in the prototype implemented in [7], the device has a QR code printed onto it, which embeds the ClientID into a URL. When the QR code is scanned, it takes the user to a claim page specific to that device.

The specific OAuth2 flow used for this work is the *Authorisation Code* flow. This is a two-part flow whereby the client is first issued an Authorisation Code (AuthCode), which is then "swapped" for a *Refresh Token* and *Bearer Token*. The second step is used to ensure that the client authenticates. This is known as a three-legged flow as the user and the client both authenticate to the DIdP.

Definition 5.5. Ownership

$RD(cid, cs) = presence!cid \rightarrow authcode?ac \rightarrow RD_A(cid, cs, ac)$

$RD_A(cid, cs, ac) = token\_ac\_req!cid.cs.ac \rightarrow token\_ac\_resp?r.a \rightarrow OD(cid, cs, r)$

This produces an *Owned Device* (OD). Every device is either owned or not owned. In other words, there is a function $owns$, such that:

$\forall\, d \in OwnedDevices, \exists\, u \in Users : owns(d) = u$

$\forall\, d \in UnownedDevices, \nexists\, u \in Users : owns(d) = u$

$Devices = UnownedDevices \cup OwnedDevices$

Fresh devices and registered devices are not owned:

$FreshDevices \subseteq UnownedDevices$

$RegisteredDevices \subseteq UnownedDevices$

This definition of ownership is very high-level. In the model, a user must *claim* a device. In the implementation, claiming is done by being the first person to scan a QR code attached to the device, but it could also be done by an NFC chip or simply typing a URL or code into a browser. The claiming process requires the device to be switched on and connected, and therefore *present*. Once the user initiates the claim, they must *login* and then *consent* to the device acting for them.

As a simplification of the model, the device only needs to remember the Refresh Token. In the OAuth2 specification, a client has both a refresh token and a bearer token. The bearer token is inherently insecure as anyone who has a copy of it can act as the client. Therefore, it expires regularly and once it has expired, the client must refresh it, which requires the client to present its credentials to the OAuth2 server. The refresh process leading to a *Secure Device* (SD) is defined follows.

Definition 5.6. Refresh Flow

$OD(cid, cs, r) = token\_refresh\_req.cid.cs.r \rightarrow token\_refresh\_resp?b \rightarrow SD(b, cid, cs, r)$

When a device was modeled above, there was nothing that addressed whether a device can publish events without those events being accepted by the Gateway. In the prototype, the MQTT protocol supports queuing at the Gateway and back-pressure on the client. Modeling queuing adds an unnecessary complexity to the model, as well as being specific to a protocol. On the other hand, if there is nothing addressing queuing or back-pressure, the model also becomes unmanageable because of state explosion. Therefore, this model uses the simplest approach, which is that the Device must wait for an acknowledgement before publishing further data. This is a simple form of back-pressure. This corresponds to protocols like Transport Control Protocol (TCP), where the server acknowledges the client messages, etc. Therefore, the refined definition of a device is as follows.

Definition 5.7. Acknowledging Device

$SENS = sense\$d \rightarrow sd!d \rightarrow dackd \rightarrow SENS$

$ACT = rc?c \rightarrow act!c \rightarrow dackc \rightarrow ACT$

$DV = SENS \;|||\; ACT$

Finally a Secure Device is one that connects using the bearer token and then publishes and subscribes.

Definition 5.8. Secure Device

$SD(bearer, cid, cs, r) = connect!bearer \rightarrow connected \rightarrow DV$

5.2. Manufacturer

In order to register a device, there needs to be an interface that will create a ClientID and a ClientSecret. This Application Programming Interface (API) is defined by the OAuth2 Dynamic Client Registration (DCR) API [40]. The details of the actual API are abstracted in the model. The manufacturer indicates that a device has been manufactured and requests a ClientID and ClientSecret, which are stored in the device.

The device itself does not connect to the DCR API. Instead, there is a manufacturer that requests the credential and updates the device. The reason for this is that in order to call the DCR API on the DIdP, the requestor needs its own credential. Adding this credential to every device would be a significant security issue. In the model, the trust relationship between the Manufacturer and the DIdP is not explicitly modeled but assumed. In an implementation, this is enabled by a specific token for the manufacturer with scope allowing DCR access.

Definition 5.9. Manufacturer

$MAN = manufacture \rightarrow dcrrequest \rightarrow credential?cid.cs \rightarrow updatedevice!cid.cs \rightarrow \checkmark$

This is the only process involving the manufacturer, which means that the manufacturer is not involved in the system any further

5.3. User Identity Provider

The modelling of federated login into the User Identity Provider (UIdP) is purposely minimal. The only requirement is that the UIdP validates the user and provides a unique identifier for the user. In addition, it needs to be clear in the model that the User's credentials are only shared with the UIdP. In the prototype, multiple approaches for UIdP login are supported, mainly based around OAuth2. As in other aspects of the model, it is possible to refine this further to cover those alternative approaches. However, unlike the device authentication and authorisation flows, these flows take place in a normal browser and are well understood, so there is no benefit in further refinement of the model in this area.

The login process starts with a request for login ($login$). The user is either successful in which case a user identifier ($lu.u$) is returned, or it fails ($failure$).

Definition 5.10. Login

$LOGIN(u) = login \rightarrow (success \rightarrow lu.u \rightarrow SKIP \ \Box \ failure \rightarrow STOP)$

This is refined to include a *federated login* ($fedlogin$), which takes a credential (fc).

Definition 5.11. User Identity Provider

$$UIDP = login \rightarrow fedlogin?fc \rightarrow (success \rightarrow lu.u \rightarrow SKIP \ \Box \ failure \rightarrow STOP)$$

The credential is unimportant. The requirement is that there is a bijective function from federated credentials to users:

$fu \in Fedcred \times Users \wedge \mathrm{dom}\, fu = Users \wedge fu(c_1) = fu(c_2) \Leftrightarrow c_1 = c_2$

where $\mathrm{dom}\, fu$ is the domain of fu.

5.4. Device Identity Provider

The *Device Identity Provider* (DIdP) implements all the identity and policy model for the device. It defers user logins to the User Identity Provider (UIdP). The DIdP is modeled as a collection of stateful processes that evolve, based on their interactions with existing devices, manufacturers and users. These processes start with issuing a credential.

