Master thesis
Computer Science

Radboud University

Attribute-based authentication & signatures for regulating home access

Author:
Koen van Ingen
koen@vaningen.email

Supervisor:
prof. dr. B.P.F. Jacobs
bart@cs.ru.nl

Tippiq supervisor:
E. van Gelderen
edward@tippiq.nl

Second accessor:
W. Lueks MSc.
w.lueks@cs.ru.nl

August 2016

Abstract

Internet of Things (IoT) is a new technology that is becoming increasingly popular. With IoT, every device in a house can be connected to the Internet, which imposes huge privacy problems. Also, it could be useful if a house could identify itself to other parties using attributes. For instance, a mailman can use the address attribute of a house to verify if a package is delivered to the right house and a boiler can retrieve a certification mark attribute if it is maintained correctly. IRMA is a privacy-friendly attribute-based identity management system, developed by the Radboud University and based on the Idemix technology. IRMA includes both attribute-based authentication and attribute-based signatures.

In this thesis, we will explore how we can use IRMA technology to provide a house with its own identity and to control IoT devices in a house in a privacy friendly manner. For this control, we make use of the Tippiq house rules. Tippiq is a project from the company Alliander. We propose the needed adaptations in IRMA for some concrete IoT scenarios. One adaption in IRMA is about attribute-based signatures. These signatures are used to ensure integrity and authenticity of Tippiq house rules with IRMA attributes. We integrate all the components in a proof of concept to illustrate how IRMA can be applied in a concrete IoT scenario: enabling access control to a house using IRMA signatures and Tippiq house rules.

Acknowledgements

This document represents my master's thesis as fulfillment for the degree of Master of Science (MSc) in Computing Science at the Radboud University Nijmegen. The research has been conducted within the Digital Security department of the Institute for Computing and Information Sciences at the Radboud University in Nijmegen in cooperation with Alliander N.V.

First and foremost, I would like to thank my supervisor, Bart Jacobs, for his continuous support and helpful feedback during my research. I would also like to show gratitude to Wouter Lueks, for sharing his opinions and thoughts on the implementation of IRMA signatures, as well as his effort in proofreading my thesis and providing extensive feedback. I am also grateful to Brinda Hampiholi for her suggestions on IRMA signatures.

I would also like to thank all the members of the Tippiq team. They really kept me motivated. It was a real pleasure to work with them. Firstly, I would like to thank Dirk Bollen, who provided me lots of feedback, ideas and suggestions on the integration of this project with Tippiq. Secondly, I would like to say thanks to Edward van Gelderen, for proofreading every part of my thesis and providing feedback. Additionally, I would like to acknowledge Tim Janssen and Mahmut Tumkaya for their help in developing a proof of concept. Finally, I would like to thank Luuk van Hees, for helping me building the proof of concept with a real 'door'.

I am also very grateful to my family and fellow students, who always wanted to help me. My thanks goes especially to Joost Rijneveld, who never hesitated to help me out when I asked him.

Contents

Chapter 1

Introduction

Internet of Things is a new and upcoming technology. Two aspects of this technology are *identification* and *authentication*. We start by introducing these two concepts in Section 1.1. After this, we look at the IRMA technology and how it improves identification schemes in Section 1.2. We introduce the company Alliander, along with the Tippiq project in Section 1.3, where my internship for this thesis is done. Finally, we show in Section 1.4 why all these concepts are useful for Internet of Things devices.

1.1 Authentication and identification

The concept of identification is very common today: everyone is used to identifying him/herself in a variety of different situations. Identification is about saying who you are. However, just *saying* who you are is often not enough: you also need to *prove* who you are. For example, when opening a bank account or booking a hotel, you have to prove that you are who you say you are. Proving who you are is called authentication and this will often involve revealing your complete identity. An example is buying alcohol: here, you have to show your identity card to prove that you are old enough to buy alcohol, but by showing this identity card, all the information that is listed on this card is revealed. This is not necessary and is in conflict with data minimisation that is required by the European Union.[1]

The party that requests proof of identity usually provides something that a user requests, and can therefore be seen as a service provider. For example, a bank provides bank accounts and a hotel rents rooms to its customers. The user has to prove his/her identity to the service provider to obtain the service.

Identities are given out by a trusted party, the identity provider. The identity provider is trusted by both the service provider and by the person that provides a proof of his/her identity. An example of an identity proof is showing a valid identity card. This card is given out by, for instance, the government who can be understood as a trusted party that is the official source of identity documents. The government is in

[1] `https://secure.edps.europa.eu/EDPSWEB/edps/EDPS/Dataprotection/Glossary/pid/74`

this case the identity provider.

Some parties use their own identification and authentication methods, rather than relying on a 'formal' identity that is issued by a trusted authority. An example of a custom identification method is a discount program in a shop, where a discount can be obtained on certain products after scanning a loyalty card. Banks also use their own identification and authentication methods in the form of a debit card. This debit card is used to prove that a payment is done by the card holder.

Authentication is also used in online scenarios. For example, the Dutch government has developed the DigiD system,[2] which is used as an online authentication system for government services. In this case, the DigiD system provides an online identity, and DigiD accounts are given out by the government. Most Dutch government services use DigiD as their authentication system, which means that a Dutch citizen needs to obtain only one online account (a DigiD account) for most of his/her government services.

Other commercial parties are also involved in online authentication. A common example is Facebook's Single Sign-on solution.[3] A service provider can let a user use his/her Facebook identity to log in to their service with Facebook's Single Sign-on. In this case, Facebook can be seen as the identity provider.

A problem that often occurs with (both online and offline) authentication systems is linkability. Linkability is the ability of an identity provider to link different authentication sessions of its users together. An example can be a public transport card, where each user uses an electronic card to pay for his/her journeys. This card links all these journeys together, which is unnecessary and can be privacy problem. Another example is Facebook's Single Sign-on solution: Facebook can see which services the user uses and service providers can cooperate together and link users between services.

There has been a movement from a purely offline and physical identity to an identity that is also usable online. We see that more and more services rely on an online identity. For instance, DigiD allows government services to be accessible online. But what would be the next step in identification? Many believe that Internet of Things could be this next step.

Internet of Things (IoT) is relatively new and still a developing technology. One part of this technology is about identification: each device could also be part of a person's identity or could even have an identity itself.

We have now seen examples of how identification and authentication are used in practice in both online and offline scenarios. We saw that identification can be wrongly used in terms of privacy: for buying alcohol we often have to reveal a complete identity, while only an age proof is sufficient. Your age is the only part of your identity that has to be revealed in this case.

In other scenarios, different parts of an identity are needed. For example a name, exact date of birth, nationality, whether someone is a student, et cetera. All these parts together build up a complete identity. We will from now on call these parts *attributes*. We can then define an identity as a collection of attributes that hold for a

[2] `https://www.digid.nl/`

[3] `https://www.facebook.com/notes/keith-watson/single-sign-on-notes/10150294801872451/`

particular person.

If we use attributes for an identity proof, we can reveal only the minimal part of an identity that is needed for the requesting service provider. For buying alcohol, only an age proof is needed, and for a discount in a certain supermarket, only a proof of a supermarket attribute could be enough to take part in the discount program of that supermarket. Disclosure of a complete identity is certainly not needed in this case.

We need to divide an identity into the right attributes to achieve the desired minimisation. For buying alcohol, one could use an *over18* attribute and for free traveling with public transport as a student, one could use a *student* attribute. A complete set of attributes could build up an identity, but one attribute on its own can be completely anonymous. This is the case with the *over18* attribute. Then, the service provider obtains only a proof that his/her customer is a person that is over eighteen years old and nothing more.

1.2 The IRMA project

IRMA is an ecosystem from the Radboud University that is using attribute-based authentication. This means that only the attributes that are really needed by the service provider are revealed to the service provider. IRMA is an acronym for 'I reveal my attributes'. IRMA uses the Idemix technology developed by IBM [1].

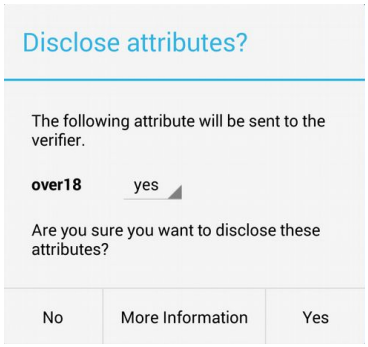
All the IRMA attributes are stored on a secure token, which can be a smart card (as an alternative to the 'classic' identity card), or a smartphone. The underlying Idemix technology allows someone to reveal an attribute by giving a zero-knowledge proof. In this way, one can cryptographically prove that he/she possesses a certain attribute, without revealing the corresponding private key or anything else beyond the requested attribute. We will only consider the case where attributes are stored on a smartphone.

Attributes need to be obtained and stored on a secure token (smartphone), which is a process we call 'issuing' and is done by an authority. We call this authority the *issuer*. An issuer can issue credentials, which are cryptographic containers that group attributes. An example is a 'Government root' credential that contains a person's identity with a name, social security number and a date of birth attribute. This root credential is issued by the government, which is the identity provider for this credential.

Credentials are signed by the issuer's private key and bound to the user's private key that is generated and stored during the initialisation phase of the user's IRMA token.[4] The user's private key is needed to provide an attribute proof to a service provider. This key is securely stored, which means that the user can use it to prove possession of his/her attributes, but he/she is not supposed to directly access it. Attributes are therefore non-transferable: it is not possible to copy 'your' *over18* attribute to someone else's smart card or smartphone, because direct access to the private key is needed to achieve this.

[4]See the link `https://www.irmacard.org/wp-content/uploads/2013/05/Idemix_overview.pdf` for a short overview on how this mathematically works

1) The website/verifier that wants an attribute proof from the user shows a QR code. The user scans this code with his/her smartphone



2) The user allows the *over18* attribute to be revealed to the verifier on his/her smartphone

3) Both the website and the user's smartphone will show a success dialog if the IRMA proof is successful

Figure 1.1: Example of the user interaction when disclosing IRMA attributes to a verifier

Besides being non-transferable, attributes are also unlinkable. Unlinkable means that a service provider cannot link two non-identifying attribute proofs together, and cannot determine which private key of which user is used in the proof (but note that the issuer's public key must be known to the verifier).

The service provider is the party that demands a proof of one (or multiple) attributes. We call this party the *verifier*: it verifies that attributes are valid. It is important that a verifier does not request more attributes than needed to fulfil a required transaction. This restriction could be achieved by both technical and organisational measures. We will not take this issue into account in our research.

Recall that a smartphone can act as a secure token containing attributes and that a verifier demands a proof of possession of some attributes. To achieve such a proof, interaction between the two parties is needed, and therefore a communication channel is needed. Since both parties normally have access to the Internet, IRMA uses the Internet as communication channel.

If a verifier wants an attribute proof from a user, it shows a QR code that contains a URL with a session identifier to the user. The user scans this code with his/her IRMA smartphone app to connect to the verification server that is linked with the verifier. The established link can be used by the verifier to request attributes from the IRMA smartphone app. The user needs to give consent to reveal the requested attributes. If he/she consents, then the attributes will be sent to the verifier, and the transaction will be logged on the phone. Figure 1.1 shows how this process looks for the user.

1.3 Alliander

Alliander is the parent company of Liander, a grid operator that distributes gas and electricity to large parts of the Netherlands.[5] The main stock holders are regional governments in the Netherlands. One of Alliander's goals is to facilitate the energy transition towards sustainable energy production and smart autonomous power grids.

In this section, we will briefly explain what is meant with the term 'energy transition' and what it means for Alliander. After this, we will look into what Alliander's customers desire. This will give the rationale that we will use to introduce Alliander's new research project, named *Tippiq*.[6]

1.3.1 Energy transition and communication

Fossil fuels are currently the main source for electricity. However, they will eventually deplete, which causes the need for alternative electricity sources. Current alternatives are mostly natural resources like water, wind and sun. These sources are variable and volatile: solar panels, for example, only work when the sun is shining brightly enough.

Also, electricity is more locally generated: customers have their own solar panels on their houses, which will increasingly deliver electricity back to the power grid. This essentially makes each house a small power plant. From a grid operator's perspective, this is hard to manage, because the power grid is not capable of handling a two-way electricity flow. Therefore, huge infrastructural investments are necessary. Furthermore, electricity is hard to store so all the power that is produced must be used instantly. This is a balancing problem that will become more difficult when more volatile power sources are added to the grid.

To avoid the need for upgrading the whole power grid, which requires huge investments, it is possible to invest in a *smart power grid*. A smart power grid uses a two-way information flow, for locally balancing the demand and supply of electricity. This means that there exists a two-way communication flow, which for instance can be used to control devices in occupant's[7] houses. For example, the laundry machine could only run when there is enough electricity available. Another example could be temporarily storing electricity in the battery of a parked electric vehicle. A survey of the most important literature and background information about smart power grids can be found in [2].

The complete transition from fossil fuels to natural resources in combination with a smart power grid is what we call the *energy transition*. Alliander is trying to facilitate this transition with a portfolio of projects and initiatives.[8]

One challenge of a smart power grid is that potentially privacy sensitive informa-

[5] `https://www.alliander.com/en/about-alliander`

[6] `http://www.tippiq.nl`

[7] In this thesis, we frequently use the term occupant. With this term, we mean the people that live in a certain house. We call the person that owns the house (the house owner) the main occupant of a house.

[8] See for a few examples the Dutch article at `http://www.cstories.nl/c-stories/mt500/2015/alliander-opent-duurzame-energiemarkten.427025.lynk`

tion from the customers is needed. For example, a grid operator needs energy usage patterns of customers to balance the production of electricity to the grid in realtime. In an ideal scenario, it is also desirable to control devices of customers: turn energy heavy devices on when the power grid allows it and use the customer's electric vehicles' batteries to store electricity. If a grid operator is going to exert such control or is going to collect this information, then it will interfere with the privacy of its customers.

Smart electricity meters are the first attempt to provide the desired information to the grid operators. Liander was obliged by law to roll out smart meters to every household in The Netherlands. Smart meters contain a data channel between the meter and the grid operator to directly communicate the electricity that is used in a certain period. It is technically possible to send usage updates every 15 minutes, but grid operators and energy suppliers in the Netherlands are currently restricted by law to read out these data at a maximum of once per two months.[9] Electricity usage data are sensitive, since these can for instance be used to detect what users are watching on television, as shown in Greveler et al. [3]. So current smart meters certainly have a privacy problem, and a customer can consequently not be obligated to replace his/her old meter with a smart meter. Internal research of Alliander confirms that many customers rejected a smart meter because of the privacy problems.

Another challenge, besides the privacy laws, is the fact that, in the Netherlands, a grid operator is not allowed to communicate directly with its customers: only the energy supplier is allowed to do that.[10] This means that Alliander has to come up with 'creative' approaches to obtain the desired data in a privacy-friendly way.

Alliander also notices that other companies are struggling with the problem of communication with occupants and the privacy problems that arise there as well. Other companies see the adoption of IoT as well and want to make use of it. These companies certainly desire a privacy-friendly trust platform that can be used to communicate to occupants. This trust platform has to guarantee that privacy sensitive information is handled in the right way.

1.3.2 Customers

Liander's customers mainly live in the Netherlands. As occupants of a house, they are not interested in sharing information about energy. They only want that 'it just works' and they certainly do not feel the need to spend extra time or effort in both communication or disclosing information to a company that they hardly know, like Liander.

However, occupants are interested in sharing information about other (slightly related) themes around their house. For example, topics like home safety and medical care around and in houses are interesting for occupants. They are willing to share privacy sensitive information in these cases, but only if they can retain *control* of this information. Occupants want to be sure that the information will not be shared

[9] See §6.1.4 of `https://autoriteitpersoonsgegevens.nl/sites/default/files/downloads/med/med_20120518-besluit-gedragscode-persoonsgegevens-slimme-meter.pdf`

[10] See the Dutch article about the 'StroomOpwaarts' regulation in the Netherlands: `http://www.edsn.nl/wp-content/uploads/2012/05/MPM-Leveranciersmodel-v6.0.pdf`

with unknown third parties, and they prefer to know which information is shared for which purpose. They also expect something in return: they want to know why they have to share information and what they get in return for it.

An example case that shows how information sharing could be done is the Dutch 112 alert website,[11] which was a pilot project of Tippiq, a project from Alliander that we introduce in the next section. Occupants could give information about their house (like in which room the children sleep and where the flammable substances are stored) to the 112 alert website. 112 alert will share this information with the fire department, but *only* in case of a fire and it will in that case also send an sms to the occupant that it shared information and why it shared this information.

112 alert was just a pilot that ran for a few weeks. It was very popular in those few weeks and it indicated that there was a great demand for such a service. Besides the popularity, it also confirmed that occupants really chose differently when they had the choice in information sharing. Some occupants provided information, but did not choose to share information with the fire department, which apparently means that they trusted 112 alert more than the fire department. The pilot also showed that integrating an extra 'privacy step' in the registration process did not lead to an increased exit rate and that an external privacy protector 'Tippiq', led to additional trust among the users.

The information was still centrally stored at the 112 alert website in this pilot phase. If 112 alert was implemented for real, then this would be changed to a decentralised system, so that 112 alert cannot access the information when it is not needed.

1.3.3 Tippiq

In summary, Alliander wishes to facilitate the energy transition. In order to achieve this goal, the company needs to communicate with its customers. It is difficult for Alliander to do so, because a grid operator is not legally allowed to. Alliander also recognizes that customers are not interested in communication about energy (it is a low interest product). Nevertheless, Alliander also sees that there is a need for a privacy-friendly and trusted communication platform: both customers and commercial partners demand this. Customers are in general not aware of the fact that a system is needed that can balance energy locally. Customers are also not aware that access to their devices is needed in order to do so. Otherwise, energy will become more expensive.

On the other hand, we see that it is getting increasingly important to handle customer data with care. Alliander is very anxious and only wants to handle data when they have explicit permission of the data owner, which is the customer. Therefore, they are working to put a platform in place where data permissions can be set and managed for all kind of data. For this reason, Alliander started the Tippiq project.

Tippiq is a project that will provide occupants insights in what is happening in their residential area. This will be done in two ways. The first way is by broadcasting information *to* occupants. This can for instance be messages about when the garbage in the street will be picked up or messages about power outages in the city. All these messages are different for each physical house address. In order to provide this in-

[11] `https://www.112alert.nu`

formation, Tippiq needs to digitalise the house addresses of its users. Using Tippiq, an occupant will in the future be able to claim this address in order to manage the dataflows around it.

The second way is gathering information *from* occupants in a privacy friendly way. Occupants should be able to choose what they want to share. For example, the website thuisbezorgd.nl requires the house address where it needs to deliver an order and the fire brigade would be more efficient if it has access to the floor plan of a house in case of a fire. The user should have the most important role, as he/she should get complete control over his/her data, and over which party is allowed access to which data. He/she should also be able to see when information is shared and for which purpose. This system has to be a decentralised system, where Tippiq could act as the trusted party and the supplier for the needed devices and software.

1.4 Internet of things

IoT is upcoming. Current studies are looking into *how* IoT can be utilised. See Bhuvaneswari [4] for an overview of current IoT technologies, possibilities and some examples of IoT.