Definition 5.12. Device Registration API

$DCR(cid, cs) = dcrrequest \rightarrow credential.cid.cs \rightarrow UR(cid, cs)$

Once a credential is issued, the system must support *User Registration*. User registration for a device (URD) is as follows:

Definition 5.13. User Registration of Devices

$$URD(cid, cs) = presence.cid \rightarrow claim.cid \rightarrow login \rightarrow URD_A(cid, cs)$$

$$URD_A(cid, cs) = success \rightarrow lu?u \rightarrow URD_B(cid, cs, u) \ \Box \ failure \rightarrow error \rightarrow STOP$$

$$URD_B(cid, cs, u) = devconsent \rightarrow authcode\$ac \rightarrow TOKEN\_AC(p(u), cid, cs, ac, PdRc) \ \Box \ noconsent \rightarrow error \rightarrow STOP$$

Firstly, the device must be present. There must be a claim by a user, which initiates a user login, which either succeeds or fails. If it fails then the process ends with an error. Otherwise, the federated login returns a user identifier. Then there must be user consent for a device. This consent is by definition consent for the device to use scope $PdRc$, which means the device can publish data and receive commands. If the consent is granted, the device receives an AuthCode. At this stage the DIdP is now ready to handle the second half of the authorisation code flow (i.e. the swapping of the AuthCode for the Refresh Token by calling $TOKEN\_AC$ and $TOKEN\_REF$, which are defined in Definitions 5.18 and 5.19, respectively).

The scopes defined in this model are deliberately kept simple in order to demonstrate the concept only:

Pd	Publish Data
Pc	Publish Commands
Rd	Receive Data
Rc	Receive Commands
PdRc	Publish Data and Receive Commands
RdPc	Read Data and Publish Commands

Table 2: Scopes and their Meanings

Definition 5.14. Scopes

$Scope = PdRc \mid RdPc \mid Pd \mid Rc \mid Rd \mid Pc$

The meaning of these scopes is described in Table 2. Note that there are other scopes in the overall system (e.g. a *dcr* scope which allows manufacturers to call the DCR system, and an *introspection* scope that allows a gateway to call the introspection API. These are not formally modeled as they don't affect the core model or the proofs of data sharing.

At this stage, the DIdP applies a *pseudonymisation* function $p(u)$, which replaces the username with a random pseudonym. Note that the definition of pseudonymisation in this work is somewhat different to other privacy-enhancing systems. In many systems, the user may choose to use multiple pseudonyms that are shared with third-parties to reduce identifiability. In this system, the pseudonym is automatically generated and remains hidden from third-parties. It is used to make the attack tree more complex by requiring an attacker to attack two different systems to identify a user.

Definition 5.15. Pseudonymisation

$\forall\, u \in Users, p(u) \in Pseud$

$\forall\, u_1, u_2 \in Users,\ p(u_1) = p(u_2) \Leftrightarrow u_1 = u_2$

Before continuing with the device flow of the DIdP, it is useful to examine the application approval process. A user must also authorise an application to be able to interact with the system. In this case, an application is expected to have the scope $RdPc$, which allows it to receive data and publish commands. A further refinement of the model could support apps that only send commands, or only receive data. Similarly the model can support devices that can only act and not sense. However, this refinement is not required to demonstrate the core properties of the model.

Definition 5.16. User Approval of Applications

$$URA(cid, cs) = useraccess \rightarrow login \rightarrow URA_A(cid, cs)$$

$$URA_A(cid, cs) = success \rightarrow lu?u \rightarrow URA_B(cid, cs, u) \ \square\ failure \rightarrow error \rightarrow STOP$$

$$URA_B(cid, cs, u) = appconsent \rightarrow authcode\$ac \rightarrow TOKEN\_AC(p(u), cid, cs, ac, RdPc) \ \square\ noconsent \rightarrow error \rightarrow STOP$$

This definition is analogous to the device consent flow, except for two differences. First, instead of a claim and presence event, there is simply $useraccess$. This event signifies that a user has requested access to an application. Second, the user offers $appconsent$, which provides the app with the $RdPc$ scope.

The overall user approval process is the combination of these two processes:

Definition 5.17. User Approval support in the DIdP

$UR(cid, cs) = URD(cid, cs) \,\Box\, URA(cid, cs)$

This ties back to Definition 5.12. The DIdP supports a Token interface (part of the OAuth2 specification) which supports the AuthCode flow and the Refresh flow.

Definition 5.18. Token Response to AuthCode

$$TOKEN\_AC(p, cid, cs, ac, scope) = \\ token\_ac\_req.cid.cs.ac \rightarrow token\_ac\_resp\$r\$a \rightarrow TOKEN\_REF(p, cid, cs, r, scope)$$

The refresh flow allows a client to send a valid refresh token, together with the client's credentials, and receive a fresh, active bearer token. Once this is done, the DIdP can then support *introspection* of the bearer token.

Definition 5.19. Token Refresh Flow at DIdP

$$TOKEN\_REF(p, cid, cs, r, s) = \\ token\_refresh\_req.cid.cs.r \rightarrow token\_refresh\_resp\$bearer \rightarrow INTRO(bearer, p, s)$$

This introspection is defined by the OAuth2 Introspection API [41]. The introspection looks at a bearer token and returns the validity, user and scope of the token. In the model, introspection only returns the pseudonym instead of the actual user information.

Definition 5.20. Introspection

$$INTRO(b, p, s) = introspect.b \rightarrow valid!p!s \rightarrow INTRO(b, p, s) \\ \Box\, introspect?x : \left\{ x \;\middle|\; x \in Bearer, x \neq b \right\} \rightarrow invalid \rightarrow INTRO(b, p, s)$$

This definition says that the specific process that has previously initialised the client with a bearer b, pseudonym p, and scope s will respond to events "querying" a bearer token. If the bearer token queries matches, the scope and pseudonym will be returned, otherwise it will return $invalid$.

Note that a further refinement of this is possible using *Timed CSP* [42] which allows one to model time. This refinement would include the timing-out of bearer tokens, forcing a refresh. While this would be a nice enhancement to the model, it would not aid in proving any of the fundamental properties of the model and therefore it is left for further work.