IoT embodies the idea that many devices in a house will be connected with both each other and the Internet. These devices contain not only sensors that collect information, but also actuators that send out signals or control other devices in a house. Examples are an electronic lock on a door, an electronic thermostat and even a laundry machine. A laundry machine could for instance send a signal to its owner when the laundry is done. These examples indicate that IoT can be used for both access and information control in a house.

The sensors in IoT devices in our houses can collect a lot of privacy sensitive data. It would be preferable if house owners have complete control over these data. In this thesis, we propose the use of a 'central house gateway' that controls all the data that flows in and out of a house. This gateway could then also control access to devices in a house: if someone wants to open or control an electronic door lock, then the gateway has to allow this action.

The gateway acts as the central part of a house and could be seen as a device that permits external parties to communicate with (devices in) a house. It also helps occupants by giving them an easy way to control all their IoT devices. We can therefore compare the gateway to a device that takes care of a house, which is almost the same as the definition of a concierge.[12] We will from now on call the house gateway the 'concierge' of a house.

For external parties, it can sometimes be desirable that a house can identify itself in the same way as a person can. To allow identification of a house, we could provide the concierge with an identity. To allow authentication of a house to a service provider, the concierge also needs to be able to prove its identity.

Recall that a person's identity can be built using attributes. This allows a person to authenticate to service providers in a privacy-friendly way. We can also apply this principle to houses and divide the identity of a house into different attributes. A

[12] `http://dictionary.cambridge.org/dictionary/english/concierge`

house can for instance contain an *address* attribute, which can be used to prove the location of a house. The data that was stored by 112 alert could also be stored in an attribute. Another example are certification marks for a house, for instance an energy label or a certification mark on the boiler of a house (to prove that it is maintained correctly). Attributes can also be used to couple occupants to their houses. If we issue the same attribute to both the occupant of a house and the concierge of a house, then an occupant can prove that he/she lives in a certain house.

One of Tippiq's goals is to give occupants control over their data and IoT devices, which can be done by only disclosing the attributes of a house that are really needed. If thuisbezorgd.nl requires the delivery address of a house or if the fire brigade requires the floor plan of a house, then only the corresponding attributes could be revealed. We introduced IRMA as an attribute-based identification system. We can provide a house with IRMA attributes and we can issue these attributes to the concierge to build up an identity for a house. Since we are using IRMA as identity technology, we call the concierge the 'IRMA concierge'.

Tippiq is still examining how it can give occupants control in a reliable and secure way. One way do achieve this goal is by building the central house gateway that can control the dataflows and devices in the house. It should also be able to authenticate to service providers that desire access to the devices, which can be done by disclosing attributes. IRMA is a secure and privacy friendly system that uses attribute-based identification and authentication. In this thesis, we investigate how IRMA can be integrated into Tippiq.We implemented a prototype using IRMA that can be used by Tippiq as a way to reach their goals: providing information to occupants and give occupants control over their data and IoT devices.

Chapter 2

Research goals

In our research, we investigate how we can adapt IRMA technology to make it useful in an IoT and house context. We issue IRMA attributes to a house, to provide it with its own IRMA identity. With this IRMA identity, a house can reveal attributes to service providers. This can be useful in different scenarios.

Occupants should have maximal control over the dataflows around their houses, and they should be able to select which attributes can be revealed to which service provider. Besides control over dataflows, we also want to provide occupants control over their IoT devices. We investigate how IRMA needs to be adapted to provide this control.

2.1 Research question

In our research, regarding the combination of the IRMA technology and the goals of Tippiq, we will answer the following research question:

> How can we use or adapt the IRMA technology in an IoT/house context, where a house has its own IRMA identity and occupants have maximal control over access and authorisation of their IoT devices and data?

2.2 Method & Sub questions

We investigate which IRMA roles are relevant in an IoT context, because IRMA in houses differs from the normal situations. In 'classic' IRMA, we have on one side users with an IRMA token containing credentials and on the other side issuers and verifiers, as discussed in Section 1.2. When we look in the IoT context, we have devices which provide services. We regard a house as an IoT identity that can both provide a service to its occupants and external parties, as well as identify itself to other devices, occupants and external parties. Maybe, it would be even possible to let the IRMA concierge issue credentials to other parties, making each house an issuer and identity provider. All of this means at least that the separation between issuers and verifiers

is becoming a little bit ambiguous.

To solve this ambiguity, we have to specify which IRMA roles and attributes are relevant in an IoT context, which leads to the first sub question:

1. Which IRMA roles are relevant in an IoT context?

After the IRMA roles have been defined, we analyse what is needed for the required adaption of IRMA to houses. We investigate how IRMA would work in some typical IoT applications by constructing protocols for some example scenarios using Message Sequence Charts, which answers the second sub question:

2. What are relevant scenarios, and how can we apply IRMA technology to them?

The Tippiq project wishes to implement a generic rule system for providing authorisation to the data that, for instance, could be provided by IoT devices in a house. We can use IRMA attributes to couple an occupant to a house. We can also use these attributes to preserve integrity and authenticity of Tippiq's rule system system. Occupants can in this case use their IRMA attributes to digitally sign a rule, when they add it to the system. This is a form of attribute-based signatures using IRMA. Chapter 3 explains what IRMA signatures are and Chapter 5 demonstrates an example implementation. This leads to the third sub question:

3. How can we implement and use IRMA signatures to preserve integrity and authenticity of Tippiq's rule system?

At this point, we introduced many different components that, when combined, could work as a complete IRMA solution in an IoT context. However, we still have not proved that this is a working solution. Therefore, we have to integrate all the concepts into a practical service that acts as a proof of concept for our IRMA solution. This proof of concept is used to answer the fourth sub question:

4. How can we integrate the concepts into a practical service and, if possible, the Tippiq project?

After we developed an example implementation, we evaluate it and analyse how a house as identity that is built using attributes is used. Is it really necessary to provide a house with an identity, or could we achieve the same without it? This analysis is completed by answering the final sub question:

5. What are the implications of perceiving a house as an identity with attributes?

2.3 Outline

In the next chapter, we will provide some background knowledge about IRMA and introduce a concept called IRMA signatures. After this, we will introduce some scenarios in Chapter 4, which are used to answer the first three research questions. We will also explain Tippiq's rule system in this chapter. Chapter 5 presents our implementation of IRMA signatures that we use to provide integrity and authenticity for Tippiq's rule system, which answers the question on how we need to adapt IRMA to make it usable with Tippiq's rule system. After this implementation, we will, in Chapter 6, demonstrate an example application that has all the components integrated into one system. This presents the integration of the different concepts into a practical service and will answer the fourth sub question. Subsequently, we will evaluate the system and look at the implications of perceiving a house as an identity, which answers the last sub question. We summarise our findings and conclude in Chapter 7.

Chapter 3

Introduction to IRMA signatures

In this chapter, we introduce an extension of the IRMA system, namely IRMA as an attribute-based signature system, a concept that is introduced in Hampiholi et al. [5]. To understand this use case of IRMA, we first introduce some background concepts that are used within IRMA and the underlying Idemix technology. After this introduction, we will explain what digital signatures are and how IRMA can be used as an attribute-based signature system. In this chapter, we limit us to the theoretical background. We will discuss the practical details and our implementation of IRMA signatures in Chapter 5.

3.1 Zero-knowledge proof

Possession of IRMA attributes is proved using a zero-knowledge proof. With a zero-knowledge proof, a prover can prove knowledge of a secret to a verifier, without revealing this secret to this verifier. However, the verifier will be convinced by the proof that the prover knows the secret. We will now give an introduction about how zero-knowledge proofs are mathematically constructed.[1]

3.1.1 Schnorr's zero-knowledge protocol

An example of a protocol that makes use of a zero-knowledge proof is Schnorr's zero-knowledge protocol [6], which is similar to IRMA's zero-knowledge protocol. For this protocol, we use a cyclic group G, that is generated by a generator $g \in G$ with a prime order n. A value x is randomly chosen from $\mathbb{Z}_n$ (also notated as $x \in_R \mathbb{Z}_n$) as the private key of the prover. After this x has been chosen, the public key h is defined as $h = g^x$. It is assumed that both the prover and the verifier are already aware of the values g, n and h. Only the prover knows x. It is computationally infeasible to

[1] Our introduction to zero-knowledge proofs is largely based on lecture notes of Berry Schoenmakers, which are available here: `http://www.win.tue.nl/~berry/2DMI00/LectureNotes.pdf`

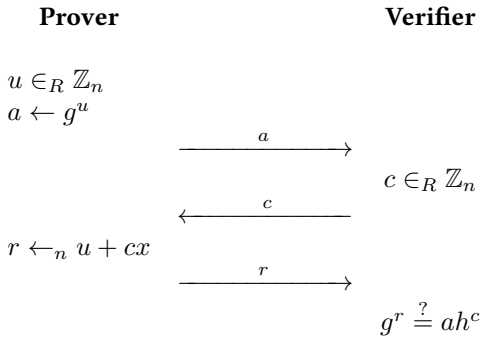


Figure 3.1: Schnorr's zero-knowledge protocol

calculate x, given only h, g and n: this problem is also known as the *Discrete Log problem*.

When the prover needs to prove his/her knowledge of x to a verifier, he/she chooses $u \in_R \mathbb{Z}_n$, calculates an announcement $a \leftarrow g^u$, and sends this a to the verifier as a commitment. After receiving a, the verifier calculates a challenge $c \in_R \mathbb{Z}_n$ and sends this to the prover. The prover will now calculate $r \leftarrow_n u + cx$ and sends r back to the verifier, see Figure 3.1 for a graphical representation of this protocol.

The verifier can use r to check if the prover has responded correctly to the challenge, by checking if $g^r = ah^c$. This check is valid, because:

$$g^r = g^{u+cx} = g^u(g^x)^c \equiv ah^c$$

The verifier must be convinced by the prover that the prover actually knows the secret value x. This is called the soundness property in zero-knowledge proofs.[2] To show this property, we have to prove that a malicious prover (who does not know secret x) is not able to respond to a challenge c with a non-negligible probability after he/she has committed to an announcement a. To prove this, we first determine how we can extract the secret x (also called the *witness*) using two different runs of the protocol with the same announcement a. Given two runs of the protocol with two different challenges c and c' ($c \neq c'$), we obtain two conversations $(a; c; r)$ and $(a; c'; r')$. This means that $g^r = ah^c$ and $g^{r'} = ah^{c'}$, hence $g^{\frac{r-r'}{c-c'}} = h = g^x$. We can now extract x as $x = \frac{r-r'}{c-c'}$. From this extraction, we can conclude that for every announcement a that a (potentially cheating) prover can send to the verifier, there is at most one possible challenge that he/she can answer correctly without knowing x. If there are two possible challenges that can be answered correctly with the same a, the witness and private key x can be extracted from the two conversations, which means that the prover has to know x. This means that if the challenge c is chosen from a large enough set (in this case $\mathbb{Z}_n$), the probability that a prover can answer correctly to this challenge without knowing x is negligible.

We still need to prove why this protocol is zero-knowledge. Zero-knowledge means in this case that the verifier cannot obtain any information from the prover, besides

[2]See §1.2 of `https://cs.nyu.edu/courses/spring07/G22.3220-001/lec3.pdf` for more information on the soundness property of zero-knowledge proofs.

that the prover knows secret x. If this protocol is run multiple times, then the verifier obtains multiple conversations (a, c, r). But the verifier can also generate these conversations him/herself, by using the public key h and choosing c and r randomly from $\mathbb{Z}_n$ ($c, r \in_R \mathbb{Z}_n$). Since we know $g^r = ah^c$, the verifier can calculate a using r and c:

$$a = g^r h^{-c}$$

In this way, the verifier can (without the prover) obtain a valid conversation (a, c, r). Such generated conversations are also called *simulated* conversations. Simulated conversations cannot be distinguished from real conversations, because the probability distributions of simulated and real conversations are identical, since both conversations use two random values from $\mathbb{Z}_n$:

$$\begin{array}{ll} \text{Real:} & \{(a; c; r) : u, c, \in_R \mathbb{Z}_n; a \leftarrow g^u; r \leftarrow_n u + cx\}, \\ \text{Simulated:} & \{(a; c; r) : c, r \in_R \mathbb{Z}_n; a \leftarrow g^r h^{-c}\} \end{array}$$

However, it is important to notice that the challenge c *must* be randomly chosen from $\mathbb{Z}_n$. Since the verifier chooses this c, he/she must be honest in choosing a random c. Therefore, the Schnorr protocol is only zero knowledge if it is used with honest verifiers.

3.1.2 Fiat-Shamir heuristic

Schnorr's zero-knowledge protocol requires interaction between the prover and verifier, which is sometimes not desired. Interaction can be removed by applying the Fiat-Shamir heuristic [7] to the protocol, which converts Schnorr's zero-knowledge protocol into a non-interactive protocol.

To calculate a non-interactive proof, a prover has to obtain a challenge c that is normally provided by the verifier. By applying the Fiat-Shamir heuristic, the prover generates the challenge c by calculating $c \leftarrow H(a)$, where H is a secure one-way cryptographic hash function and $a \leftarrow g^u; u \in_R \mathbb{Z}_n$. The hash function ensures that the prover commits to a. If the prover first calculates c and r, he/she cannot reconstruct a such that $c = H(a)$ holds (this would break the pre-image resistance property of the hash function). The prover can now calculate r in the same way as in the non-interactive protocol. After calculation, the prover sends the pair (c, r) to the verifier.

Since $a = g^r h^{-c}$, the verifier can verify the output of the prover by recomputing a and checking if the equation $c = H(a)$ holds. If this holds, the verifier accepts the proof. It is important to notice that this protocol is *not* zero-knowledge, since there is no way to simulate a non-interactive proof. Every non-interactive proof is always generated by someone that knows the secret x.

While a prover is able to create valid proofs by using $c \leftarrow H(a)$, this does not guarantee freshness. If an attacker obtains a pair (c, r), he/she can reuse it in another session. To solve this problem, a nonce n can be added to the hash function. This changes the challenge to $c \leftarrow H(a, n)$. The nonce can be provided by the verifier to ensure freshness.

As the nonce n can be any arbitrary message, it can also be replaced with a message M. As the pair (c, r) that contains M can only be generated by a prover knowing x,

it can be said that (c, r) can act as a *digital signature* on M. With a digital signature, a verifier can verify that only the prover can calculate this signature. This can be used to ensure the integrity of M. Using this property, the Fiat-Shamir heuristic can be converted into a *digital signature scheme*. We introduce digital signatures and digital signature schemes in Section 3.3.

3.2 Zero-knowledge proofs in IRMA

The process of disclosing some attributes from a secure IRMA token to a verifier is called *selective disclosure*. In this case, the IRMA token proves knowledge of a secret (in the form of a credential) that is issued to the token by an issuer. This zero-knowledge proof convinces the verifier that a certain credential is valid and issued by the right issuer.

Instead of the secret x, as part of the public key $h = g^x$ (as used in Schnorr), Idemix uses credentials containing attributes. These credentials are *Camenisch-Lysyanskaya signatures* (CL-signatures) [8]. They are provided by the issuer in the issuing phase. CL-signatures allow a prover to prove in zero-knowledge that he/she possesses a valid credential. Using proofs, only some selected attributes are revealed, while other attributes in the same credential remain hidden. The verifier will be convinced by the proof that the values of the revealed attributes are the same as the values that the issuer issued to the IRMA token. The verifier will also be convinced that the non-revealed attributes are present.

The IRMA system uses the Idemix zero-knowledge disclosure protocol, which uses the non-interactive version of Schnorr's protocol as a basis. See Section 6.2 of the Idemix Specification [1] for a detailed description of the Idemix disclosure protocol. At a very high level, the verifier sends a nonce n, the user generates a challenge c using n and responds with a non-interactive proof of knowledge on that challenge, which proves the possession of the required attributes. To provide freshness, n must be a fresh nonce.

3.3 Digital signatures

A digital signature scheme is a cryptographic scheme that is used to preserve the integrity and authenticity of a digital message or document. A signature is created with asymmetric cryptography and is constructed using someone's private key (only known to that person) when signing a message. The public key is used to verify signed messages. A signature guarantees that the content of a document is preserved in the way it was meant by the signer at creation time. Only the signer knows the private key and can therefore not deny that he/she signed the document, which provides a property that is called non-repudiation.

Digital signatures can, for instance, be used to sign an online bank transaction. In this example, the signature is used to preserve the integrity and authenticity of the transaction, meaning that no one else except the signer can alter the transaction. Because only the signer is able to create the signature, it also provides non-repudiation, which means that the signer cannot deny that he/she has performed the transaction. Other

use cases of digital signatures are signing an e-mail, or signing an arbitrary file where a file could for example be a PDF document containing a contract.

A verifier of a signature needs the public key of the signer in order to check whether a signature is valid. This requires a complex infrastructure, because it is hard to make sure that a certain public key belongs to a certain person or entity.

To formalize the way a signature is created and verified, a *digital signature scheme* is used. A digital signature scheme consists of three algorithms, namely a *key generation* algorithm, a *signature generation* algorithm and a *signature verification* algorithm. The *key generation* algorithm generates the private and public key of the signer. The *signature generation* algorithm generates a signature using the private key and the *signature verification* algorithm describes how a signature can be verified using the signer's public key.

3.4 Attribute-based signatures

A keypair of a signer normally belongs to a single person, which means that a person can be identified using his/her public key and that all signed messages are linked and traceable to that single person. While this is sometimes desirable, it can also be a privacy problem. A possible example is renting a house via AirBnB. Here, it is desirable to sign a renting contract (with an obligation to pay), but it is not needed to reveal a complete identity. It would be enough to sign the contract with, for instance, only a name or bank account attribute. AirBnB can use this information in case of damage: it can use the bank account attribute to retrieve money for damage recovery and it can use the name to blacklist this person. Another example could be a medical statement that is signed by a doctor. In this example, the doctor can be seen as a person in a certain role. In a medical statement, the role is more important than the person itself. It is important to notice that, these cases, non-repudiation is not completely provided because the signer as person remains anonymous. The only non-repudiation that is provided in this case, is the fact that a person with certain attributes has signed the statement.

Considering the examples from the previous paragraph, it seems logical to focus on a person's *role* rather than a person's identity. We could use a different signing key for every role, but this makes a public key infrastructure even more difficult to manage. A better solution would be to use attribute-based signatures, that are introduced in [9]. With attribute-based signatures, we use attributes to sign a message.

We have already seen IRMA as an attribute-based identification and authentication system. Within an IRMA proof, a challenge is sent by the verifier to the user, where the user gives a signature as response that contains the required disclosed attributes. This response allows the verifier to check that the user is in possession of the required attributes, as explained in Section 3.2

Instead of using a random nonce as a challenge for an IRMA proof, we could also send the hash of a message. By sending the hash of a message, we let the user sign a message hash instead of a random challenge, which can be used to create a signature using IRMA attributes. If we do this, only a few alternations are needed to the IRMA protocol, which means that it is probably easy to implement this in the current IRMA

ecosystem. The idea of using IRMA for attribute-based signatures has been proposed in [5]. From now on, we will refer to *IRMA signatures* if we use IRMA to create an attribute-based signature.

3.4.1 The IRMA signature scheme

To create a signature scheme for IRMA signatures, it is needed to implement the three algorithms that together form a signature scheme. The authors of [5] proposed to expand the *key generation* algorithm with an *attribute issuance* algorithm. *Key generation* is done during initialisation of the IRMA token. At that moment, a private key is generated that is used to bind all the credentials (that will be issued later) to the IRMA token. In the *attribute issuance* algorithm, new credentials will be issued to the token. Issuers sign these credentials with their private key, where verifiers use the corresponding public key to verify the credentials during the disclosure protocol.

In the *signature generation* algorithm, the hash of a message is used as input for the algorithm. This hash is used instead of the challenge that an IRMA verifier would send to an IRMA user in a normal IRMA disclosure proof. Furthermore, the required attributes are selected from the user's token. These attributes and the hash are used on the token to create an IRMA disclosure proof. This IRMA disclosure proof will be a non-interactive proof on hash of the message. This disclosure proof is the main part of the IRMA signature. Other parts of the IRMA signature are used to check if the IRMA signature is created in compliance with its specification. These parts are introduced in Section 5.6.

For the *signature verification* algorithm, the issuer's public key is needed to verify the disclosure proof in the IRMA signature. The verifier calculates the hash of the message and uses the public key and some system parameters to verify this disclosure proof. The verifier also needs to check if the other parts of the IRMA signature are valid. This is done by checking the signature specification, which we introduce in Section 5.2.3.

3.4.2 Diversification between signatures and disclosure proofs

It is desirable to support both signatures and 'normal' disclosure proofs, where the normal disclosure proofs can still be used for authentication. In this case, it is essential to separate the signature domain from the disclosure proof domain. It must not be possible that a hash of a message can be used in a disclosure proof because that means that the user is signing something while he/she thinks that he/she is just authenticating to a verifier. The authors of [5] proposed to append a *Dbit* in front of the challenge. This bit is set to *one* in case of a signature, and to *zero* in other cases. A verifier can use this bit in the reconstruction of the challenge to check if the proof has to be verified as a signature or as a normal disclosure proof.

However, we can improve on the proposed domain separation technique by using the fact that, in the current IRMA protocol, the challenge that is sent to the user is *ASN.1* encoded. *ASN.1* describes a unique representation of the data that preserves type information, which can be used to differentiate between the normal disclosure proof and signature challenges. We can use different data types for signatures and disclosure

proofs. In Section 5.2.4, we describe how we exactly implemented the diversification between the two types of challenges.

3.4.3 Pre-selected vs self-selected attributes

In a normal IRMA disclosure proof protocol, the service provider, which acts as the verifier, determines which attributes are revealed by a user. After the verifier has determined which attributes are needed, he/she sends a request to the user containing a challenge and the attributes that are required. The user sends a response, which the verifier will check. If this response is valid, the verifier will provide the user access to a service or something else that the user has requested.

For signatures, this process could be the same. A party that requests a signature from the user can be seen as a *signature requester* and could send a request to the user containing the required attributes and a message that the user has to sign. We call this request a *signature specification*. The user will send the signature back to the signature requester, which will verify it, use it, and/or store it.

If a signature is stored, it can be verified later by a *signature verifier*. This signature verifier is different from the usual IRMA disclosure proof verifier, because it does not have to interact with the user (and send, for instance, a challenge to him/her). It only has to check if an IRMA signature is valid by checking if the hash of the message corresponds to the used challenge and if the signature is signed using the required attributes that are specified in the corresponding signature specification. Another difference between an IRMA disclosure proof and an IRMA signature is the fact that an IRMA signature can be verified by multiple signature verifiers, where an IRMA disclosure proof will usually only be verified by one verifier.

The signature specifications that are sent by a signature requester already contain a fixed set of attributes that the user has to reveal and include. The signature requester can also enforce certain values for attributes (for instance *over18* must be *yes*). If the user refuses to reveal or does not possess one of the attributes that is requested, then he/she cannot create a valid signature that the service provider, or any other signature verifier that uses the same signature specification, would accept. This means that the attributes of the signature are fixed.

However, signatures could also be created *without* a signature requester. If a user would like to sign a message on his/her own initiative, then this must be possible without requiring another party that sends a signature request to him/her. For this use case, the user must be able to create such requests him/herself.

If a user creates a signature request him/herself, then he/she is able to choose every attribute that he/she possesses. This means that we can distinguish two types of IRMA signatures, namely signatures with *pre-selected* attributes and signatures with *self-selected* attributes. A signature verifier will probably only verify signatures that adhere to a known signature specification, which means it will only accept signatures with *pre-selected* attributes. For example, a verifier of a signed renting contract would require that this contract is signed with certain pre-selected attributes like the full name of the signer.

Chapter 4

Example use cases

In this chapter, we will elaborate on some example use cases. We make use of the IRMA concierge as the central house identity. It can control the dataflows (from for instance IoT devices) that go in and out a house. The IRMA concierge also regulates access to the house itself. Access can either be physical (through for instance a door with an electronic lock) or digital to data and devices. We introduce some example third parties that desire access to IoT devices and data in the house. We also explain the house rules and IRMA credentials that can be used in typical IoT scenarios. In the next section, we start by introducing some general components. After this introduction, we elaborate on several scenarios in the next sections. We did not implement any of the scenarios that are mentioned in this chapter. Instead, see Chapter 6 for a complete implementation of a proof of concept that is based on the ideas and scenarios that we present in this chapter.

4.1 Components for the scenarios

First, we introduce the components which are used to explain some scenarios. Since we are using Tippiq's generic ruling concept in the scenarios, we start with a description of this concept. After this description, we provide an overview of the IRMA credentials that we use in the scenarios. Once the IRMA credentials are introduced, we describe the notation we use to describe the protocols in the scenarios with Message Sequence Charts. We also show how we apply IRMA signatures that we introduced in the previous chapter.

4.1.1 Tippiq House Rules

Tippiq aims to use a generic ruling concept that we call *house rules*. A house rule consists of five main parts and the generic form is constructed in the following way:

In my house holds:

```
[actor] is allowed to do [action] with [actee]
if [condition] to achieve [goal].
```

A house rule is meant to offer an easy to configure, yet powerful language for occupants to control the dataflows of IoT devices in their houses and to control the IoT devices themselves. This means that house rules can also be used to regulate access to the house. As we have shown in the generic form, house rules are constructed using an `actor`, `action`, `actee`, `condition` and `goal`.

The `actor` determines the *who* or *what* that is allowed to do an action and the `action` determines *what* the actor is allowed to do. An action is normally done with another actor, which is called the `actee`. Actions can be restricted to only be allowed if a certain condition is met, which we specify with the `condition`.

A house rule is commonly created for a fixed goal. This `goal` describes the intention of the rule. This intention is usually not checked when evaluating a rule, since it is unknown at that moment. If a rule for instance describes access to a house, then it will not be known what the person's intentions are if he/she accesses the house. However, the goal can be useful for auditing afterwards. We can, for instance, log all the applications of the house rules and check afterwards if the corresponding actions of the house rule that is applied are really taken. An example is a house rule that allows a person access to the house for the goal 'watering the plants'. If this person did other things in the house as well, then this action will not match the goal of the house rule anymore.

We can use house rules to define concrete IoT scenarios. We can, for instance, construct a house rule for the 112 alert case that we described in the introduction. This house rule will allow firefights to retreive information about the occupant's house if it is on fire:

```
The firefighters are allowed to retrieve information
about my house and the occupants of my house if my
house is on fire to better prepare their rescue actions.
```

We can split this rule into the five parts that we discussed earlier:

`actor`	The firefighters
`action`	are allowed to retrieve
`actee`	information about my house and the occupants of my house
`condition`	if my house is on fire
`goal`	to better prepare their rescue actions.

4.1.2 Basic IRMA credentials

Recall that we have defined both the occupants of a house and the house itself as entities that have their own identity. Therefore, we will provide them both with their own relevant credentials and attributes. Table 4.1 shows an overview of the used credentials and attributes that we issue to them. Note that we call the IRMA identity 'house' the *IRMA Concierge*, because this is the device that both contains and verifies attributes that control access in a house to IoT devices and other parties.

We start by giving both the house and its occupants a credential. The house obtains a *CHouse* credential, containing an attribute with a unique and identifiable number called the *houseID*. We also issue this *houseID* attribute to each occupant in an *OHouse*

IRMA identity	Credential	Attributes
Concierge	CHouse	houseID
		address
Occupant	Person	personID
		name
	OHouse	houseID
		role

Table 4.1: The basic IRMA credentials

credential, which couples the occupants to the house they live in. We should emphasise that the two credentials are different, which explains why we call the credential of the concierge *CHouse* and the credential of the occupants *OHouse*. We assume that the *houseID* attribute can be used to look up a house, which means that a party can connect and send data to a house once it knows the value of the *houseID* attribute of that house.

A house has an address. It could be very useful to prove the validity of this address to external parties, for instance to authenticate the house address to a post service. Therefore, we also provide the house with an *address* attribute in the *CHouse* credential. For some scenarios, it could also be useful to add an *address* attribute to the occupant's IRMA token in the *OHouse* credential. However, we do not use this in our scenarios so we decided to not include it.

As we have coupled an occupant to his/her house with the *houseID* attribute, it is now possible for an occupant to prove that he/she lives in a certain house. However, it could be useful to make a separation between the different roles in a house. The house owner could for instance be allowed to sell the house, while a 'normal' occupant is only allowed to access the house or to deliver a package to the house's mailbox. Therefore, we add another attribute to the *CHouse* credential, called the *role* attribute. In our case, this could either be *isOccupant* for 'normal' occupants or *isMainOccupant* for the house owner. This attribute could in the future be expanded with other roles, for instance an *isGuest* role.

Sometimes, it is desirable to uniquely identify a person without linking this identification to the house he/she lives in. To allow this unlinkable identification, we issue a *Person* credential to the occupant's IRMA token. This credential contains the *personID* attribute, which is a unique number. Besides the *personID* attribute, we also add a *name* attribute to this credential.

We realise that the attributes of the *Person* credential overlap with the information stored on a person's identity card. The IRMA team has already proposed a credential that represents the information from a Dutch identity card using IRMA attributes.[1] We could use that credential instead of our *Person* credential. However, we try to keep the protocols as simple as possible, which also means that we only specify attributes that we really use in the scenarios.

The attributes we defined in this section will be used as the basis for the scenarios we describe in Section 4.2. However, for some scenarios, it will be needed to expand on these attributes and issue extra attributes. We will explicitly mention these attributes

[1]See Github for the specification of the Dutch *MijnOverheid* credential: `https://github.com/credentials/irma_configuration/tree/master/MijnOverheid`

in the scenarios when this is the case.

The credentials we introduced (*CHouse*, *OHouse* and *Person*) need to be issued by a trusted party. The *Person* credential will probably be issued by the government, although it needs to be expanded to contain all the attributes that are currently present on an identity card (see for instance the *MijnOverheid* credential of the IRMA project that we discussed earlier).

The *CHouse* and *OHouse* credentials could be issued once you move into a new house. Such movement is usually registered at the notary which could be a logical location to issue house credentials. We could in that case also make each notary an issuer of these credentials. Another solution could be making Tippiq an issuer of some credentials, which especially in the beginning might be a viable alternative.

We also considered making each house its own issuer. If we would do this, then each house could issue its 'own' *OHouse* credential. However, this would make scheme management much harder (we discuss the IRMA scheme manager in Section 5.1.2). Each participant of the IRMA protocol should have access to the IRMA scheme that contains the credential descriptions and public keys of the issuers. If each house would be an issuer, then each house must have a public key in the central IRMA scheme. Maybe it would be possible to decentralise the IRMA scheme in the future, which would make scheme management easier.

Another issue of making each house an issuer is about trust management. If each house could issue its own *OHouse* credential, then a house can issue this credential to anyone it wants. This allows a house to add a 'house owner' credential to a person that does not live in the house. If we centralise the *OHouse* credential at a trusted party, such as the notary, then only this party can issue this credential. This party will only issue after the occupant is sufficiently checked by, for instance, showing an ID document.

4.1.3 Message Sequence Charts

We describe all the scenarios with message sequence charts. A message sequence chart (MSC) contains the flow of the messages sent between different parties. Each arrow in the chart indicates a message, where we added the content of the message above the arrow. Most messages are preceded with a number that corresponds to the number in the description of the scenario.

Arrows that are connected in a grey box denote an established channel. Within this channel, it is not possible to swap messages or IRMA tokens. This prevention is enforced on a lower level in the IRMA software library.

Besides arrows, we also have boxes. These boxes contain actions that do not require communication and must be done by the entity itself.

An arrow with $RV(x_y)$ indicates that the sending party, where the arrow originates from, sends a *disclosure proof request* containing attribute x from credential y to the other party. A disclosure proof request is a request from a verifier to a prover that asks the prover to give a proof of possession of some (in the request) specified attributes. The other party can respond with a *disclosure proof* containing a proof of the attributes that are requested. Such a disclosure proof is denoted with $SD(x_y)$, which refers to

a disclosure proof of attribute x from credential y.

We illustrate issuing by sending credential 'directly' from the issuer to the user. Issuing can only be done with full credentials (and not with single attributes). $I(x)$ therefore means that a party issues credential x to the other party. While we denote issuing with only one arrow from the issuer to the user, this is not the case in reality. There are multiple message sent between the two parties. However, since these details are not relevant for our protocols, we omit those details.

Furthermore, we use IRMA signatures as introduced in Section 3.4.1. An IRMA signature is denoted as $\text{Sig}(\text{data})_x$, where *data* is the data that is signed and x is the set of attributes that is used in the signature. We omit the credential in this case for brevity, but we will always mention it in the corresponding scenario description. As we introduced in Section 3.4.3, signatures can be created with *self-selected* or *pre-selected* attributes. For the *pre-selected* case, a *signature request* must be sent by the signature requester. A signature request is indicated by an arrow with $RS(x_y)$, where a signature with attribute x from credential y is requested. In some cases, a certain value for an attribute is enforced by the signature requester. If this is the case, we will mention this in the corresponding scenario description.

4.1.4 Application of IRMA signatures

As we introduced a notation for IRMA signatures for our MSC, we will use them in our scenarios. We will use IRMA signatures for three different cases:

- In the first case, we use IRMA signature to sign house rules. House rules are used to control access to devices. We need to make sure that house rules can only be added or changed by an authorised occupant. Therefore, we sign each house rule with IRMA attributes from the occupant. IRMA signatures for house rules are created on the occupant's IRMA token and sent to the IRMA concierge. For each house rule, we have to determine which attributes are needed in the signature (a rule for allowing a mailman to deliver a package in the electronic post box can for instance be allowed by every occupant, while a rule to allow someone access to the house can be restricted to main occupants only). After this has been determined, the required attributes for each house rule are fixed, which means these signatures are signatures with *pre-selected* attributes.
- For the second case, IRMA signatures are used to authorise verifiers to request attributes from the IRMA concierge. Verifiers can only retrieve attributes from the IRMA concierge if occupants have authorised them beforehand. This authorisation must be done beforehand, because the IRMA concierge must be able to operate on its own. This means that we cannot show a normal confirmation dialog to the occupant. Therefore, we let the user use his/her IRMA token to authorise these verifier requests by sending an IRMA signature containing this authorisation to the IRMA concierge. These signatures are always signed using the *houseID* attribute from the *OHouse* credential on the occupant's IRMA token. Because this attribute is fixed, these signatures are also signatures with *pre-selected* attributes. After a verifier is authorised by an IRMA signature, it can request the attributes from the IRMA concierge.
- In the third case, we use IRMA signature to sign official contracts and docu-

ments. Service providers require some documents and contracts to be signed by the occupant. In these cases, service providers will send a signature specification to their customers, which means that in this case, again signatures with *pre-selected* attributes are used.

For adding a signed house rule to IRMA concierge, we use the following syntax in the MSC charts of our scenarios (as introduced in Section 4.1.3):

$$\text{Sig}(\text{houseRule})_x$$

In this notation, x is the list of attributes that is included in the signature, and *houseRule* is the rule that will be signed. For brevity, we will in the scenarios only mention the house rule that is signed, without describing the complete interaction between the IRMA concierge and occupant to create a signed house rule. This interaction will always look as follows:

- The occupant's IRMA token sends a house rule that the occupant would like to add/change to the IRMA concierge.
- The IRMA concierge determines which attributes are needed to sign this rule. When the required attributes are determined, the IRMA concierge sends a signature request to the occupant's IRMA token.
- The occupant's IRMA token uses this request to create an IRMA signature, and it asks the occupant for confirmation to sign the house rule. If the occupant agrees, then the IRMA token sends the signed house rule to the IRMA concierge.
- The IRMA concierge adds the house rule and the signature to the database.

These steps are the same with each added/changed house rule, which justifies our decision to omit them and only mention the short signature notation.

For the second case (adding verifier permissions), the same protocol as used for adding house rules can be applied. However, since the included attribute in the signature is fixed in this case, the signature specification could also be created by the IRMA token itself. Considering that in this section, we only propose some scenarios, we will not look into detail in how this will exactly be implemented.

For the third case (official contracts and documents by service providers), the signature specification will be determined by the service provider. For this specification, we use the explicit notation '$RS(x_y)$', as specified in Section 4.1.3.

4.2 Scenario: Ordering at an online shop

Our first scenario describes an occupant who orders a product at an online shop that will be delivered to his/her house by a mailman. This scenario consists of four parties, namely the occupant, the IRMA concierge of the occupant, the mailman and the online shop. All these parties desire validation of some attributes of other parties in order to trust these other parties. We will first examine which *trust relations* are needed between parties. After this, we will define the needed IRMA attributes to achieve these relations. With these attributes defined, we will describe the protocol of the scenario itself.

4.2.1 Trust relations

The occupant has to pay the online shop to receive products that he/she ordered. This payment must be done securely and to the right party, which means that the online shop has to prove its identity to the occupant. We could use IRMA for this proof, which would mean that the online shop has to prove possession of an IRMA attribute to the occupant.

However, IRMA is not needed in this case since there exists already a mechanism that lets the shop prove its identity to its users. This mechanism is called Transport layer security and requires the shop to provide a TLS certificate to its online visitors. The shop has to prove its identity to a certificate authority in order to obtain such a certificate, especially in the case of Extended Validation certificates.[2] When the occupant visits the website of the shop, he/she will see a green bar with the name of the shop in the address bar in his/her browser if the shop provides an EV certificate that is issued by a trusted certificate authority.

It would also be desirable for the occupant if the mailman could authenticate him/herself when delivering the product. A mailman is currently probably trusted if he/she wears a mailman uniform, which seems to be enough in most cases. However, we could add an electronic lock to a mailbox to prevent stealing of delivered mail and packages. In that case, it is desirable that this box can only be opened by an authenticated mailman if this mailman is delivering a package to the corresponding house.

To authenticate the mailman, he/she could be provided with a mailman IRMA attribute and reveal this if he/she delivers a package. The electronic mailbox will in this case only open after a valid proof of a mailman attribute and only if there is a package to be delivered. The mailman can use his/her phone or a physical smart card as an IRMA token. In addition, the mailbox could be equipped with a screen that shows a QR code, which is used to establish a connection between the mailbox and the mailman's IRMA token.