The DIdP that supports a given credential is now defined, and this can be generalised:

Definition 5.21. Device Identity Provider

$$DIDP = \Box_{cid \in ClientID, cs \in ClientSecret} DCR(cid, cs)$$

5.5. Third Party Application

The DIdP and the Device, are now defined, so it is a good time to look at the system that interacts at the other end. This is called a *Third Party Application* (TPA) or simply an *App*. The reason that this is called a *third-party* application is that the model enforces that no system is inherently trusted to look at device data. Therefore, every app that wishes to access data from a device or send commands to a device must register with the DIdP and gain consent, just as a third-party would in any OAuth2 flow.

The *data consumer* (DC) is a component that receives data (rd), and then logs that data.

Definition 5.22. Data Consumer

$DC = rd?d \rightarrow logdata.d \rightarrow DC$

A *command publisher* first *demands* an action in the form of a command, and then *sends* that command (sc).

Definition 5.23. Command Publisher

$CP = demand\$c \rightarrow sc!c \rightarrow CP$

An application can support data consumption and command publication concurrently.

Definition 5.24. Application

$APP = DC \;|||\; CP$

Once again there is a refinement of this that supports acknowledgements.

Definition 5.25. Acknowledging Application

$DC_{ack} = rd?d \rightarrow logdata!d \rightarrow aackd \rightarrow DC_{ack}$

$CP_{ack} = demand\$c \rightarrow sc!c \rightarrow aackc \rightarrow CP_{ack}$

$APP_{ack} = DC_{ack} \;|||\; CP_{ack}$

The logic of an application is very similar to the logic of a device. Given the similarity, the logic can be presented more concisely before discussing the differences.

Definition 5.26. Full Application

$$APPCREATE = dcrrequest \rightarrow credential?cid?cs \rightarrow RA(cid, cs)$$

$$RA(cid, cs) = useraccess \rightarrow authcode?ac \rightarrow RA_A(cid, cs, ac)$$

$$RA_A(cid, cs, ac) = token\_ac\_req!cid.cs.ac \rightarrow token\_ac\_resp?r.b \rightarrow SA(cid, cs, r)$$

$$SA(cid, cs, r) = token\_refresh\_req!cid.cs.r \rightarrow token\_refresh\_resp?bearer \rightarrow TA(bearer, cid, cs, r)$$

$$TA(b, cid, cs, r) = connect!b \rightarrow connected \rightarrow APP_{ack}$$

The difference is minor: the application needs a user to request access ($useraccess$). Other than that, the consent flow is analogous as far as the app is concerned. However, the user sees a different flow, explored next.

5.6. User

The only key roles that the users play are:

- to login,
- to claim devices,
- to request access to apps, and
- to provide consent to devices and apps.

The user participates in two flows, consenting to devices and applications. The user can claim a device. In this, the User u specifically claims a device with identity cid. The user must login successfully and consent to the device, or the login may fail, or the user may not agree to the scope of sharing requested.

Definition 5.27. User Registration of a Device

$$USERCLAIM(fc, cid) = claim.cid \rightarrow login \rightarrow (fedlogin!fc \rightarrow (success \rightarrow lu!u \rightarrow (devconsent \rightarrow SKIP \ \square \ noconsent \rightarrow STOP) \ \square \ failure \rightarrow STOP))$$

A given user u may claim any device:

$$UC(fc) = \square_{cid \in ClientID} USERCLAIM(fc, cid)$$

The second user flow is similar, where the user consents to an application seeing their data and sending commands to their devices.

Definition 5.28. User Approval of an Application

$$USERAPPROVE(fc) = useraccess \rightarrow login \rightarrow (fedlogin!fc \rightarrow (success \rightarrow lu!u \rightarrow (appconsent \rightarrow SKIP \ \square \ noconsent \rightarrow STOP)) \ \square \ failure \rightarrow STOP)$$

A user is simply the combination of these two processes.

Definition 5.29. User

$$USER(fc) = UC(fc) \ ||| \ USERAPPROVE(fc)$$

An important aspect of a user's capability is the opportunity to revoke tokens and remove access. This applies to both devices and applications. A device that has a token revoked needs to be informed of the revocation by the DIdP. This is managed by an appropriate return code from the DIdP to the device on the $TOKEN\_REF$ interaction. This then changes the device from the OD stage back to RD, where it can then be claimed by the same or another user. Similarly an App that has been revoked will lose all access and will need to be re-authorised by a user.

5.7. Intelligent Gateway (IG)

The role of the IG is to validate the bearer token of either the device or the app using the introspection defined in 5.20. This returns a pseudonym and a scope. The IG should not share this pseudonym, and the DIdP permissions do not allow other parties to access introspection. Once the IG has a valid response, it passes the message to the Personal Cloud Middleware (PCM), which implements the scope sharing.

Definition 5.30. Gateway

$$IG = connect?b \rightarrow introspect!b \rightarrow (valid?p?s \rightarrow connected \rightarrow PCM(p, s) \ \square \ invalid \rightarrow error \rightarrow STOP)$$

5.8. Personal Cloud Middleware (PCM)

Each user has their own instance of PCM. If u is a user, then $PCM(u)$ is the user's PCM.

$$\forall \, u \in Users \, \exists \, p \in PCM, p = PCM(u)$$

The PCM is modeled in two halves: a sending component and a receiving component. This way, it can be ensured that a device that has permission to publish data is allowed to publish

and an app that is allowed to receive data can correctly subscribe. In order to model the PCM properly, once again there is need to support simple acknowledgements to prevent state explosion in the model. The acknowledgements are in two parts. Firstly, the PCM acknowledges that it has received or sent messages to the device or app (these acknowledgements were seen in Definition 5.7).