For the online shop, it would be desirable to obtain a proof of the house address of the customer. This would prevent delivery of packages to fake or non-existent addresses. The occupant can prove his/her address to the online shop by authorizing the shop to retrieve the *address* attribute from the concierge. In this case, the shop is the verifier, and the concierge the IRMA user.

In addition, we could let the online shop issue an IRMA attribute to the occupant's IRMA concierge. This attribute could be checked by the mailman when he/she is delivering the package to the mailbox to be sure that the package will be delivered to the right address.

4.2.2 Relevant IRMA attributes

In the previous section, we introduced the trust relations that are desirable. We also suggested some possible solutions to create these relations. In this section, we will show which IRMA attributes we use in the protocol. These attributes are summarised

[2] See for instance the Certificate Practice Statement of DigiCert about the validation they require for an EV certificate: `https://www.digicert.com/docs/cps/DigiCert_EV-CPS.pdf`

IRMA identity	Credential	Attributes
Concierge	CHouse	houseID
		address
	Order	orderID
Occupant	OHouse	houseID
		role
Mailman	MailmanID	mailmanID

Table 4.2: The IRMA credentials and attributes used in the online shop scenario

in table 4.2. In the next section, we will describe the protocol itself and show the MSC chart corresponding to these attributes and this protocol.

The protocol begins with an occupant that buys a product at the online shop. The shop desires a proof of the occupant's address for this purchase. This proof can be provided with the *houseID* attribute on the occupant's IRMA token. The online shop can use this attribute connect to the occupant's IRMA concierge. After the occupant has authorised the shop to retrieve the *address* attribute from the IRMA concierge, this connection can be used to read out this attribute. We will later show how this works in this protocol.

The mailman needs to prove his/her identity to the IRMA concierge for which we will use a *mailmanID* attribute. We choose an identifying attribute, to allow investigation and logging of the mailman for the case that a package is lost. If someone possesses this attribute, then this person can be seen as a valid mailman. This means that if you allow anyone with a *mailmanID* attribute to do an action, you allow *everyone* with the role mailman to do this action. On the other hand, if you, besides only possession, also enforce a specific value for the *mailmanID* attribute, then you allow only a *specific* mailman to do the action.

The occupant has to give permission before the mailman is able to open the mailbox. This will be done using a Tippiq house rule, which is signed with the *houseID* and the *role* attribute from the *OHouse* credential. The value for this *role* attribute can either be *isOccupant* or *isMainOccupant*, since both normal occupants as well as main occupants are allowed to add this house rule. The *houseID* attribute is needed to prove that the occupant is really an occupant of this house.

The mailman wants to check if he/she is delivering the package to the right house. This can be done by letting the shop issue an *orderID* attribute to the IRMA concierge. This identification number will also be physically printed on the package itself, which allows the mailman to do the check that is desirable by verifying this attribute from the IRMA concierge.

The IRMA concierge also wants to check the *orderID* of the package that the mailman delivers, before it allows the mailman to open the mailbox. It would therefore be logical to also issue this attribute to the mailman's IRMA token, which can then be revealed to the IRMA concierge. However, this is hard to achieve, since it is not beforehand known which mailman is going to deliver the package. This means that is not possible to issue this attribute to the right mailman. This is also the case with the *mailmanID* attribute that we discussed, but in that case we solved it by allowing every mailman with a *mailmanID* attribute access to the mailbox. Another issue is that a mailman would need multiple IRMA attributes per credential type (since he/she deliv-

ers multiple packages with multiple *orderID* attributes). However, this would not be an issue since it is recently possible to have multiple instances of the same attribute on an IRMA token.[3].

Therefore, we choose to only print this *orderID* on the package that the mailman will deliver. The mailman still needs to authenticate him/herself using the *mailmanID* attribute and is only allowed to open the post box, if he/she possess a package with a valid *orderID*. We enforce this in the house rule that we will discuss in the next section.

4.2.3 The protocol

As we have introduced both the trust relations and the needed IRMA attributes, we can now propose an example protocol for this scenario. The corresponding MSC can be found in Figure 4.1.

We will describe the global overview of this scenario in a numbered list, where each number refers to a message in the MSC:

1. The occupant orders a product (with a certain *orderID*) at the online shop and pays (using an electronic payment system).
2. The shop requests a disclosure proof of the *houseID* attribute of both the occupant and the IRMA concierge. In this request, it also demands the *address* attribute of the IRMA concierge. The occupant has to allow this request on his/her IRMA token. On this token, the occupant authorises the shop to retrieve the *houseID* attribute from his/her personal IRMA token. He/she also authorises the shop to retrieve the *houseID* and *address* attributes from his/her IRMA concierge. This authorisation can be done in one confirmation dialog on the token. After confirmation, the IRMA token will create an IRMA disclosure proof and an IRMA signature. The IRMA disclosure proof is used to send the *houseID* from his/her personal IRMA token to the shop (step 3 of this scenario). The IRMA signature is sent to the IRMA concierge and is used to authorise the shop to retrieve the *houseID* and *address* attributes from the IRMA concierge (step 4 of this scenario).
3. An IRMA disclosure proof of the *houseID* attribute is sent from the occupant's IRMA token to the online shop. At this moment, the online shop has obtained a validated *houseID* attribute from the occupant. This attribute is needed to connect to the occupant's IRMA concierge. When the shop connects to the IRMA concierge, it will retrieve the *houseID* and *address* attributes (step 5 of this scenario). The shop needs to check if the two *houseID* attributes match, in order to verify that the occupant really is an occupant of the house that the shop is connecting to.
4. The occupant's IRMA token sends an IRMA signature to the IRMA concierge. This signature authorises the online shop to request the *houseID* and *address* attributes from the IRMA concierge. Since this message is used to authorise a verifier (the shop) to retrieve IRMA attributes from the IRMA concierge, this

[3]See this Git commit: `https://github.com/credentials/irma_android_cardemu/commit/bd8aabaa25f50f244bbc71573885b2f26d46ab10`

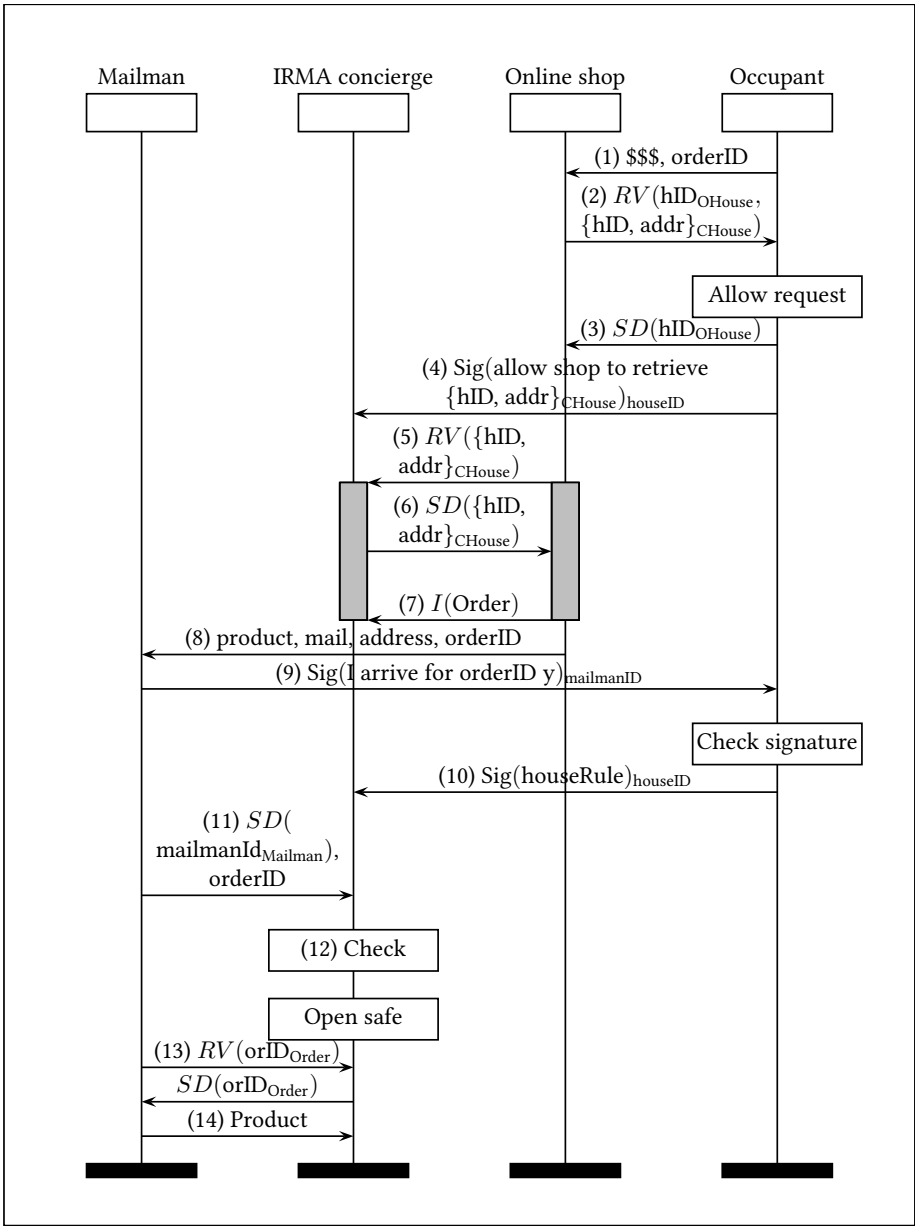


Figure 4.1: MSC: Ordering at an online shop

message is signed with the *houseID* attribute of the occupant (note that only occupants of the corresponding house are able to create such signatures, since only they possess the right *houseID* attribute).

5. The shop requests a disclosure proof of the *houseID* and *address* attribute from the IRMA concierge. We assume that the shop (which is the verifier) is authenticated to the IRMA concierge. This can either be done with TLS, or in the future with IRMA verifier authentication that currently is not implemented (see Section 5.1.5 for details on verifier authentication).

6. The IRMA concierge allows this request (because it has received a signed message from the occupant to allow requests from this shop). It will send a disclosure proof with the requested attributes back.

7. After the verification has succeeded, the shop will issue an *orderID* attribute (in an *Order* credential) to the IRMA concierge.

8. Once the order has completed, the shop will hand it over to the mail service, together with the destination address and the *orderID* printed on the package.

9. The mail service sends a message to the occupant containing the order number of the package. This message is signed with a *mailmanID* attribute from a mailman. This can be a different *mailmanID* attribute than the *mailmainID* attribute of the mailman that delivers the package. In the future, we could add a delivery date to this message. This date could then also be added to the house rule in the next protocol step.

10. The occupant checks the signature of this message, and adds a house rule to the IRMA concierge that allows a valid mailman (validity can be checked by requesting the *mailmanID* attribute) to open the mailbox if this mailman has to deliver the package with the right *orderID*. This rule is signed with the *houseID* attribute of the occupant and consists of the following parts:

`actor`	An authorised mailman
`action`	is allowed to open
`actee`	the mailbox of my house
`condition`	if he/she has a package for my house with *orderID* X
`goal`	to deliver this package in my mailbox.

11. When the mailman arrives at the house to deliver the package, he/she needs to prove to the IRMA concierge that he/she is the right mailman by sending over a disclosure proof of the *mailmanID* attribute. He/she also needs to prove that he/she possess the package with the right *orderID*. We already argued that it is hard to use IRMA attributes for this, so we let the mailman send the *orderID* to the IRMA concierge as a plain number, over the same channel as we send the disclosure proof. The mailman can retrieve the *orderID* by scanning a QR or bar code on the package, which contains this number.

12. The concierge verifies the *mailmanID* attribute and stores it in a log file. It also checks the *orderID* that the mailman provided. If the *mailmanID* attribute is valid, and if the *orderID* matches the *orderID* attributes was issued by the shop, the concierge opens the safe and allows the mailman to verify the *orderID* attribute.

13. The mailman sends a disclosure proof request for the *orderID* attribute, and will get a disclosure proof back from the IRMA concierge.
14. If the verification of this request succeeds, then the mailman will put the package into the mailbox. The IRMA concierge records this action, and will from now on disallow any other mailman that wants to deliver a package with the same *orderID*, since this package is already delivered. A notification to the occupant and the web shop can be sent if the package is delivered.

4.3 Scenario: 112 alert

In Section 1.3.2, we introduced 112 alert as an example and a working prototype that was used to test the need for a platform such as Tippiq. This prototype provided control over data to customers. The 112 alert service only shared the information that is provided by customers to the fire brigade in case of a fire.

One problem with 112 alert is that it is a centralised system. Customers had to provide all their privacy sensitive information to 112 alert, so they had to trust 112 alert completely with their data. It would be more privacy-friendly if these data are stored locally, for instance on the IRMA concierge. While we normally prefer decentralised control and storage, we should emphasise that in the case of a fire, it is very important that the devices that control the data and house access, as well as the devices that store the data must still remain working. This means that these devices need to be fireproof. If it is not possible to achieve this, then a centralised solution could be preferable. In that case, it still needs to be considered whether the advantage of using a centralised service is bigger than the loss of privacy and control.

Another problem is the fact that 112 alert still relies on a central emergency room for the detection of a fire. In other words: it will provide the data to the fire brigade if the emergency room notifies 112 alert of a fire, and this will also happen in the case of a false alarm. A better approach would be to use the sensors in a house to detect if there is a fire.

If the house is equipped with a digital lock that can be controlled from the IRMA concierge, then it might also be useful to automatically grant the firefighters that are sent by the fire brigade access to the house in the case of a fire. When a house is on fire, there is a chance that the IRMA concierge and other IoT devices in the house will stop working. In that case, the fire fights will break the lock on the door, in the same way as is done in a 'non-digital' house.

4.3.1 Trust relations & IRMA attributes

As with the previous scenario, we have an occupant and an IRMA concierge. We also assume that the occupant's house contains some sensors to detect fire. The smoke detectors can notify the IRMA concierge in case of a fire. We assume that this local connection is done securely.

The IRMA concierge notifies the emergency room in case of a fire. It will include the required house data in this notification. To ensure integrity and authenticity, these data need to be signed. We use an IRMA signature with the *houseID* attribute, to

uniquely identify this house. We also include the *address* attribute. This attribute is used by the emergency room to send the fire fighters to the right location.

The emergency room will send firefighters to the house, and provide them with the house data so that they know where for instance the children are sleeping. We assume that the firefighters have their own IRMA tokens that contain a *Firefighter* credential with a *firefighter* attribute. With this credential, they can prove that they are indeed firefighters. We will use this to provide the firefighters access to the house in case of a fire.

4.3.2 Protocol

Before any data can be shared, the occupant has to enter these data into the IRMA concierge. This is the same data as was sent in the 112 alert case, and we will refer to these data as *'house data'*.

After entering the data, the occupant needs to create two house rules. One for allowing the emergency room to retrieve the house data in case of a fire and one for the firefighters to allow them to open the door of the house in case of a fire.

House rule #1:

`actor`	The IRMA concierge
`action`	is allowed to send to the emergency room
`actee`	my house data
`condition`	if my house is on fire
`goal`	so the firefighters can better prepare their actions.

House rule #2:

`actor`	The firefighters
`action`	are allowed to open
`actee`	the door of my house
`condition`	if my house is on fire
`goal`	to extinguish the fire in my house.

We allow every firefighter with a valid *firefighter* attribute to open the door, which means that we only look at the *role* of these persons, instead of looking at *specific* persons. This is the same as with the *mailmanID* as we discussed earlier.

We will use these rules in the protocol that we will describe now. The corresponding MSC can be found in Figure 4.2.

1. The occupant enters the house data in his/her IRMA concierge. These data are signed with the *houseID* attribute, which is used to check if these data correspond to the right house. If we only want to allow main occupants to add house data, then we would also include the *role* attribute, which in that case can be used to verify if an occupant is the main occupant of the house.
2. The IRMA concierge will sign these data with the *houseID* and *address* attribute (as discussed in the previous section) and store the data in its database. The reason that the IRMA concierge (re)signs these data instead of the occupant is because only the IRMA concierge contains the *address* attribute.

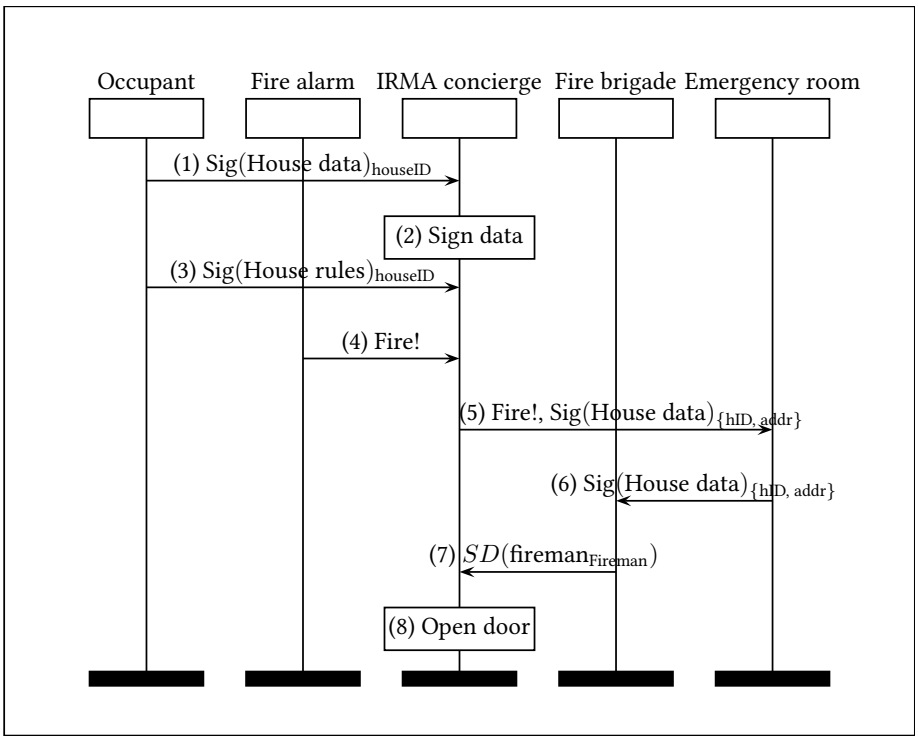


Figure 4.2: MSC: Allow firefighters access to house and house data

3. The occupant enters the two house rules in his/her IRMA concierge. These rules are just as the house data signed with the *houseID* attribute. In this case, we could also include the *role* attribute if we want only main occupants to be able to add house rules about retrieving house data.
4. If there is a fire, then the fire alarm will send a signal to the IRMA concierge.
5. The IRMA concierge notifies the emergency room and includes the house data that the occupant sent to it in message (1) in this notification. Note that these data are signed with the *houseID* and *address* attributes by the concierge. The address attribute is used to send the firefighters to the right and verified location.
6. The emergency room checks the signature, sends the firefighters to the occupant's house, and provides them with the house data it received from the IRMA concierge. If the signature does not verify, then the firefighters will probably still be sent to the house, because it is important to extinguish the fire. However, this is up to the emergency room to decide.
7. Once the firefighters reach the house, they disclose their *firefighter* attribute to the IRMA concierge.
8. After disclosure, the IRMA concierge will open the door.