The second type of acknowledgements defined as $(pcmacksd, pcmackrd, pcmacksc, pcmackrc)$ allows the two halves of the PCM to acknowledge delivery. For example, if a device is correctly connected to the PCM, but there is no authorised application, the PCM will not acknowledge that messages have been delivered. The connection between the two halves is modeled using the linked parallel construct described above. For example, the "device half" of the PCM will $senddata.p!d$ which will be renamed to be received by the "app half" of the PCM as $recdata.p?d$. Notice that these messages are synchronised on the pseudonym. This models the privacy of the PCM: each user's PCM can only communicate with its other half. The PCM is defined using pattern matching on the different scopes:

Definition 5.31. Personal Cloud Middleware

The "device half" is defined as follows:
$PCM(p, Pd) = sd?d \rightarrow senddata.p!d \rightarrow pcmacksd \rightarrow dackd \rightarrow PCM(p, Pd)$

$PCM(p, Rc) = reccom.p?c \rightarrow rc!c \rightarrow pcmackrc \rightarrow dackc \rightarrow PCM(p, Rc)$

$PCM(p, PdRc) = PCM(p, Pd) \;|||\; PCM(p, Rc)$

The "app half" is defined as:
$PCM(p, Pc) = sc?c \rightarrow sendcom.p!c \rightarrow pcmacksc \rightarrow aackc \rightarrow PCM(p, Pc)$

$PCM(p, Rd) = recdata.p?d \rightarrow rd!d \rightarrow pcmackrd \rightarrow aackd \rightarrow PCM(p, Rd)$

$PCM(p, RdPc) = PCM(p, Rd) \;|||\; PCM(p, Pc)$

One of the concepts that is important in the PCM is that of summarisation and filtering. These were not modeled in the PCM. This is because any proofs established in the CSP model will become intractable in the presence of summarisation and filtering. In addition, the complexity of the model increases making it much harder to analyse automatically using FDR.

A summarisation or filtering is a function defined on a stream of data or commands:

$stream =< d_1, d_2, d_3, d_4, \ldots d_n \ldots >$

$f(stream) =< d'_1, d'_2, \ldots d'_m \ldots >$

where the number of output data elements can be the same, fewer or even more than the number of input data points. Summarisation and filtering are, as identified in [43], key technologies to fight against fingerprinting. In addition, during the creation of this model, we identified the possibility of filtering commands, which did not emerge in [43].

Filtering commands may initially seem to be counter-intuitive: if a device user wishes to turn off a light, that user does not want the light turned off "on average". However, take the example of a connected car. Command filtering could be seen as an example of an application-level firewall for device commands. For example, the filter may allow commands to remotely switch on the engine when the device is parked, but disallow any events that effect the speed or direction of the car. A more complex filtering rule might allow a command to switch off the engine when the speed is zero and the parking brake is applied, but filter out any other attempts to switch off the engine. While there are examples of specific firewalls, e.g, for cars, in the literature, there was no evidence of a more general filtering and summarising approach to commands. Next, we combine the components into a full architecture and to demonstrate specific properties of the system.

6. An Event-Sharing Analysis

This section builds up the composition of individual components and analyses the events that are shared between those components. The first part is to connect a device to the manufacturer, DIdP, user and gateway.

Definition 6.1. System with Connected Device

Connecting the manufacturer and the DIdP:

$DSYS_1 = MAN \ {}_{me}\|_{didpe} \ DIDP$

where me is the set of events that the manufacturer interacts on, and $didpe$ is the set of events that the DIdP interacts on.

$$me = \{manufacture, dcrrequest, credential.cid.cs, updatedevice.cid.cs \mid cid \in ClientID, cs \in ClientSecret\}$$

The set of events that the DIdP interacts with is quite extensive. Firstly, the events between the device and the DIdP are:

$$didp2d = \{presence.cid, authcode.ac, token\_ac\_req.cid.cs.ac, token\_ac\_resp.r.b, token\_refresh\_req.cid.cs.r, token\_refresh\_resp.b \mid cid \in ClientID, cs \in ClientSecret, ac \in AuthCode, r \in Refresh, b \in Bearer\}$$

The events between the manufacturer and the DIdP are:

$$didp2m = \{dcrrequest, credential.cid.cs \mid cid \in ClientID, cs \in ClientSecret\}$$

The events between the DIdP and the application creator are the same:

$$didp2ac = \{dcrrequest, credential.cid.cs \mid cid \in ClientID, cs \in ClientSecret\}$$

The events between the DIdP and the Gateway are:

$$didp2gw = \{introspect.b, valid.p.s, invalid \mid b \in Bearer, p \in Pseud, s \in Scope\}$$

The events between the DIdP and the User are:

$$didp2u = \{lu.u, claim.cid, useraccess, devconsent, appconsent, noconsent, login, success, failure \mid cid \in ClientID, u \in User\}$$

The error event is:

$didperr = \{error\}$

And finally, all the DIdP events are:

$didpe = didp2d \cup didp2m \cup didp2ac \cup didp2gw \cup didp2u \cup didperr$

As discussed in the introduction to CSP above, the *alphabetised parallel* operator $({}_A\|_B)$ has the following rule:

Law 6.1. Alphabetised Parallel

$$P \ {}_A\|_B \ Q = (P \underset{\Sigma \setminus A}{\|} STOP) \underset{A \cap B}{\|} (Q \underset{\Sigma \setminus B}{\|} STOP)$$

where Σ is the set of all events in the system.

In other words, when the alphabetised parallel operator $P \ {}_A\|_B \ Q$ is used, all possible events that communicate between P and Q can be defined as the intersection $A \cap B$. One of the key aspects of this system is where there are multiple parties conjoined using alphabetised parallel. Alphabetised parallel has both symmetry and associativity laws.

Law 6.2. Alphabetised Parallel Symmetry

$$P \ {}_A\|_B \ Q = Q \ {}_B\|_A \ P$$

Law 6.3. Alphabetised Parallel Associativity

$$(P \ {}_A\|_B \ Q) \ {}_{A \cup B}\|_C \ R = P \ {}_A\|_{B \cup C} \ (Q \ {}_B\|_C \ R)$$

This gives the following theorem:

Theorem 6.1. (Manufacturer to DIdP Events)

As this is the only place where the DIdP interacts with the manufacturer, we can assert that the Manufacturer only sees the events $didpe \cap me$:

$$\{dcrrequest, credential.cid.cs \mid cid \in ClientID, cs \in ClientSecret\}$$

The proof is obvious instantiation of Law 6.1. In addition, this can be derived using the probe capabilities of FDR that allow this combined process to be explored.

Of course the manufacturer also interacts with the device, but in this case the device is simply receiving data that the manufacturer provides to the device. One thing to note is that the manufacturer can retain the ClientID and ClientSecret. A possible improvement to the model would be to enable a flow whereby the device directly contacts the DIdP to update the ClientSecret, which would ensure that the manufacturer is not in possession of the device credentials. This increases security at the cost of making it harder for the manufacturer to provide support. This is left for further work.

The next stage of constructing the system is to conjoin this with a fresh device:

$$DSYS_2 = FD \ {}_{deve}\|_{didpe \cup me} \ DSYS_1 \setminus me$$

where $deve$ is the set of events that the device communicates on. Note that the overall set of events that a device interacts on are specified here individually, but later the actual events that two parties interact on is formally derived using laws of CSP.

The device to manufacturer events are:

$$dme = \{manufacture, updatedevice.cid.cs \mid cid \in ClientID, cs \in ClientSecret\}$$

The device to DIdP events are:

$$\begin{aligned} dde = \{&presence.cid, authcode.ac, token\_ac\_req.cid.cs.ac, token\_ac\_resp.r.b, \\ &token\_refresh\_req.cid.cs.r, token\_refresh\_resp.b \mid \\ &cid \in ClientID, cs \in ClientSecret, \\ &ac \in AuthCode, r \in Refresh, b \in Bearer\} \end{aligned}$$

Note that $dde = didp2d$.

The device to gateway events are:

$$dge = \{connect.b, connected, sd.d, rc.c, dackd, dackc \mid b \in Bearer, d \in Data, c \in Command\}$$

The devices own events are:

$hde = \{act.c, sense.d \mid d \in Data, c \in Command\}$

All device events are the union of these:

$deve = dme \cup dde \cup dge \cup hde$

The device is conjoined with both the DIdP and the Manufacturer. Once this happens, the the manufacturer's events are no longer visible, and therefore are hidden.

Theorem 6.2. (Device to Manufacturer Events)

The device to manufacturer events are defined as $deve \cap me$. The intersection is calculated as:

$\{manufacture, updatedevice.cid.cs \mid cid \in ClientID, cs \in ClientSecret\}$

This can be proven using FDR or by applying the alphabetised parallel rule twice.

Theorem 6.3. (Device to DIdP Events)

The device to DIdP events are defined as the intersection $didpe \cap deve$. This intersection is calculated by FDR as:

$$\{presence.cid, authcode.ac, token\_ac\_req.cid.cs.ac, token\_ac\_resp.r.b, token\_refresh\_req.cid.cs.r, token\_refresh\_resp.b \mid cid \in ClientID, cs \in ClientSecret, ac \in AuthCode, r \in Refresh, b \in Bearer\}$$

In order to authorise the system, the user must connect to the UIdP. Firstly, the events that the UIdP interacts on:

$fedevents = \{fedlogin.fc, login, lu.u, success, failure \mid u \in User, fc \in Fedcred\}$

The set of events that the federated user interacts on are:

$$fue = \{claim.cid, useraccess, fedlogin.fc, lu.u, login, success, failure, appconsent, devconsent, noconsent \mid cid \in ClientID, fc \in Fedcred, u \in User\}$$

Once the federated user is combined with the UIdP, this communicates with the DIdP on a set of events due:

$$due = \{claim.cid, useraccess, lu.u, login, success, failure, appconsent, devconsent, noconsent \mid cid \in ClientID, u \in User\}$$

A user claiming a device is defined as:

$FUSER(fc) = UC(fc) \ {}_{fue}\|_{fedevents} \ UIDP \upharpoonright due$

Theorem 6.4. (User to UIdP Events)

By application of the alphabetised parallel law[1], it can be ascertained that the user to UIdP events are $fue \cap fedevents$. These are calculated by FDR as:

$$\{login, lu.u, success, failure, fedlogin.fc \mid u \in User, fc \in FedCred\}$$

Now this user can be conjoined with the existing system, hiding the $login$ and lu events from the rest of the system. $DSYS3(fc)$ indicates the user with credentials fc approving a fresh device (with the associated DIdP, UIdP and manufacturer).

$$DSYS_3(fc) = DSYS_2 \ {}_{didpe \cup deve}\|_{due} FUSER(fc) \setminus \{login, lu.u \mid u \in User\}$$

Theorem 6.5. (UIdP to DIdP Events)

Using the Associative Law 6.3 it can be shown that the DIdP to UIdP events are defined as the set $(didpe \cup deve) \cap fedevents$, which (using FDR) evaluates to:

$$\{login, lu.u, success, failure \mid u \in User\}$$

Theorem 6.6. (User to DIdP Events)

The user to DIdP events are defined as the set $(didpe \cup deve) \cap ue$. These are calculated by FDR.

$$\{devconsent, appconsent, noconsent, login, lu.u, success, failure, claim.cid, useraccess \mid u \in User, cid \in ClientID\}$$

Definition 6.2. Device System

This is now joined with the Gateway (IG):

$$DSYS(fc) = IG \ {}_{gwe \cup pcme}\|_{didpe \cup gwe \cup hde} DSYS_3(fc)$$

where gwe are the events that the gateway interacts on, and $pcme$ are the events that the two halves of the PCM interact on:

$$gwe = \{connect.b, connected, introspect.b, valid.p.s, invalid, dackd, dackc, aackd, aackc, rd.d, sd.d, rc.c, sc.c \mid b \in Bearer, p \in Pseud, d \in Data, c \in Command, s \in Scope\}$$

$$pcme = \{senddata.p.d, sendcom.p.c, recdata.p.d, reccom.p.c, pcmackrd, pcmacksd, pcmacksc, pcmackrc \mid d \in Data, c \in Command, p \in Pseud\}$$

This process ($DSYS(fc)$) indicates a user with credential fc approving a device that is now fully connected to the rest of the system (IG, DIdP, UIdP, Manufacturer and User).

Theorem 6.7. (Device to Gateway Events)

Once again the associative law can be used to calculate all events that the device can communicate with the gateway on, which are defined as the set $(gwe \cup pcme) \cap deve$:

$$\{sd.d, rc.c, connect.b, connected, dackd, dackc \mid d \in Data, c \in Command, b \in Bearer\}$$

[1]Note that the user is also present in the app approval process below. The law is applied to both conditions, but the theorem is presented here for reasons of explication.

Theorem 6.8. (Gateway to DIdP Events)

The events shared between the DIdP and the IG are derived as $(gwe \cup pcme) \cap (didpe \cup hde)$, which evaluates to:

$$\{introspect.b, valid.p.s, invalid \mid b \in Bearer, p \in Pseud, s \in Scope\}$$

Theorem 6.9. (The Gateway Shares no Events with the UIdP, the User and the Manufacturer)

By similar calculations of the alphabetised parallel laws, it is shown that the set of events shared between the gateway and the manufacturer, user, and UIdP are all empty.

The creation of the App half of the system is analogous, and therefore is presented without discussion.