4.4 Scenario: rent out your house

Sometimes, it can be desirable to grant someone else access to your house. If the house is equipped with an electronic lock, we could use IRMA attributes to allow and revoke access. This allows a house owner to rent his/her house to someone else, without giving away a physical key. For renting a house, a contract could be signed, for which we can use an IRMA signature.

4.4.1 Trust relations & IRMA attributes

In this scenario, we have three entities, namely the main occupant, the renter and the IRMA concierge. We assume the IRMA concierge is connected to an electronic lock, and is able to open this lock.

The occupant and renter need to agree upon a renting contract, where the renter has to prove possession of the *personID* attribute (although 'anonymous' renting would be possible by choosing another non-identifying attribute). The occupant has to prove that he/she is the main occupant of the house that is going to be rented. The occupant can do so by proving possession of the *houseID* and *role* attribute.

4.4.2 Protocol

We will now describe the renting protocol. Figure 4.3 shows the corresponding MSC.

1. The main occupant and the renter agree upon rental of the occupant's house. This will in this case be done 'in person'; we will later show an example where

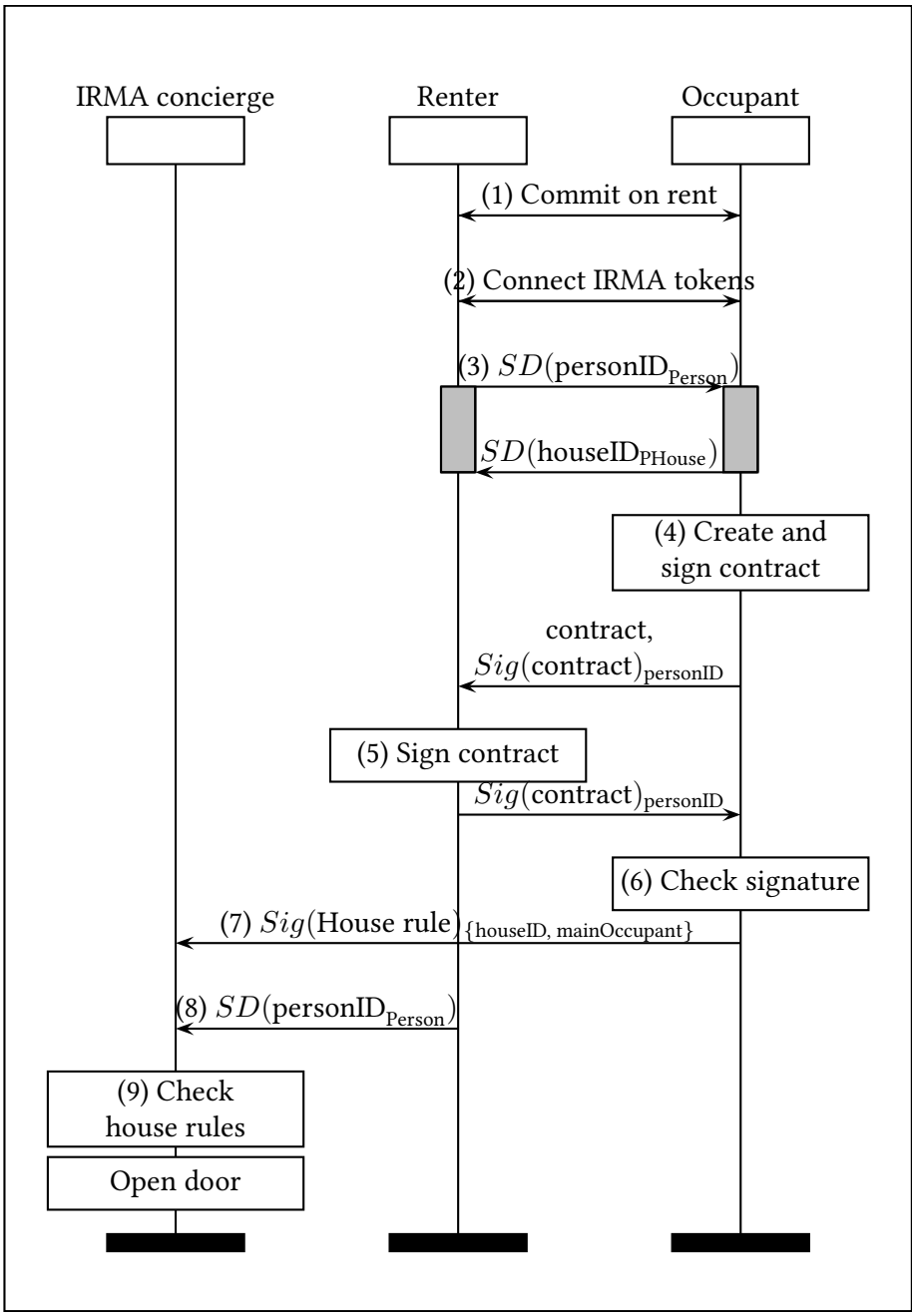


Figure 4.3: MSC: Allow a renter access to your house

this is done digitally.

2. The renter's and the occupant's IRMA tokens need to be connected with each other. In the case that both IRMA tokens are equipped with an NFC reader, this could be done by placing both phones back to back to establish a connection using NFC. Another option is letting one phone scan a QR code on the other phone.

3. Both parties need to disclose an IRMA attribute: the renter reveals his/her *personID* and the occupant reveals his/her *houseID*. In this way, the occupant proves that he/she lives in the house and the renter proves his/her personal identity.

4. The occupant creates a contract which contains the period, the costs and maybe some other conditions. This contract needs to be signed by the renter, which means that he/she must be certain the right contract will be signed. The occupant requests the renter to sign the contract with his/her *personID* attribute, which means that this will be an IRMA signature with *pre-selected* attributes. Signing happens on the IRMA token, which shows a dialog containing the message that will be signed. The message can either be the contract itself, or it can link to the contract in an external file. We show in Section 5.2.3 how this exactly could be achieved. After creation, the occupant signs the contract with his/her *personID* attribute and sends both the contract and the IRMA signature to the renter.

5. The renter verifies the IRMA signature and reads the contract. If the signature is valid and he/she agrees with the contents of the contract, he/she sends it back, signed with his/her *personID* attribute.

6. The occupant verifies the signature on the contract.

7. The occupant adds a house rule to the IRMA concierge that allows the renter to open the door. This will be done by allowing the *personID* attribute of the renter access to the house with the following house rule:

`actor`	Person with *personID* = X
`action`	is allowed to open
`actee`	the door of my house
`condition`	between startDate and endDate
`goal`	to enter my house that he/she rented.

   This rule needs to be signed with the *houseID* attribute and the *role* attribute set to *mainOccupant*.

8. If, after some time, the renter wants to enter the house, then he/she will disclose his/her *personID* attribute to the IRMA concierge.

9. The IRMA concierge checks the corresponding house rules and opens the door if the renter is allowed to access the house, since only main occupants are allowed to give someone else access to the house.

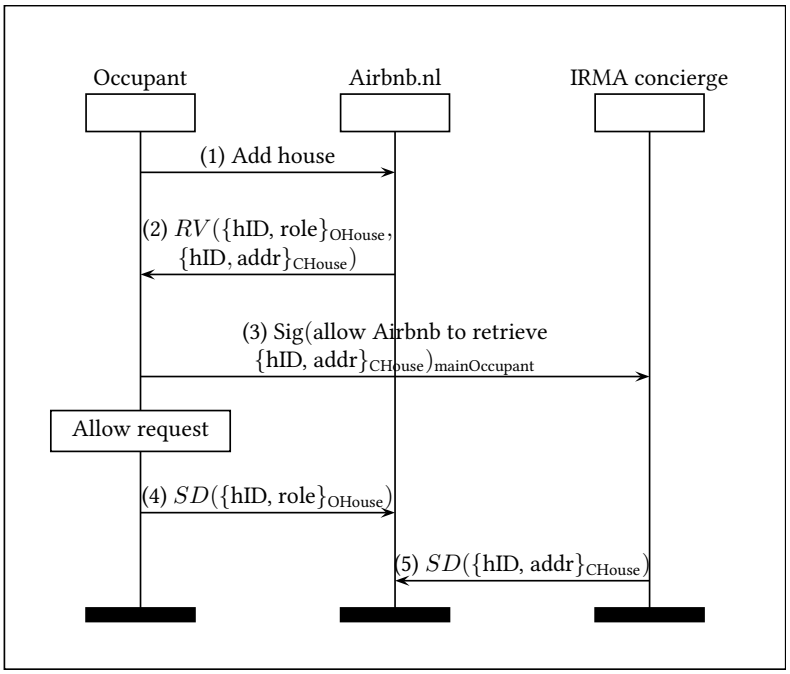


Figure 4.4: MSC: Add a house to Airbnb

4.5 Scenario: Airbnb

The rent a house scenario is limited, because it is required to meet in person to sign the renting contract. In that case you can also physically hand over a key for the house, so the use of an electronic lock does not add much value. We could improve this scenario by integrating it into Airbnb.[4] The Airbnb platform allows its customers to both rent and hire each other's houses, where Airbnb is the party that connects them.

We split this scenario into four parts, to make a clear separation between the distinctive steps:

1. The occupant adds his/her house to Airbnb.
2. The renter finds a house on Airbnb and rents it.
3. The occupants obtains a confirmation for the rent from Airbnb, verifies this confirmation and provides the renter access to his/her house.
4. The renter accesses the house by scanning a QR code on the touchscreen on the door.

The first step is adding a house to Airbnb. The MSC can be found in Figure 4.4.

1. The occupant adds his/her house to Airbnb.
2. Airbnb requests a proof of both the *houseID* and the *role* attributes from the

[4] `https://www.airbnb.nl/`

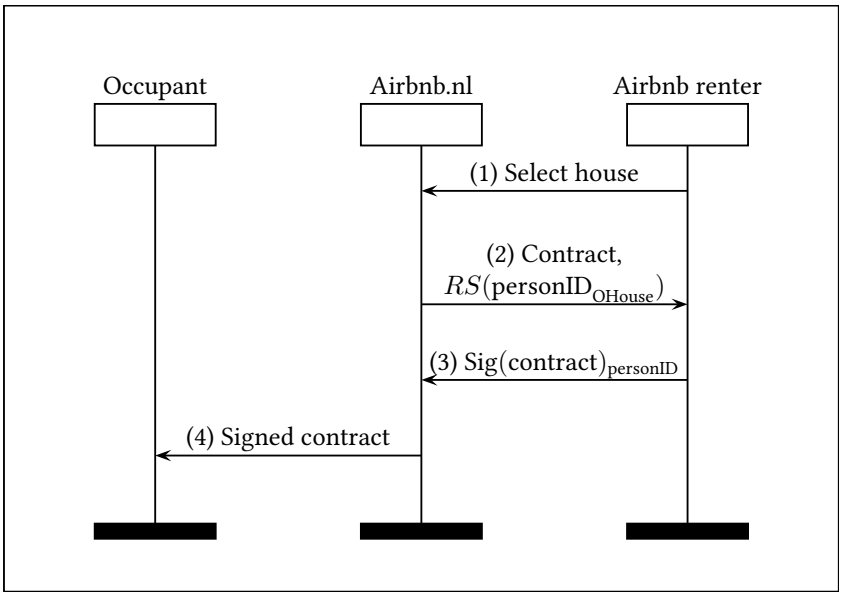


Figure 4.5: MSC: Rent a house on Airbnb

occupant's IRMA token, to be sure that the occupant is the main occupant of the house with the right *houseID* that he/she is adding to Airbnb. Airbnb also desires a proof of the *houseID* and *address* attribute from the occupant's IRMA concierge. In this way, Airbnb can verify the address of the house that the occupant is adding to Airbnb. The occupant has to authorise Airbnb to retrieve these attributes. This can be done in the same way as with the online shop scenario.

3. After scanning, the occupant sends an IRMA signature to the IRMA concierge to allow Airbnb to retrieve the *houseID* and *address* attributes from the IRMA concierge. The occupant also allows the request to reveal the *houseID* and *role* attributes from his/her own IRMA token.
4. The occupant sends a disclosure proof containing the *houseID* and *role* attribute to Airbnb.
5. The IRMA concierge sends a disclosure proof of the *houseID* and *role* attribute to Airbnb.

4.5.1 Rent house on Airbnb

The second step consists of a renter that rents the house that has been added by the occupant on Airbnb. The MSC can be found in Figure 4.5.

1. If, after some time, a renter wants to rent the house, he/she will select the house and enters his/her (payment) information.
2. To confirm the agreement, the renter needs to scan a QR code, which creates a link between the renter and Airbnb. If the link has been created, then Airbnb

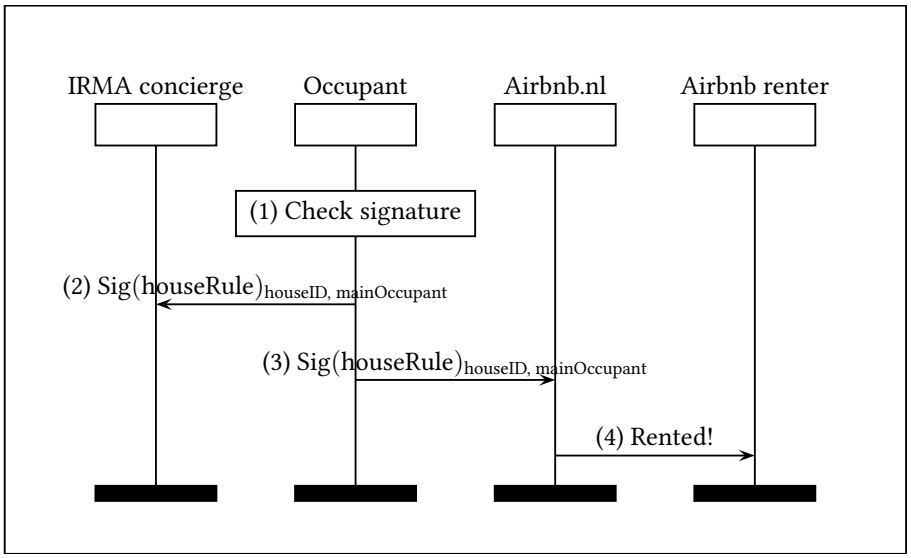


Figure 4.6: MSC: Provide access to Airbnb renter

will send a contract to the renter and requests the renter to sign this with his *personID* attribute from the *OHouse* credential. This is a *pre-selected* signature: Airbnb selects the attributes that are needed to sign the signature and also shows the data (the contract) that has to be signed.

3. The renter needs to sign this contract with his/her *personID* attribute, and sends the IRMA signature back to Airbnb.
4. Airbnb sends the signed contract to the occupant and notifies that his/her house has been rented.

4.5.2 Provide renter access to the house

The occupant will provide access to the house by adding a house rule after the house has been rented. The MSC of this protocol step can be found in Figure 4.6.

1. The occupant verifies the signature on the contract.
2. If the signature is valid, then the occupant will add a house rule to the IRMA concierge to allow the renter access to the occupant's house. This rule is signed with the *houseID* and *role* attributes and consists of the following parts:

`actor`	Person with *personID* = X
`action`	is allowed to open
`actee`	the door of my house
`condition`	between STARTDATE and ENDDATE
`goal`	to enter my house that he/she rented.

3. The occupant sends this house rule also to Airbnb as a confirmation that access has been provided.

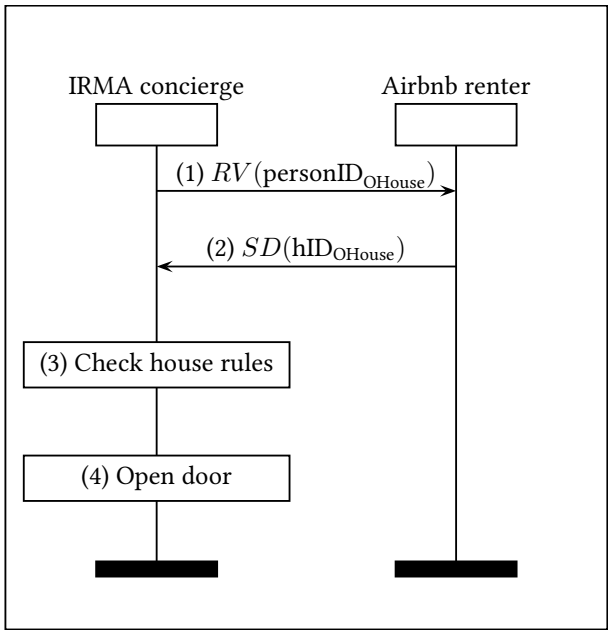


Figure 4.7: MSC: Renter opens the house

4. Airbnb notifies the renter that the request has been completed and that the renter is allowed to access the house.

4.5.3 Renter accesses the house

The final step consists of the Airbnb renter that will open the door of the house that he/she rented. The MSC can be found in Figure 4.7.

1. When the renter wants to access the house and stands in front of the door, the IRMA concierge requests the renter to disclose his/her *personID* attribute to the IRMA concierge.
2. After this request, the renter will give a proof of his/her *personID* attribute to the IRMA concierge.
3. The IRMA concierge checks if there is a valid house rule that allows this person access.
4. If there is a valid house rule, then the IRMA concierge will send a signal to the electronic lock, which will open the door.

Chapter 5

Implementation of IRMA signatures

In Chapter 3.3, we introduced IRMA signatures. In this chapter, we describe our implementation of IRMA signatures. We will start by introducing some technical details of the current verification protocol. The explanation of these details is required to implement the needed changes in the protocol that add support for IRMA signatures.

5.1 Current verification protocol

We first introduce the current verification protocol. We will start by providing a high-level overview of the different parties and components in the protocol. After this overview, we show how attributes are specified. Using this specification, we explain how a complete IRMA verification session is conducted.

5.1.1 High-level overview

As already introduced, the IRMA system consists of three main actors. The first actor is the *user*, who needs to prove the possession of some attributes to a service provider. The service provider acts in this case as the second actor, the *verifier*. The attributes are issued by a third actor, the *issuer*, to the user. The issuer is only involved during the issuance of attributes; during an attribute disclosure proof, only the user and service provider communicate.

Because of the direct communication between the three different parties, it is required that all parties support some parts of the IRMA protocol. Nowadays, every user has a smartphone, which means that it is easy to provide each user with an IRMA smartphone application that can act as a secure IRMA token. However, this is not the case for service providers, because they need to adapt their website to support the IRMA protocol, which means they have to support the cryptographic IRMA operations. This requires an effort from their side, which should be avoided if possible.

To make IRMA support easier to implement and maintain for service providers, the IRMA team introduced a fourth party, namely the IRMA API server, which is used to separate between the logic of the website of the service provider and the cryptography that is needed to verify attributes. The IRMA API server can be hosted by the service provider itself, but this task can also be outsourced to another party.