Definition 6.3. Application Connected to the DIdP, UIdP, User and IG

$$FUA(fc) = USERAPPROVE(fc) \ {}_{fue}\|_{fedevents} \ UIDP \upharpoonright due$$

All the events an app can participate in are defined as ae, and the set of events that the DIdP communicates with apps as $appdidpe$. The user to App events are uae.

$$ASYS_1 = DIDP \ {}_{appdidpe}\|_{ae} \ APPCREATE$$

$$ASYS_2(fc) = (ASYS_1 \ {}_{didpe \cup ae}\|_{uae} FUA(fc)) \setminus \{login, lu.u \mid u \in User\}$$

$$ASYS(fc) = IG \ {}_{gwe \cup pcme}\|_{appdidpe \cup gwe \cup hae} ASYS_2(fc)$$

The process $ASYS(fc)$ indicates an app that has been approved by user with credential fc that is connected to the DIdP, UIdP, User, and IG.

6.1. The Complete System

Finally, these two systems ($DSYS$ and $ASYS$) are connected together. They synchronise only at the PCM using the PCM events. This is modeled using the linked parallel approach described above. The parameterisation of the users' credentials is retained so that the system can be tested with different users (with one credential being used to authorise the device and the other to authorise the app).

Definition 6.4. The Complete System

$$SYS(fc_1, fc_2) = \left(DSYS(fc_1) \begin{bmatrix} reccom \leftrightarrow sendcom \\ pcmacksd \leftrightarrow pcmackrd \\ pcmackrc \leftrightarrow pcmacksc \\ senddata \leftrightarrow recdata \end{bmatrix} ASYS(fc_2) \right) \Theta_{\{error\}} STOP$$

Theorem 6.10. (Application Events)

The same logic as above can be applied to calculate the events that the App shares with different components. The proofs can be derived by the use of the Associative Law, or more effectively through probing the model with FDR. These events are summarised in Table 3.

7. Properties of the OAuthing System

Two sets of properties of the system are derived from the model presented above: end-to-end and data sharing properties.

User	$\emptyset$
UIdP	$\emptyset$
Device	$\emptyset$
Manufacturer	$\emptyset$
DIdP	$\{presence.cid, authcode.ac, token\_ac\_req.cid.cs.ac, token\_ac\_resp.r.b, token\_refresh\_req.cid.cs.r, token\_refresh\_resp.b \mid cid \in ClientID, cs \in ClientSecret, ac \in AuthCode, r \in Refresh, b \in Bearer\}$
Gateway	$\{rd.d, sc.c, connect.b, connected, aackd, aackc \mid d \in Data, c \in Command, b \in Bearer\}$

Table 3: Events Shared Between the App and Other Components

7.1. End-to-End Analysis

The definition of a complete system allows reasoning about the whole system and not just its individual components. To do this, specifications can be defined in CSP that are either expected to pass or fail.

Theorem 7.1. (Consent Cannot Follow Failed Login)

In order to demonstrate that consent must follow successful login, an "anti-specification" can be disproved:

$LSPEC = failure \rightarrow appconsent \rightarrow STOP \Box\ failure \rightarrow devconsent \rightarrow STOP$

The specification suggests that the model will support $appconsent$ or $devconsent$ after a failure. This must be disproven. The specification will only be evaluated on these events:

$lspecevents = \{error, appconsent, devconsent, success, failure\}$

Because this is an anti-specification, disproving means proving that the traces of this are not a subset of the traces of the system.

Therefore FDR is used to evaluate whether or not:

$$SYS(FC.0, FC.0) \upharpoonright lspecevents \sqsubseteq_{\mathrm{T}} LSPEC$$

where $FC.0$ is a valid credential in the system.

FDR shows that this is not true with the following trace of events:

```
Counterexample (Trace Counterexample)
            Specification Debug:
                Trace: <failure>
                Available Events: {error, success}
            Implementation Debug:
                LSPEC (Trace Behaviour):
                    Trace: <failure>
                    Error Event: devconsent
```

which means that the system does not accept $devconsent$ after $failure$.

A positive specification is that when the system is properly consented on both halves, by the same user, then data that is sensed in the device will be logged in the app, and commands that are demanded in the app, will be acted on in the device.

Definition 7.1. Normal System Specification (NS)

This outlines the possibilities for consent. Either there are both of the consents (in either order), or consent on one side followed by an error, or an error. Internal choice properly identifies the specification because it does not consider whether the consent failed because of a failed login, or because the user refused to grant consent, and therefore there are non-deterministic choices of how these event patterns emerge.

$$NSSUCC = appconsent \rightarrow devconsent \rightarrow NS2 \sqcap devconsent \rightarrow appconsent \rightarrow NS2 \sqcap error \rightarrow STOP \sqcap appconsent \rightarrow error \rightarrow STOP \sqcap devconsent \rightarrow error \rightarrow STOP$$

A successful operation is defined as:

$$NS2 = sense\$d \rightarrow logdata.d \rightarrow NS2 \,\Box\, demand\$c \rightarrow act.c \rightarrow NS2$$

This specification only considers the following events:

$$specevents = \{error, appconsent, devconsent, logdata.d, act.c \mid d \in Data, c \in Command\}$$

The overall specification is then:

$$NS = NSSUCC \,\Theta_{\{error\}}\, STOP \upharpoonright specevents$$

The system is now evaluated using FDR's refinement checker.

Theorem 7.2. (The System Meets the Specification if the Same User Authorises the Device and the App)

$$NS \sqsubseteq_{\mathrm{T}} SYS(FC.0, FC.0) \upharpoonright specevents \wedge NS \sqsubseteq_{\mathrm{F}} SYS(FC.0, FC.0) \upharpoonright specevents \wedge NS \sqsubseteq_{\mathrm{FD}} SYS(FC.0, FC.0) \upharpoonright specevents$$

This is proved by FDR:

```
NS [T= SYS(FC.0, FC.0) |\ specevents:
    Log:
        Result: Passed
        Visited States: 1,461,364
        Visited Transitions: 7,440,420
        Visited Plys: 68
        Estimated Total Storage: 268MB
NS [F= SYS(FC.0, FC.0) |\ specevents:
    Log:
        Result: Passed
        Visited States: 1,461,364
        Visited Transitions: 7,440,420
        Visited Plys: 68
        Estimated Total Storage: 268MB
NS [FD= SYS(FC.0, FC.0) |\ specevents:
    Log:
        Result: Passed
        Visited States: 1,461,364
        Visited Transitions: 7,440,420
        Visited Plys: 68
        Estimated Total Storage: 268MB
```