The communication between the three parties (user, service provider, IRMA API server) proceeds as follows (Figure 5.1 shows an MSC chart of this protocol)[1]:

1. The user visits the website of the service provider, who requests the user to reveal some IRMA attributes.
2. In order to obtain these IRMA attributes, the service provider sends a *disclosure proof request* to the IRMA API server. This request contains the attributes that the user has to reveal to the service provider. We will explain the disclosure proof request data type in Section 5.1.3.
3. The IRMA API server returns a session identifier to the service provider. This identifier is wrapped in a URL to the api server.
4. The service provider communicates this URL to the user via a QR code.
5. The user scans this QR code with the IRMA smartphone app. The app will follow the URL in the QR code, which connects the smartphone to the IRMA API server.
6. The IRMA API server requests the attributes from the user's IRMA app by sending the disclosure proof request to the user's IRMA app.
7. The IRMA app asks the user for consent. If consent has been provided, the IRMA app sends a disclosure proof of the requested attributes to the IRMA API server.
8. The IRMA API server verifies the disclosure proof and sends the result of this verification to the service provider in a signed JSON token. Note that the service provider has to trust the IRMA API server completely, since it does not verify the disclosure proof itself. The service provider also has to possess the public key of the IRMA API server, because this is used to verify the JSON tokens.

Issuance of credentials will follow almost the same protocol, except for the second step: instead of a *disclosure proof request*, an *issuing request* will be sent. An issuing request specifies the values of the attributes in the credentials that will be issued to the user. It can also request the user to first disclose some attributes, before the new credentials will be issued. Since issuing is a separate step and not relevant for our IRMA signature implementation, we will not discuss the issuing protocol in detail.

5.1.2 Attribute specification

It is required that issuers, verifiers and the IRMA application of the users know which credentials exist and how credentials are built (meaning which attributes are part of a credential and which are not). It is also required to specify the issuer of a certain

[1] See Github for the original documentation on this protocol: `https://credentials.github.io/proposals/irma-without-apdus/`

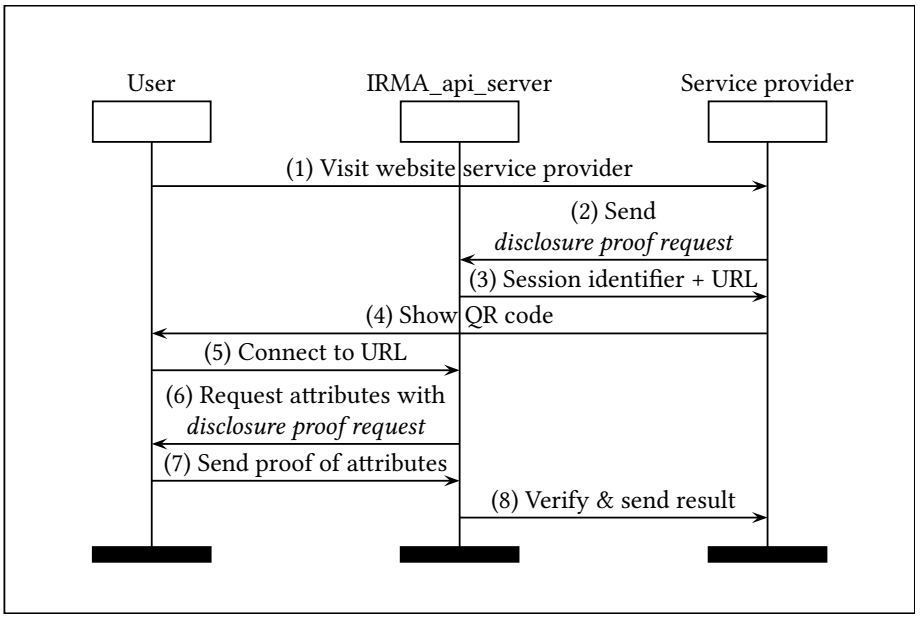


Figure 5.1: MSC chart of IRMA disclosure protocol

credential. The issuer's public key needs to be distributed to all the parties as well. All this information is combined in a *scheme*, which is managed by a *scheme manager*. Currently, there exists only one scheme, named `irma-demo`.[2] This scheme is managed by the IRMA team.

An IRMA scheme consists of a directory with XML configuration files. The directory structure is shown in Listing 5.1. The root directory (`SchemeManager`) contains several subdirectories (`IssuerName`) for each party that is allowed to issue IRMA credentials. These `IssuerName` directories contain three subdirectories: one directory with the issuer's private keys (`PrivateKeys`), one directory with the issuer's public keys (`PublicKeys`) and one directory containing the credentials that this issuer is allowed to issue (`Issues`). Note that the `PrivateKeys` directory is normally not present (because private keys remain private for each issuer), and only available in the `irma-demo` scheme, which is a demo implementation of a scheme. The `Issues` subdirectory contains the credentials that an issuer is allowed to issue. Each credential has its own directory (`CredentialName`) that contains a specification in an XML file: `description.xml`. This XML file contains a short description of the credential, a unique ID, the name of the issuer and the included attributes. Each attribute is described by a name and a description. Listing 5.2 shows an example of an issuer XML specification (with some less important XML tags omitted). This listing provides an example of a credential containing an *over18* and a *fullName* attribute.

```
<IssueSpecification>
<Name>testIssuer</Name>
<CredentialID>testNameAgeCredential</CredentialID>
```

[2]See Github for the specification of the IRMA schemes: `https://github.com/credentials/irma_configuration/tree/combined`

```
SchemeManager
+-- IssuerName
|   +-- Issues
|   |   +-- CredentialName
|   |       +--- description.xml
|   +-- PublicKeys
|   |   +-- 0.xml
|   |   +-- 1.xml
|   +-- PrivateKeys (need not be present)
|   |   +-- 0.xml
|   |   +-- 1.xml
|   +-- description.xml
|   +-- logo.png
+-- description.xml
```

Listing 5.1: The IRMA scheme (taken from `github.com/credentials/irma_configuration`)

```
<Description>
  Short description of credential
</Description>

<Id>1</Id> <!-- ID of credential -->
<Attributes>
  <Attribute>
    <Name>over18</Name>
    <Description>
      Whether the holder is over 18 or not.
    </Description>
    <Name>fullName</Name>
    <Description>
      The holder's full name
    </Description>
  </Attribute>
</Attributes>
</IssueSpecification>
```

Listing 5.2: Short version of XML issuer specification

5.1.3 Disclosure proof request

In the second step of the verification protocol, a *disclosure proof request* is sent from the service provider to the IRMA API server. This request is a JSON data type and contains the list of attributes that the user has to reveal to the service provider, along with an optional nonce and context string. Listing 5.3 shows an example of a disclosure proof request. As pointed out in the listing, the disclosure proof request contains three fields at the top-level, namely `nonce`, `context` and `content`. We will now explain the three fields in detail.

The `content` field is used to specify which attributes the user has to reveal to the service provider. This is specified with a list of JSON objects, where each object contains a label (`label`) and a list of *attribute identifiers* (`attributes`). The label is just a text string that is shown to the user in the IRMA app. An attribute identifier is a string in the form `Scheme-manager.Issuer.Credential.Attribute`. In our example listing we use a label `Age: 18+`, with two attribute identifiers, namely:

```
{ "nonce" : 123,
  "context" : 123,
  "content" : [
    { "label" : "Age: 18+",
      "attributes" : [
        "irma-demo.MijnOverheid.ageLower.over18",
        "irma-demo.Facebook.onlineAge.over18"
      ]
    },
    { "label" : "Name",
      "attributes" : [
        "irma-demo.MijnOverheid.Name.fullName"
      ]
    }
  ]
}
```

Listing 5.3: An example disclosure proof request

`irma-demo.MijnOverheid.age.over18` (referring to the *over18* attribute of the *age* credential, issued by *MijnOverheid* (meaning the Dutch Government)) and `irma-demo.Facebook.onlineAge.over18` (referring to the *over18* attribute of the *onlineAge* credential, issued by *Facebook*).

The list of attribute identifiers is interpreted as a *disjunction*: the user needs to possess/reveal only one of the attributes in the disjunction. However, multiple JSON objects (with a label and attribute identifiers) can be present at the same time. In that case, a user has to reveal at least one attribute from each label/attribute identifiers combination, which makes this list of JSON objects effectively a big conjunction containing disjunctions, for instance:

$$(A \vee B) \wedge (C \vee D \vee E) \wedge (F)$$

In this example, the disjunctions (so $A \vee B$, $C \vee D \vee E$ and F) are the different JSON objects in the list, where each of them contains their own label and where A until F are the attribute identifiers. This means that the user has to reveal at least *one* attribute from each disjunction (so that the complete disjunction will evaluate to `true`).

In our example from Listing 5.3, it means that the user has to reveal the *fullName* attribute from the *MijnOverheid* credential *and* either the MijnOverheid *over18* attribute *or* the Facebook *over18* attribute. This allows a service provider to specify multiple issuers as valid identity providers for a certain attribute. Figure 5.2 shows how this example disclosure proof request is communicated to the user in the IRMA application: the user can choose between either the MijnOverheid *over18* attribute, or the Facebook *over18* attribute.

One problem of specifying only the attributes itself in a list of disjunctions is that it is impossible for the verifier to enforce certain attributes values. This means that a user has to reveal the requested attributes, without knowing if they satisfy the requirements of the verifier. An example is a service that requires its users to be older than

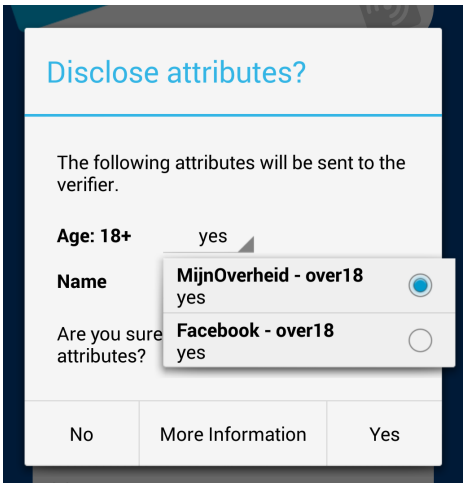


Figure 5.2: Disclosure dialog: the user can choose between MijnOverheid's and Facebook's *over18* attribute

eighteen. Here, an *over18* attribute could be used, where the value must be equal to *true*. We solved this problem by also specifying conditions in the specification. This allows the IRMA app to abort the signing protocol if some attribute values are not met. We discuss this improvement in Section 5.2.3, where we describe the signature specification data type. Recently, the IRMA team has implemented their own solution by adding a `value` tag to the disjunction.[3]

Besides the `content` field, we also have the `nonce` and `context` fields. The `nonce` field is used as challenge for the proof (see Section 3.2 for details). The `context` field contains the *context string*, which is a string that is used to list all the public parameters as well as the issuer's public key. The context string prevents values generated during this proof to be reused in another verification session, as explained in §4.2 of the Idemix Specification [1]. Note that in the current IRMA implementation, the validity of the context string is not checked. Also, the context string will in the current implementation be set to the SHA256 hash of zero if no string is provided by the service provider.[4]

5.1.4 Disclosure proof result

The last step of the IRMA verification protocol (Figure 5.1) is about sending the result of the IRMA verification session from the IRMA API server to the service provider. This result is a *disclosure proof result* data type, which is also a JSON data type. This result contains the status of the proof that shows whether the proof is valid. If the user had to reveal attributes, their values will also be included in the result that is sent to the service provider. A *disclosure proof result* JSON object is converted to a signed

[3] `https://github.com/credentials/irma_api_common/commit/75657837f0d659441a11048a2f023b5afd456ea3`

[4] `https://github.com/credentials/irma_api_common/blob/2cf58261559db0f7c86d37b14ddb3a3b2c0db671/src/main/java/org/irmacard/api/common/SessionRequest.java#L60-L62`

```
{
    "exp": 1466682469,
    "sub": "disclosure_result",
    "jti": "data from service provider",
    "attributes": {
        "irma-demo.MijnOverheid.ageLower.over18": "yes"
    },
    "iat": 1466682409,
    "status": "VALID"
}
```

Listing 5.4: Example disclosure proof result

JSON web token by the IRMA API server, before the object is sent to the service provider.

Listing 5.4 illustrates a disclosure proof result, based on the disclosure proof request that is shown in Listing 5.3. The `exp` (expiry date), `sub` (subject), `jti` (unique identifier) and `iat` (issued at) fields are standard JSON web token fields. The `attributes` field contains a list of the revealed attributes, along with their values. The `status` field indicates the result of the disclosure proof, which can be:

`VALID` The proof was valid, which means that the service provider can reliably use the attributes values can were sent with the proof.

`INVALID` The proof was invalid, something went wrong so this result must be discarded.

`EXPIRED` One of the attributes in the proof came from an expired credential. It is up to the service provide if he/she wants to accept the proof.

`MISSING_ATTRIBUTES` One of the required attributes from the disclosure proof request is missing and not revealed in the proof.

`WAITING` The IRMA API server is still waiting for a proof from the user's IRMA app.

5.1.5 Verifier authentication

The IRMA API server will only accept disclosure proof requests from allowed service providers. The IRMA API server enforces this by requiring the service provider to sign the disclosure proof request (by enclosing it in a JSON web token). The IRMA API server maintains a list of the public keys of the allowed service providers, and rejects requests of service providers that are not on this list. In addition, the IRMA API server can also specify which service provider is allowed to verify which attributes.

Note that these signed JSON web tokens only authenticate the path between the IRMA API server and the service provider (step 2, 3 and 8 from Figure 5.1). The paths between the user at one side and the IRMA API server and service provider at the other side are not authenticated. This means that every service provider can request attributes from a user, if it sets up an IRMA API server itself. The IRMA team has some plans to add authentication for service providers in the future.

Therefore, the user has to check the disclosure dialog (Figure 5.2) on his/her IRMA app carefully to check if the request for attributes is logical (for instance an online video portal that requests you to be older than eighteen does not need your social security number to verify your age). The user also currently has to check if he/she is visiting the real website of the service provider, which can be checked by validating the TLS certificate of the website. A service provider can also choose which IRMA API server it uses. The chosen IRMA API server can read all the attribute values of all its users and must therefore be completely trusted by both the service provider and the user, but for the user there is no way to check which IRMA API server is used. The authentication of both the service provider and IRMA API server to the user is therefore still not present and needs to be worked out. This requires both technical and organisational measures which we will not discuss.

5.2 Implementation of an IRMA signature protocol

In this section, we will elaborate on some important aspects of our implementation of IRMA signatures (that we introduced in Section 3). We start by providing an informal protocol overview that shows how an IRMA signature session looks for a user. After this overview, we will describe the IRMA signature protocol itself, in the same way as we did with the IRMA verification protocol in Section 5.1.1. After the protocol, we will define the used data types and specification. We will also show our implementation of the domain separation technique, that we introduced in Section 3.4.2. When confusion is unlikely, we often abbreviate the term 'IRMA signature' to 'signature'.

5.2.1 Informal protocol overview

Before we look into the exact protocol, we will first discuss how an IRMA signature session is conducted in terms of user experience. For this, we first need to recall an important difference in types of IRMA signatures: we actually have two types of IRMA signatures, namely signatures with *pre-selected* attributes and signatures with *self-selected* attributes. The service provider (or any other 'signature requester') selects the attributes in the case of signatures with *pre-selected* attributes, while on the other hand the user can choose the used attributes him/herself in the case of signatures with *self-selected* attributes. We will now informally show how such a signature session could look for a user:

Self-selected attributes: Signatures using *self-selected* attributes require that the user chooses the credentials that he/she wants to include. It is a very ad hoc way of signing: if a user wants to sign some data (for example a document on his/her computer), then it should be possible to do that. Since there is no interaction with an external party like a service provider or api server, it is not possible to scan a QR code and let an api server send some requests to the user's IRMA app. Instead, the user could send the document to his/her IRMA application, for instance using the share button/functionality that is present on smartphones. The IRMA application could then ask the user to select the attributes that the user wants to include in the IRMA signature.

Pre-selected attributes: The protocol flow for creating a signature with *pre-selected*

attributes looks much more like the IRMA verification protocol flow: if a user wants (or, in this case, is even required) to sign some data for a service provider, he/she could scan a QR code with the IRMA app. The service provider can after scanning send a request with the required attributes and the data that has to be signed to the user's IRMA app. The user can then review the included attributes and data that will be signed. After the user has approved, the IRMA app signs the data and sends the IRMA signature back to the service provider.

Two two cases look very different in terms of protocol flow. However, they both either create or obtain a signature specification that lists which attributes are selected and included in the signature. This specification is needed by the signature verifier, to check if the correct attributes are included.

To verify a normal disclosure proof, the verifier uses the challenge, the public keys of the issuers and the response of the user. The challenge and the public keys of the issuers can be retrieved from the disclosure proof request. Since IRMA signatures are a special case of IRMA disclosure proofs, the signature verifier also needs this information to verify an IRMA signature. In addition, the signature verifier needs the original message (to calculate the hash of that message and thus the challenge) and the signature specification (to check if the correct attributes are included). Furthermore, the signature itself is required.

5.2.2 IRMA signature protocol

Since we introduced how a signature session could look for a user, we can now discuss our IRMA signature protocol, using a description of the different steps and an MSC chart. We will do this in the same way as we did with the verification protocol in Section 5.1.1.

For a signature scheme, we need to implement the three signature steps, namely *key generation*, *signature generation* and *signature verification*. The *key generation* step is already implemented, since we assume that the IRMA smartphone app already contains some credentials.

For the *signature generation* step, it depends on the type of signature that we are going to create, as pointed out by the previous section. We will first focus on signatures with *pre-selected* attributes. We saw in Section 3.4.1 that we can in this case use almost the same protocol as with IRMA verification (introduced in Section 5.1.1), except for the challenge that is sent to the user/prover. This challenge is enclosed in the *disclosure proof request* that we discussed earlier. We will instead include the values that are needed to compute the challenge in the *signature specification* file, which will also be a JSON data type, based on the disclosure proof request. We will specify this data type in Section 5.2.3.

We will first show our implementation of a protocol for IRMA signatures with *pre-selected* attributes, based on the protocol from Figure 5.1. Figure 5.3 shows an MSC chart of this protocol:

1. Like with the verification protocol, the user visits the website of the service provider, which requests the user to sign some data with IRMA attributes.
2. In order to obtain an IRMA signature, the service provider sends a *signature*

request to the IRMA API server. This request contains the attributes that the user has to include in the signature as well as the message that must be signed. We will explain the signature request data type in the next section.

3. The IRMA API server returns a session identifier to the service provider. This identifier is wrapped in a URL to the api server.
4. The service provider communicates this URL to the user using a QR code.
5. The user scans this QR code with the IRMA smartphone app. The app will follow the URL in the QR code, which connects the smartphone to the IRMA API server.
6. The IRMA API server requests the signature from the user's IRMA app by sending the signature request to the user's IRMA app.
7. The IRMA app shows the message to the user, and asks the user to sign this message with the specified attributes. If the user agrees to sign the message, the IRMA app sends the IRMA signature to the IRMA API server.
8. The IRMA API server verifies the signature and sends the result of this verification, along with the signature specification and the signature itself to the service provider in a signed JSON token. It is important to point out that the service provider has to trust the IRMA API server for the verification result, but can also verify the signature itself if that is desirable. Like with the verification protocol, the service provider does also in this case have to possess the public key of the IRMA API server, because this is used to verify the JSON tokens.