Theorem 7.3. (The System Fails the Specification if Different Users Authorise the Device and the App)

If user with credential $FC.0$ authorises the device and user with credential $FC.1$ authorises the app then there are no data or commands transferred:

$$SYS(FC.0, FC.1) \upharpoonright \ specevents \sqsubseteq_{\mathrm{T}} NS$$

In other words, this tests whether the correct traces are a subset of the traces of the incorrect system, and it is desired that this is not the case. This is once again proved by FDR, which postulates the following counter-example:

```
SYS(FC.0, FC.1) |\ specevents [T= NS:
    Log:
        Result: Failed
        Visited States: 27
        Visited Transitions: 42
        Visited Plys: 8
        Estimated Total Storage: 0B
        Counterexample (Trace Counterexample)
            Specification Debug:
                Trace: <appconsent, devconsent>
                Available Events: {}
            Implementation Debug:
                NS (Trace Behaviour):
                    Trace: <tau, tau, tau, tau,
                    appconsent, devconsent, tau, tau>
                    Error Event: logdata.D.0
```

This shows that even though $appconsent$ and $devconsent$ occur, no data is transferred as the $logdata$ event cannot occur in the system where different users consent.

7.2. Data Flow Between Components

In the analysis of the model above, the messages that flow between components are identified. Those findings are summarised in Table 4, which captures all the data elements that are transferred in messages between each component.

	Man	**Dev**	**User**	**UIdP**	**DIdP**	**GW/PCM**	**App**
Man	—	$\{ClientID, ClientSec\}$	$\emptyset$	$\emptyset$	$\{ClientID, ClientSec\}$	$\emptyset$	$\emptyset$
Device	$\{ClientID, ClientSec\}$	—	$\emptyset$	$\emptyset$	$\{ClientID, ClientSec, AuthCode, Bearer\}$	$\{Bearer, Data, Command\}$	$\emptyset$
User	$\emptyset$	$\emptyset$	—	$\{FedCred, User\}$	$\{User, ClientID\}$	$\emptyset$	$\emptyset$
UIdP	$\emptyset$	$\emptyset$	$\{FedCred, User\}$	—	$\{User\}$	$\emptyset$	$\emptyset$
DIdP	$\{ClientID, ClientSec\}$	$\{ClientID, ClientSec, AuthCode, Bearer\}$	$\{User, ClientID\}$	$\{User\}$	—	$\{Bearer, Pseud, Scope\}$	$\{ClientID, ClientSec, AuthCode, Bearer\}$
GW/PCM	$\emptyset$	$\{Bearer, Data, Command\}$	$\emptyset$	$\emptyset$	$\{Bearer, Pseud, Scope\}$	—	$\{Bearer, Data, Command\}$
App	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{ClientID, ClientSec, AuthCode, Bearer\}$	$\{Bearer, Data, Command\}$	—

Table 4: Component Data-Sharing Matrix

This data-sharing matrix captures a key property of the system. The distribution and sharing of data between components can be used to analyse the security and privacy properties of

the system. This is done by threat modeling. Threat modeling allows the analysis of different attacks on the system. Here, we only show this matrix as a demonstration of the information that can be obtained from such a formal analysis exercise, however, in a future work, we plan to use this matrix in as an input to a STRIDE [44] or a LINDDUN [45] analysis to properly understand the security and privacy properties of the OAuthing system.

8. Conclusion and Future Work

As was shown in [7], the OAuthing personal cloud-based IoT data management system provides significant improvements over existing IoT approaches, leading to much stronger guarantees of privacy, where data and identity information are not shared without consent, and personal sensitive data can be shared anonymously. We presented in this paper a formal model of the OAuthing system. We used this model to analyse end-to-end and data-sharing properties of the system.

In summary, the formal model of the system has shown that the system cannot transfer messages unless:

- A user has successfully logged in to consent,
- the user has granted consent for applications to read data and publish commands,
- the user has granted consent for devices to publish data and read commands, and finally,
- the same user has granted consent for both the application and device.

As a major result, this work proves that the OAuthing approach permits data to be shared between devices and apps if and only if the same user has consented to both systems being authorised to do so.

Future work will focus on ustilising the results of the analysis in performing a comprehensive security and privacy analysis for OAuthing using by interpreting the results using frameworks such as STRIDE [44] and LINDDUN [45]. We also plan to model the OAuthing system using other formal specification and analysis methods, e.g. Event-B [46, 47], in order to better understand the properties of the system. Another important direction for future work would consider the application of the system and the current formal analysis to a real-world case study, e.g. from the domains of manufacturing systems or power grid systems, both of which are relevant to IoT systems.

References

[1] D. Evans, The internet of things, How the Next Evolution of the Internet is Changing Everything, Whitepaper, Cisco Internet Business Solutions Group (IBSG).

[2] T. Kohno, A. Broido, K. C. Claffy, Remote physical device fingerprinting, IEEE Transactions on Dependable and Secure Computing 2 (2) (2005) 93–108.

[3] J. Wei, DDoS on Internet of Things–a big alarm for the future, `http://www.cs.tufts.edu/comp/116/archive/fall2016/jwei.pdf` (2016).

[4] N. V. Database, Cve-2014-9222, `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9222`, (Accessed on 02/02/2017) (12 2014).

[5] P. Fremantle, B. Aziz, J. Kopecký, P. Scott, Federated identity and access management for the internet of things, in: 2014 International Workshop on Secure Internet of Things, 2014, pp. 10–17.

[6] P. Fremantle, J. Kopecký, B. Aziz, Web api management meets the internet of things, in: F. Gandon, C. Guéret, S. Villata, J. Breslin, C. Faron-Zucker, A. Zimmermann (Eds.), The Semantic Web: ESWC 2015 Satellite Events, Springer International Publishing, Cham, 2015, pp. 367–375.

[7] P. Fremantle, B. Aziz, OAuthing: Privacy-enhancing federation for the Internet of Things, in: 2016 Cloudification of the Internet of Things (CIoT), 2016, pp. 1–6.

[8] C. A. R. Hoare, Communicating sequential processes, in: The origin of concurrent programming, Springer, 1978, pp. 413–443.

[9] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[10] A. W. Roscoe, Understanding concurrent systems, Springer Science & Business Media, 2010.

[11] A. B. A. R. Thomas Gibson-Robinson, Philip Armstrong, FDR3 — A Modern Refinement Checker for CSP, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 8413 of Lecture Notes in Computer Science, 2014, pp. 187–201.

[12] D. Hammer-Lahav, D. Hardt, The oauth2.0 authorization protocol. 2011, Tech. rep., IETF Internet Draft (2011).

[13] C. Alcaraz, J. Lopez, S. Wolthusen, Policy enforcement system for secure interoperable control in distributed smart grid systems, Journal of Network and Computer Applications 59 (2016) 301 – 314. doi:https://doi.org/10.1016/j.jnca.2015.05.023. URL http://www.sciencedirect.com/science/article/pii/S1084804515001629

[14] C. Alcaraz, J. Lopez, Secure interoperability in cyber-physical systems, in: Security Solutions and Applied Cryptography in Smart Grid Communications, IGI Global, USA, IGI Global, USA, 2017, Ch. 8, pp. 137–158.

[15] C. Alcaraz, Secure Interconnection of IT-OT Networks in Industry 4.0, Springer International Publishing, Cham, 2019, pp. 201–217.

[16] Power systems management and associated information exchange Data and communications security Part 8: Role-based access control (Preview), https://webstore.iec.ch/preview/info_iec62351-8%7Bed1.0%7Den.pdf, accessed: 24-06-2019.

[17] D. Ferraiolo, R. Kuhn, Role-based Access Controls, in: Proceedings of 15th NIST-NSA National Computer Security Conference, Baltimore, MD, USA, 1992, pp. 554–563.

[18] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, Role-based access control models, Computer 29 (2) (1996) 38–47.

[19] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, Discovering concrete attacks on website authorization by formal analysis1, Journal of Computer Security 22 (4) (2014) 601–657.

[20] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, S. Singh, Formal verification of oauth 2.0 using alloy framework, in: Communication Systems and Network Technologies (CSNT), 2011 International Conference on, IEEE, 2011, pp. 655–659.

[21] D. Jackson, Alloy 3.0 reference manual, Software Design Group.

[22] T. Lodderstedt, M. McGloin, P. Hunt, Oauth 2.0 threat model and security considerations (2013).

[23] A. Banks, R. Gupta, MQ Telemetry Transport (MQTT) V3.1.1 Protocol Specification: Committee Specification Draft 02 / Public Review Draft 02 (2014).

[24] B. Aziz, A formal model and analysis of an iot protocol, Ad Hoc Networks 36 (P1) (2016) 49–57.

[25] R. Baldoni, M. Contenti, S. T. Piergiovanni, A. Virgillito, Modelling Publish/Subscribe Communication Systems: Towards a Formal Approach, in: 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), IEEE Computer Society, 2003, pp. 304–311.

[26] L. Abidi, C. Cerin, S. Evangelista, A Petri-Net Model for the Publish-Subscribe Paradigm and Its Application for the Verification of the BonjourGrid Middleware, in: Proceedings of the 2011 IEEE International Conference on Services Computing, SCC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 496–503.

[27] C. Wang, A. Carzaniga, D. Evans, A. Wolf, Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems, in: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9, HICSS '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 303–.

[28] D. Garlan, S. Khersonsky, J. S. Kim, Model checking publish-subscribe systems, in: Proceedings of the 10th International Conference on Model Checking Software, SPIN'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 166–180.

[29] M. H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, M. Sebastianis, Model checking publish/subscribe notification for thinkteam ®, Electron. Notes Theor. Comput. Sci. 133 (2005) 275–294.

[30] Y. Jia, E. L. Bodanese, C. I. Phillips, J. Bigham, R. Tao, Improved reliability of large scale publish/subscribe based moms using model checking, in: 2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014, IEEE, 2014, pp. 1–8.

[31] L. Baresi, C. Ghezzi, L. Mottola, On accurate automatic verification of publish-subscribe architectures, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 199–208.

[32] F. He, L. Baresi, C. Ghezzi, P. Spoletini, Formal analysis of publish-subscribe systems by probabilistic timed automata, in: Formal Techniques for Networked and Distributed Systems - FORTE 2007, 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings, Vol. 4574, Springer, 2007, pp. 247–262.

[33] S. Gallotti, C. Ghezzi, R. Mirandola, G. Tamburrelli, Quality prediction of service compositions through probabilistic model checking, in: Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures, QoSA '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 119–134.

[34] A. Fehnker, L. V. Hoesel, A. Mader, Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks, in: Proceedings of the 6th International Conference on Integrated Formal Methods, IFM'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 253–272.

[35] A. Fehnker, P. Gao, Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast Protocols, in: Proceedings of the 5th International Conference on Ad-Hoc, Mobile, and Wireless Networks, ADHOC-NOW'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 128–141.

[36] F. Heidarian, J. Schmaltz, F. W. Vaandrager, Analysis of a clock synchronization protocol for wireless sensor networks, Theor. Comput. Sci. 413 (1) (2012) 87–105.

[37] J. Rumbaugh, I. Jacobson, G. Booch, Unified modeling language reference manual, the, Pearson Higher Education, 2004.

[38] S. et al., Openid connect core 1.0, Tech. rep., OpenID Foundation (2014).

[39] J. Tretmans, Model based testing with labelled transition systems, Formal methods and testing (2008) 1–38.

[40] N. Sakimura, J. Bradley, M. Jones, Final: OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1 (2015).
URL http://openid.net/specs/openid-connect-registration-1_0.html

[41] J. Richer, Oauth token introspection (2013).

[42] J. Davies, S. Schneider, A brief history of timed csp, Theoretical Computer Science 138 (2) (1995) 243–271.

[43] P. Fremantle, P. Scott, A survey of secure middleware for the internet of things, PeerJ Computer Science 3 (2017) e114.

[44] A. Shostack, Threat modeling: Designing for security, John Wiley & Sons, 2014.

[45] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, W. Joosen, A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements, Requirements Engineering 16 (1) (2011) 3–32.

[46] J.-R. Abrial, Modeling in Event-B: system and software engineering, Cambridge University Press, 2010.

[47] S. Schneider, H. Treharne, H. Wehrheim, A csp account of event-b refinement, arXiv preprint arXiv:1106.4098.