   We also extended the IRMA API server with an API that allows a signature verifier to verify a signature after the protocol is completed. This means that a signature verifier can always verify a signature using the IRMA API server, without requiring to run its own IRMA software, the verifier only has to trust the IRMA API server, like with the verification case. We can imagine that in the future, a standalone application or library can be developed that allows signature verifiers to verify a signature.

The other type of signatures are signatures with *self-selected* attributes. With this type of signatures, there is no other party than the user involved, so the protocol can be much simpler. We should emphasise that we did *not* implement this protocol. In our implementation, we use IRMA signatures to preserve the integrity and authenticity of the house rules and to sign a renting contract. While for the authenticity of the house rules, an occupant could be free in choosing his/her attributes, we did not implement this and used fixed attributes for the house rules. This means that the IRMA concierge always sends a signature specification containing the attributes to the occupant, which makes that we only have to deal with signatures with *pre-selected* attributes.

However, the fact that we did not implement a protocol for signatures with *self-selected* attributes does not mean that we cannot propose a protocol for them. A protocol for this type of signatures could look as follows:

1. The user sends the data to be signed to the IRMA app, via for example the share button on the phone.
2. The IRMA app shows a dialog, which allows the user to select which attributes

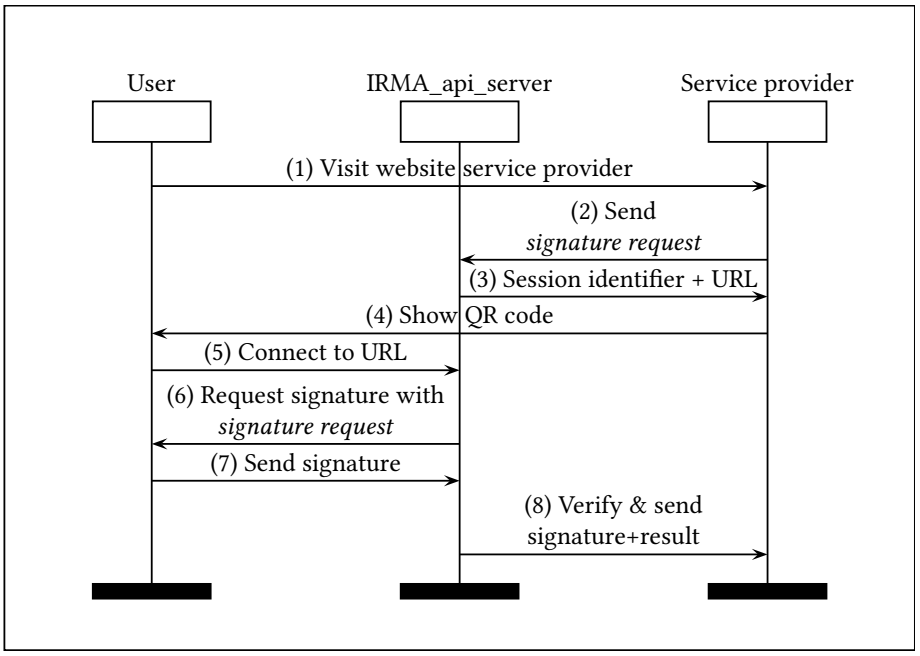


Figure 5.3: MSC chart of IRMA signature protocol with *pre-selected* attributes

he/she wants to include in the signature.

3. The IRMA app shows a final dialog to ask the user for confirmation on this signature.
4. After approval, the IRMA app asks to store the IRMA signature or share it with another application.

This protocol requires the user to somehow send the data that he/she wants to sign to the device containing the IRMA app. This is not the best thing to do in terms of user experience. We could enhance this by for instance creating an IRMA desktop or web 'helper' application. This helper application can be used to create the signature specification. The user can also use this helper application to send the data that will be signed to the device by sending or uploading these data to the helper application. The helper application can show a QR code to connect to the IRMA app on the smartphone. Note that the IRMA helper application does not need to do IRMA verifications or other cryptographic operations. In summary, this protocol could look like this (Figure 5.4 shows an MSC chart of this protocol):

1. The user adds or uploads data that he/she wants to sign to the IRMA helper application.
2. The user fills in which attributes he/she wants to use to sign the data.
3. The IRMA helper application shows a QR code containing a session identifier.
4. The user scans this QR code with the IRMA smartphone app. The app will follow the URL in the QR code, which connects the smartphone to the IRMA helper

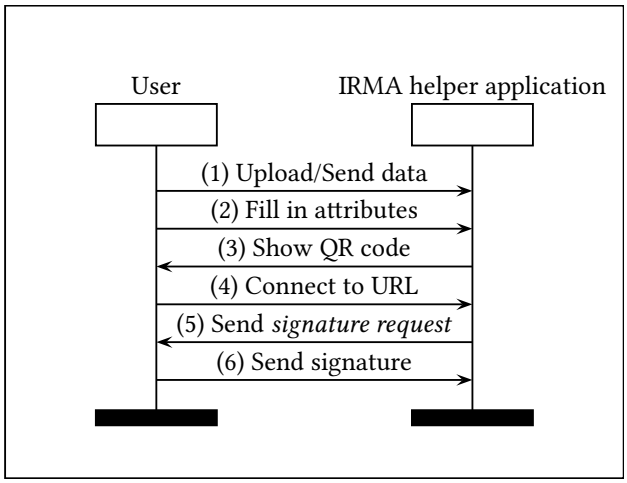


Figure 5.4: MSC chart of IRMA signature protocol with *self-selected* attributes using an IRMA helper application

application.

5. The IRMA helper application will send a signature request to the user's IRMA app.
6. The user's IRMA app asks the user for consent and if he/she agrees, creates a signature and sends it back to the IRMA helper application. The IRMA helper application can store the signature somewhere or can ask the user to store it somewhere.

The IRMA helper application needs to be directly reachable via the Internet for this protocol to work. In the case of a web application, this is not a problem, since a web application is by definition reachable via the Internet. However, using a web application would mean that the owner of the application is able to see every signature that is created with it. Another option would be to use a relay between the IRMA helper application and the IRMA smartphone app. This has been done in the past, when IRMA was only implemented on a smart card.[5]

5.2.3 Signature specification

In the previous sections, we referred to a *signature specification* data type. We will specify this data type in this section. As already described, the main use case for a signature specification file is to specify *which* attributes will be included in an IRMA signature to sign a certain statement. Listing 5.5 shows an example signature specification data type. Like with the disclosure proof request data type, it contains the `nonce`, `context` and `content` fields at the top-level in the data structure. In addition, it also contains a `message` and `messageType` field.

The `message` and `messageType` fields are used to communicate the message that must be signed. The `messageType` field determines the type of message in the

[5]See `https://github.com/credentials/irma_web_relay`

```
{
    "message" : "Message to be signed",
    "messageType" : "STRING",
    "nonce"   : "Nonce, will be combined with context and
        message hash to get challenge",
    "context" : 456,
    "content": [
        {
            "label": "Over 18",
            "condition" : "true",
            "attributes": [
                "irma-demo.MijnOverheid.ageLower.over18",
                "irma-demo.Facebook.onlineAge.over18"
            ],
            "conditions" : [
                { "irma-demo.MijnOverheid.ageLower.over18
                    " : "yes" }
            ]
        },
    ]
}
```

Listing 5.5: Example signature specification

`message` field. Currently, we have only implemented the message type `STRING`, which means that the `message` field directly contains the string that is to be signed. A string can be shown directly to the user in the IRMA app. However, the message type `STRING` is only useful for short messages without any formatting. We realise that service providers would like to ask users to sign more data, like complete PDF documents. This could be achieved by for instance introducing a message type `PDF`. In that case, the `message` field could contain a URL to a PDF file, that the user has to sign. The IRMA app can obtain the PDF file by visiting the URL, show the PDF file to the user (which allows the user to check that he/she is signing the right document), and finally sign the hash of the file. However, signing arbitrary PDF files can be a security issue because PDF files can be displayed differently in different viewers (see for instance [10] and [11]) and can also contain active and dangerous content. This means that a user could be tricked into signing something different than what he/she sees on his/her screen. Therefore, it is desirable to sign only plain text or close to plain text documents. Further research is needed for a way to sign bigger and formatted messages in a secure way.

The `content` field is used to specify which attributes need to be included in the signature. Just as with the disclosure proof request data type, it contains a `label` field that is shown to the user in the IRMA app. It also contains a list of attribute identifiers in the `attributes` field, which is also the same as with the disclosure proof request data type.

The optional `conditions` field is used to enforce certain attributes values for the attributes that are present in the disjunction list. As we discussed in Section 5.1.3, with

the current verification protocol it is not possible to enforce attribute values. For the signature protocol, we solved this by including a `conditions` field in the `content` field. The `conditions` field contains, like the `attributes` field, a list of attribute identifiers, but each attribute identifier is extended with a string value that contains the required condition. This allows the signer to abort the protocol if he/she does not possess attributes with the required values to create a valid signature.

We also specify an optional `condition` field, that is used to specify a *default* non attribute-specific conditions. This condition is used if no other attribute-specific condition is present for an attribute. However, if there is a attribute-specific condition specified, then it takes priority over the default condition.

Listing 5.3 shows an example of how these conditions work. In this example, the required value for MijnOverheid's *over18* attribute is *yes*, as specified by the corresponding `conditions` field of, while Facebook's *over18* attribute does not have such a field. Facebook's *over18* attribute value will therefore be enforced by the default `condition` field, which in this case is *true*. This means that the value for Facebook's *over18* attribute should be equal to *true*. In summary, this means that if the user want to sign a document in compliance with this specification, he/she has to use either Mijnoverheid's *over18* attribute, that must be equal to *yes*, *or* he/she has to use Facebook's *over18* attribute, that must be equal to *true*.

Note that we could also include the condition values in the `attributes` field, to avoid redundancy. This redundant specification allow us to also expand the disclosure proof request data type with a `conditions` field in the future, without breaking compatibility with older clients that do not support this field in verification sessions. However, with the very recent state of the IRMA project, this would probably not be an issue anymore. But these developments in the IRMA project were done after we constructed our specification.

The remaining fields that we have not explained yet are the `nonce` and `context` fields. We will start with the explanation of `context` field. The `context` field contains a context string, which is used to list all the public parameters in a proof. It also prevents values in a proof to be reused in another proof. We already introduced the context string in Section 5.1.3. We also saw that the context string is still not implemented in the IRMA verification protocol. This means that we cannot propose a specification of the context string for the IRMA signature protocol that is compatible with the verification protocol. We think it is an good idea to define and implement the context string at once for both the verification and signature protocol. Therefore, we decided to not include and implement the context string in our implementation that only implements the IRMA signature protocol.

The last field that we need to explain is the `nonce` field. With IRMA signatures, we already use the hash of the message as the input for an IRMA proof. However, we still want to guarantee freshness, which means that we need to include a nonce as well. The input for the proof will therefore be calculated as:

$$\texttt{input} = H(\texttt{nonce}, H(\texttt{message}))$$

We use *SHA256* as the hash function H. This double hashing seems to be unnecessary, but as [5] explained, we want to keep the changes to the current IRMA protocol to a minimum, which means that we need to use a double hashing technique to calculate the input for the IRMA proof.

Byte(s)	Explanation
0x30	DER encoding follows
0x0D	Number of bytes that follow
0x02 0x01	Integer follows (0x02) with length 1 (0x01)
0x03	Integer with value 3 (number of encoded values, which is 3)
0x02 0x01	Integer follows with length 1
0x01	Integer with value 1 (first encoded integer)
0x02 0x01	Integer follows with length 1
0x42	Integer with value 65 (second encoded integer)
0x02 0x02	Integer follows with length 2
0x04 0x01	Integer with value 1025 (third encoded integer)

Table 5.1: 1, 65 and 1025 are encoded into an ASN.1 byte sequence.

5.2.4 Domain separation

As we introduced in Section 3.4.2, we must make sure that signatures and disclosure proofs are two completely different domains that are strictly separated. The signature and disclosure proof context strings could be enough to separate these two domains if we make them different from each other. This will make the final challenge also different in the two domains. However, we saw that the current IRMA implementations do not validate the context string. Instead, they just accept the context string they receive, without checking anything. This becomes a problem if we add signatures to the protocol, because this could allow an attacker to use a signature when a 'real' disclosure proof is required. Vice versa, a disclosure proof could be used as a signature. Even if we would implement the context string for both verification and signature sessions, we would still have the risk that old clients accept everything. Therefore, we want to separate the two domains in a way that IRMA signatures surely break compatibility with older clients.

The challenge for IRMA disclosure proofs is encoded as ASN.1 As briefly mentioned in Section 3.4.2, ASN.1 describes a way to uniquely encode a data string. In this section, we will briefly introduce ASN.1 and we will show how it can be used to separate the verification and signature domains. If we look into the current IRMA implementation at how the challenge c is calculated, then we see that this is calculated as follows:

$$c = \text{SHA256}(\text{ASN1}(\texttt{context}, A, Z, n))$$

In this case, A and Z are fixed numbers used in the disclosure proof[6] and n is the input nonce, which is calculated by combining and hashing the nonce and message string from the signature specification together (as we saw in the previous section). The `context` variable is the previously discussed context string and is in the current IRMA implementation set to the SHA256 hash of the empty string.

Before the four values will be hashed, they are *ASN.1* encoded with *DER* encoding. ASN1 is a notation that describes rules on how certain data should be encoded and decoded during transfer [12]. Type information and the data structures will be preserved. There is only one correct way to encode a data string, which makes this encod-

[6]See the link `https://www.irmacard.org/wp-content/uploads/2013/05/Idemix_overview.pdf` for a short overview on how this mathematically works

Byte(s)	Explanation
0x30	DER encoding follows
0x10	Number of bytes that follow (now 3 more)
0x01 0x01	Boolean follows (0x01) with length 1 (0x01)
0xFF	Boolean with value true (to indicate that this is a signature)
0x02 0x01	Integer follows (0x02) with length 1 (0x01)
0x03	Integer with value 3 (number of encoded values, which is 3)
...	From now on same values as with the disclosure proof example

Table 5.2: 1, 65 and 1025 are signature-encoded into an ASN.1 byte sequence.

ing unique. Within IRMA, the following ASN.1 structure is used for the calculation of a challenge:

```
ASN1.Sequence< length(int), value(int), ...
```

Table 5.1 shows an example of how this encoding is used. In this example, the integers 1, 65 and 1025 are encoded into the following ASN.1 sequence:

```
0x30, 0x0D, 0x02, 0x01, 0x03, 0x02, 0x01, 0x01,
0x02, 0x01, 0x41, 0x02, 0x02, 0x04, 0x01
```

In order to support signatures with a proper domain separation, we need to change this encoding in such a way that will never be compatible with the current encoding. We can do this by adding a boolean (0x01) in front of the integer (0x02) that represents the length. This boolean will be set to true (0xff) in case of a signature, while it does not exist at all in the case of a disclosure proof. Table 5.2 shows how the previous example should be changed to ASN.1 encode the values for a signature, which results in the following byte stream:

```
0x30, 0x10, 0x01, 0x01, 0xFF, 0x02, 0x01, 0x03, 0x02,
0x01, 0x01, 0x02, 0x01, 0x41, 0x02, 0x02, 0x04, 0x01
```

We can now look at the data type of the third octet (either integer or boolean), which determines if we need to deal with a signature or a disclosure proof. This ASN.1 encoding scheme allows for proper separation of the verification and signature domains, while still retaining compatibility with the current IRMA implementation.

5.2.5 Storing a signature

Signatures need to be stored and sometimes also transferred to other parties. For disclosure proofs, the *disclosure proof result* data type is used. That data type lists the attribute values and the result of the proof. We can use this as a basis and define a *signature proof result* data type. For IRMA signatures, we also need to include the signature itself in the data type. Furthermore, as we discussed before, the signature specification file is needed to verify the signature. We want to avoid the need of having two separate files for one signature. Therefore, we decided to include the signature specification in the signature proof result data type. Listing 5.6 shows an example IRMA signature that is in compliance with the earlier introduced signature specification.

As the listing shows, an IRMA signature has some overlap with a disclosure proof

```
{
    "exp": 1448636691,
    "sub": "signature_result",
    "jti": "data from service provider",
    "attributes": {
        "MijnOverheid.ageLower.over18": "yes",
    },
    "iat": 1448636631,
    "status": "VALID",
    "signature" : "<<disclosureProof>>",
    "message" : "The message that has been signed by this
        signature",
    "messageType" : "STRING",
    "nonce"    : "Nonce, will be combined with context and
        message hash to get challenge",
    "context" : 456,
    "content" : { // Original DisjunctionList with all
       the attributes and conditions
    }
}
```

Listing 5.6: Example IRMA signature in compliance with the earlier introduced signature specification

result data type. The `exp`, `sub`, `jti`, `attributes`, `iat` and `status` fields are the same as with the disclosure proof result data type. The `signature` field contains the disclosure proof, which is a serialised version of the *ProofList* that represents an IRMA disclosure proof.[7] This field can be seen as the mathematical signature itself, and can, if combined with the challenge, be verified stand-alone. Everything else is only needed to verify if a signature is in compliance with the required specification.

The remaining fields (`message`, `messageType`, `nonce`, `context` and `content`) are included from the signature specification file and are used to calculate the challenge and verify the signature. The condition fields are in this case compared with the attributes that will be returned from the signature verification. Note that the `attributes` field is redundant and can also be derived from the `signature` field. We choose to also include it, because this makes it at least possible for service providers that do not support IRMA to 'read' a signature.

5.2.6 Unimplemented parts

As [5] already suggested in §4.1, timestamps are required. A timestamp on a digital signature determines when a document or statement is signed. It proves that the content of the signed document has not been changed since it was signed and that the document existed at that point in time. To properly use timestamping, we have to

[7] Source code of ProofList object: `https://github.com/credentials/credentials_idemix/blob/7f8b95fda07cc7b3e09a38708da4a38a5b17a847/src/org/irmacard/credentials/idemix/proofs/ProofList.java`

use a trusted timestamp provider. We have not implemented this in our work, which means that this is something that still needs to be done.

The second important unimplemented part is revocation: if certain attributes get revoked at a point in time, then signatures should be revoked if they are created after the attributes are revoked. There are a few ways to achieve revocation in IRMA. One way is discussed by Lueks et. al in [13]. However, revocation also still needs to be implemented for disclosure proofs, which is why we have not implemented this for signatures. Although revocation is not implemented, it is already possible to detect IRMA signatures that are created with expired attributes.

We discussed the differences between signatures with *pre-selected* and *self-selected* attributes. We only implemented a protocol for signatures with *pre-selected* attributes. However, we have proposed some protocols for signatures with *self-selected* attributes as well. Since both types of signatures require a signature specification, it will be easy to add support for signatures with *self-selected* attributes, without breaking or changing our implemented protocol for signatures with *pre-selected* attributes.

5.2.7 Integration with official standards

In the case that IRMA signatures will be used in practice, it might be possible that multiple implementations will be created for both the IRMA signature algorithms as well as for other parts of the IRMA eco system. One important part in this case is standardisation, which means that different implementations and different types of IRMA tokens should be able to create and verify each other's signatures. In this section, we will briefly look if standardisation could be achieved for IRMA signatures.

For normal digital signatures, an official[8] standard exists, namely the *Digital Signature Standard* [14]. However, this standard only describes the algorithms to generate and verify a signature, by describing *key generation*, *signature generation* and *signature verification*. Storing a signature is not specified in this standard, but an RFC standard is mentioned for RSA signatures: *PKCS #1* [15]. PKCS #1 describes how RSA encryption, decryption and signing is done and how the resulting data can be stored in ASN.1 encoded byte sequences. A general and not algorithm-specific standard is the *Cryptographic Message Syntax*, specified in RFC 5652 [16]. This standard is based on *PKCS #7*.

IRMA is based on Idemix, and Idemix has a complete specification [1]. At the lowest level, IRMA still uses the same format for the disclosure protocols and attribute proofs. However, IRMA extended on Idemix by for instance always adding a validity date and some credential semantics.[9] IRMA also introduced high-level protocols with different parties (for instance the IRMA app, the IRMA API server and the clearly distinctive tasks of the service provider/verifier), which we explained in Section 5.1. Since these protocols are still work in process and since these protocols are still changed/extended, no specification and complete documentation exist yet. Before IRMA signa-

[8] With official, we mean in this case 'technical': a format on how signatures are generated and stored. There also exist legal standards for digital signatures that describe the legal role of a digital signature, see for instance the following EU regulation: `http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2014.257.01.0073.01.ENG`. We will only consider the technical standard.

[9] See `https://github.com/credentials/credentials_idemix` for the low-level IRMA layer that communicates with the Idemix layer.

tures can be standardised, we think that it is important that the other IRMA protocols are officially specified first.

5.2.8 Source code

Our implementation of IRMA signatures is available on Github. We are still trying to get it merged with the core IRMA software. Here are the links of the relevant components:

- `https://github.com/koen92/credentials.github.io`
- `https://github.com/koen92/credentials_idemix`
- `https://github.com/koen92/credentials_api`
- `https://github.com/koen92/irma_api_server`
- `https://github.com/koen92/irma_api_common`
- `https://github.com/koen92/irma_js`
- `https://github.com/koen92/irma_android_cardemu`
- `https://github.com/koen92/irma_android_library`

Chapter 6

Implementation of a Proof of Concept

We have introduced some known IRMA techniques, as well as described how we implemented IRMA signatures. We also introduced some scenarios where we make use of IRMA in an IoT context. These scenarios, however, were still only a high-level overview of how IRMA *could* work in an IoT context. Many details were ommitted, as well as issues that might arise when a scenario was implemented for real.

Therefore, we implemented a complete proof of concept (PoC), which we will describe in this chapter. This PoC is used to show how the concepts can be integrated into a practical service that is closely related to the goals of the Tippiq project. This PoC is based on the Airbnb scenario that we introduced in Section 4.5.

In the Airbnb scenario, we showed how an occupant can put his/her house on the Airbnb website (Figure 4.4). After the house is available on the website, a renter can rent it (Figure 4.5). After the renter agrees on the renting (by signing a contract), the occupant provides him/her access to the house by adding a house rule to the IRMA concierge (Figure 4.6). The renter can subsequently access the house by proving to the concierge that he/she is allowed to access the house. The IRMA concierge will open the door if the proof is correct (Figure 4.7).

We implemented a basic and slightly customised version of the Airbnb scenario into a proof of concept. In this chapter, we will show some design and implementation details. We will start by introducing the different components we used in Section 6.1. After this introduction, we show all aspects of the protocol in Section 6.2, which means that we explain the used IRMA identities and the used Tippiq house rules. We will elaborate on the protocol itself with UML sequence diagrams, which specifies all the user and device interactions. We also provide some screenshots of the complete PoC setup. We finish this chapter by suggesting some relevant improvements for our PoC in Section 6.3.

6.1 Components

With this PoC, we want to show how we can adapt IRMA to an IoT and house context. We built a central house gateway that we call the IRMA concierge. We provided the IRMA concierge with its own IRMA attributes.

However, we try to make our proof of concept as *generic* as possible. While we only work out one scenario (Airbnb), we split the IRMA concierge into different components, that can be swapped for other components. This approach allows Tippiq to implement parts of our proof of concept in their product, without being dependent on the complete PoC. In this section, we will introduce the different software components. After that introduction, we will show the hardware components. When we have showed both the hardware and software components, we will introduce the used IRMA attributes.

6.1.1 Software components

As we already pointed out in Section 4.5, we have the following parties:

Airbnb.nl The Airbnb website. We will refer to this party as one independent component. We let the other parties interact with the website, and will propose only small changes for this website. Because changing something on the Airbnb website requires cooperation from Airbnb in our project. We do not have this cooperation. We did not implement this part of the PoC. However, we will mention the required changes to the Airbnb website.

IRMA concierge The IRMA concierge is the central house gateway, as we already introduced. The IRMA concierge will directly interact with all the other parties. It will also store and evaluate the Tippiq house rules.

Occupant The occupant is the owner of the house that is going to be rented. Therefore, the occupant can control the IRMA concierge by adding and removing house rules. The occupant has a smartphone with the IRMA application, containing his/her personal attributes. Note that these attributes differ from the attributes stored on the IRMA concierge (see Section 4.1.2).

Airbnb renter The Airbnb renter will visit the Airbnb website to rent the occupant's house. He/she will also interact with the IRMA concierge, when he/she is entering the occupant's house. However, there is no direct interaction between the occupant and the renter. Like the occupant, the renter also has a smartphone containing IRMA attributes.

The IRMA concierge has to do various tasks: it interacts with all the other parties, it verifies IRMA attributes and signatures, it stores house rules, it checks house rules for validity and it controls the electronic lock of the door. We will separate all these parts into different components. We let only one component do the IRMA verifications and the interactions with the other parties. This will be the IRMA module. Since we are using a relatively simple electronic door lock, we will let the IRMA module also control this door lock.

We could have offloaded the IRMA-specific tasks to the IRMA API server (as introduced in Section 5.1.1), but the IRMA API server did not exist at the time we developed

this PoC. Therefore, we had to manually integrate all the IRMA libraries in our IRMA module. This also explains why we moved all the other non-IRMA tasks to different components.

This separation means that we, besides the IRMA module, introduce three other components:

House rules database This component is just a database, where all the house rules are stored. All the rules that are stored are signed with an IRMA signature. The reason for signing all the rules will be explained later.

Control API This component checks if a house rule is valid. It retrieves requests with input data from the IRMA module. Examples of such input data are the `actor` that is standing in front of the house and the `action` that will be taken (opening the electronic door lock). The control API will, after a request, retrieve the relevant house rules from the house rules database. It checks if one rule matches the input data and is valid. If there is a valid rule, the control API replies with `yes` to the IRMA concierge. Otherwise, it replies with `no`. It adds the corresponding IRMA signature of the house rule to this reply. This signature will be checked by the IRMA module.

House rules API The house rules API is the interface to the house rule database. It allows occupants (and in the future maybe also service providers and other third parties) to add house rules to the database by providing a web interface to the occupants. It also requests the occupants to sign new and changed house rules, before the rules will be inserted into the database.

All these components communicate with each other via a JSON API over HTTP. Since all the components run on the same hardware device in our PoC, we did not implement authentication between these components. The protocol for entering the house (Figure 4.7) now looks as follows at a high level:

- The IRMA module displays a QR code on the screen that is attached to the outside of a house.
- The renter scans this QR code. After the connection has been established, the IRMA module requests the *personID* IRMA attribute from the renter's smartphone (we will show later in this section which IRMA attributes are used by which party).
- If the attribute is valid, then the IRMA module sends it to the control API, along with the action (opening the door).
- The control API retrieves the relevant house rules from the database, and checks whether this person (with this *personID*) is allowed to open the door. If he/she is allowed, then the control API replies with `yes`, otherwise with `no`.
- The IRMA module verifies the IRMA signature that is sent by the control API. If this signature is valid, it sends a signal to open the door if it receives `yes` from the control API. If it receives a `no`, then it will communicate this to the renter/person in front of the door by showing this on the screen.

The second part of the protocol is adding new house rules that are signed with an IRMA signature. These rules are used to provide other people access to the house:

- The occupant visits its 'house rule portal', where the house rules can be man-

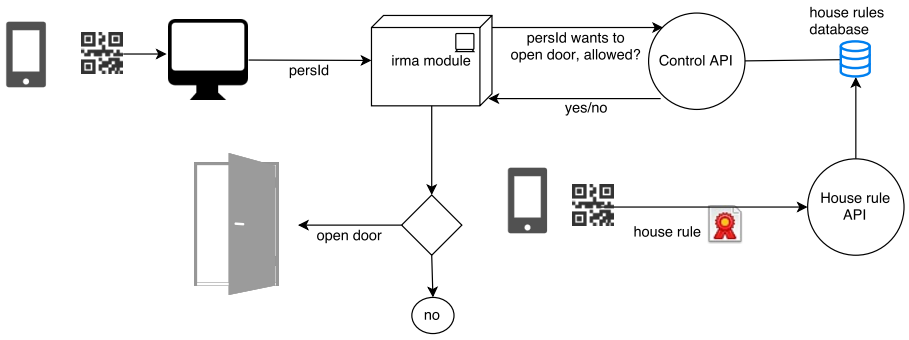


Figure 6.1: Overview of interaction between the different components

aged.

- The occupant fills in the data about the renter/guest that will be granted access to the house. These data are used to create a house rule. In this PoC, we used the name of the renter/guest, the *personID* IRMA attribute, the start and end date and the reason for access.
- The house rules API constructs a house rule from these data and requests the occupant to sign the house rule with his/her *houseID* attribute by showing a QR code.
- The occupant scans this code with his/her IRMA app, verifies the house rule that will be shown in the app, and confirms. The app will send the created IRMA signature back to the house rules API.
- The house ruled API verifies the IRMA signature on the house rule and adds it to the database.

Figure 6.1 shows a schematic overview of these two protocols, using the different components.

6.1.2 Hardware components

Although we separated all the components in software, we still run them on the same hardware device. Since all the components communicate with each other over HTTP, they can be easily 'moved' to another device if necessary. We used the following hardware:

Raspberry Pi V2 This device runs all the software. It contains connectors for both an external screen and an GPIO connector that is used to open the electronic lock.

Electronic lock For demonstration purposes, we used a simple electronic lock, with just two wires. If we put a voltage of 12V on the wires, the lock will close. Otherwise, it remains open.

12 Volt relay To transform the 5V voltage of the Raspberry Pi's GPIO connectors to the 12V voltage of the lock, we need to place a relay in between.

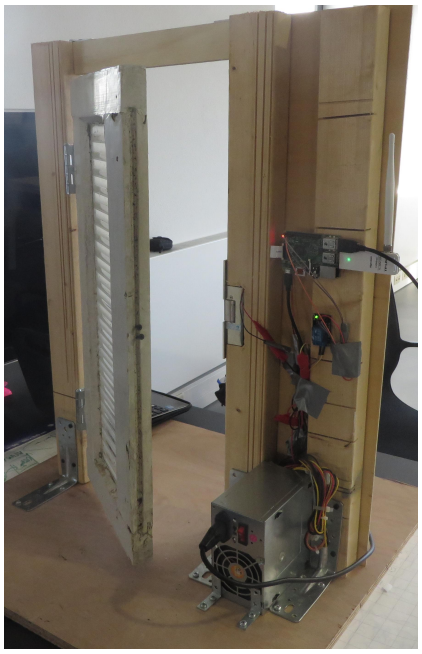


Figure 6.2: Picture of the real PoC hardware

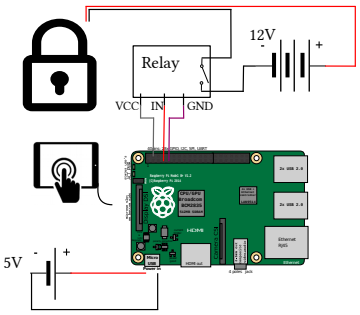


Figure 6.3: Schematic overview of the hardware

Raspberry Pi touchscreen To communicate with the person that is standing in front of the door, a touchscreen is used. This screen displays for instance the QR code that the person has to scan in order to open the door. Although not implemented, it could also be used as a 'door bell' by adding a button 'ring' to the touchscreen.

USB WiFi device The Raspberry Pi creates its own WiFi access point that allows anyone nearby to connect and communicate with the device. In a real scenario, this would be replaced by connecting the Raspberry Pi to the Internet, but for easier demonstration purposes, we chose to use our own network.

Door We used a simple old window as a door, along with some wooden boards to support all the other parts.

We connected all the hardware components to each other, Figure 6.2 shows a picture of our PoC setup, and Figure 6.3 shows a schematic view of the hardware setup.

6.1.3 Used IRMA attributes

In Section 4.1.2 we introduced some basic IRMA credentials that we used in the scenarios. For our PoC, we will stick to these credentials. However, in our PoC scenario, we did not implement the AirBnB part. This part is the only part from the AirBnB scenario where the IRMA concierge discloses an IRMA attribute (see Figure 4.4) from the *CHouse* credential. Because we did not implement this part, we can omit the *CHouse* credential from Table 4.1. By omitting the *CHouse* credential, we restrict ourselves in

IRMA identity	Credential	Attributes
Occupant and renter	Person	personID
		name
	OHouse	houseID
		role

Table 6.1: The IRMA credentials used in our PoC

the PoC to only the *Person* and *OHouse* credentials. Table 6.1 shows these credentials, along with their attributes.

The the *Person* and *OHouse* credential contain the following attributes: *personID*, *name*, *houseID* and *role*. The *personID* and *name* attribute are included in the house rules that provide access to the house. The *personID* attribute is unique, and makes sure that the right person will be added to the house rule. The *name* attribute is used for readability, so an occupant can quickly see which person is included in the house rule. The *houseID* and *role* attributes are used to sign the added house rules, as we explained in Section 4.1.4.

Currently, it is not possible to use the *houseID* attribute to grant access, since in our PoC, we still used the old IRMA protocol that does not support the modern disclosure proof requests with the conjunction of disjunctions (which we explained in Section 5.1.3). With the modern protocol, we could create a disjunction to request either the *personID* or *houseID* attribute in a disclosure proof.

6.2 Protocol

In this section, we show how all the components interact with each other in our implemented protocol. We start by showing the house rules that are used. After this, we specify the protocol flow with UML sequence diagrams. Such diagrams make sure that we also cover the user interaction as extensively as possible.

In our PoC, we implemented two parts of which we described the high-level protocol overview in Section 6.1.1:

- A person that accesses a house by showing his/her *personID* attribute.
- An example 'control panel' that allows the occupant to add house rules that allow a person to access the house.

6.2.1 House rules

In our PoC, we only used one house rule. This rule is used for controlling access to the house:

`actor`	Person with *personID* $= 42$ and *name* $=$ Edward
`action`	is allowed to open
`actee`	the front door of my house
`condition`	between startDate and endDate
`goal`	to enter the house.

This house rule is constructed by the house rules API. After construction, it is stored in the house rules database and evaluated and checked by the control API. We want the house rule system to be as generic as possible, by making it easy to add other house rules for other actors, actions, actees and conditions to the system. Each of the five components of the rule will therefore be stored in a different database column in our house rules database. The sixth column in the database will contain the IRMA signature over the other five columns.

In our PoC, the control API retrieves the values of the *personID* and *name* attributes from the IRMA module (see Figure 6.1). The control API uses this input to check each part of the house rule separately. It will only return *yes* if there is a matching rule. In our PoC, this matching is done in a simple way:

actor The actor is checked by splitting the actor part on spaces, and after this, by splitting on the '=' sign. The results are pairs with attribute names on the left and attribute values on the right. In our example, this leads to (*personID*, 42) and (*name*, Edward).

action In this PoC, we only store the action *open*, which means that we only need to check if this action is equal to *open*.

actee The actee is always 'the front door of my house'. Therefore, we can just verify whether this matches the actee of the rule in the database that is checked.

condition Condition is the hardest part to check. There is no generic function which can catch all types of conditions. Therefore, we restricted ourselves in this PoC to just checking a start date and end date. We check if the current time falls in the period that is described by the start and end date of the house rule.

goal As mentioned before, the goal is not evaluated and only used for communication with the occupant. But, since the complete house rule is signed by an IRMA signature, the goal is included in the signature. This requires a house rule to be signed with a new IRMA signature if the goal is changed.

6.2.2 UML sequence diagrams

We will now describe the two protocols that are used in our PoC, using UML sequence diagrams. As explained before, the IRMA concierge is separated into four main parts: the control API, the house rules API, the house rules database and the IRMA module. The IRMA module also contains the user dialog via a web application that is shown on the Raspberry Pi touchscreen. In order to show the interaction between the user and the IRMA module, we separated the IRMA module and the web application in the UML sequence diagram.

We will start with the first protocol: allowing a renter access to the house. The UML diagram can be found in Figure 6.4 and we will now provide a short description of each UML function:

`send_auth_req` The web application sends a request to the IRMA module to obtain a token.

`send_auth_conf` This token is converted into a QR code and sent to the web application.

`update_screen` The web application displays the QR code on the touchscreen.

`open_app` When the renter arrives at the door to access the house, he/she opens the IRMA app on his/her smartphone.

`scan_screen` The IRMA app scans the QR code from the touchscreen, which connects the IRMA app to the IRMA concierge of the house.

`send_data(connected)` The IRMA app sends a message that it is connected to the IRMA module.

`send_data(req_personID, req_name)` The IRMA module sends a request to the renter's IRMA app to reveal the *personID* and *name* attributes.

`send_req(confirmation)` The IRMA app asks confirmation to the renter to reveal these attributes.

`send_confirmation` The renter agrees with this request (if not, the protocol will be aborted).

`send_data(SD(personID), name)` The IRMA app sends a disclosure proof of the requested attributes to the IRMA module.

`send_req(open, personID, name)` The IRMA module sends a request, containing the keyword '*open*', along with the values of the two revealed IRMA attributes to the control API.

`req_houserule(personID, name)` The control API retrieves all the house rules containing the actor with the used *personID* and *name* from the house rules database.

`send(houseRules)` The house rules database sends all the matched house rules to the control API.

`check(houseRules)` The control API checks all the rules, in the way we described in Section 6.2.1.

`send_conf(yes / no)` The control API sends the result (either *yes* or *no*) to the IRMA module. The IRMA modules relays this result to the web application. Depending on the result, two action can be taken from now.

`send_req(open_door)` If the result is *yes*, then the application sends a signal to the electronic lock to open the door.

`update_screen(green / red)` The application updates the screen to display a green icon if the result was equal to *yes*. Otherwise, a red icon will be displayed on the screen.

`show(green / red)` The IRMA module also sends the result to the IRMA app on the renter's smartphone. This is also done in the form of either a green or red icon.

The second part of the PoC is the control panel, where the occupant allows a renter access to the house. This control panel is for this PoC developed in the form of a web application that is hosted on the IRMA concierge of the house. We assume that the occupant is already authenticated to this web application, which could for instance be done by providing an IRMA disclosure proof. In the future, this web application could be replaced by a 'Tippiq Home App' on the smartphone, or by a portal / touch

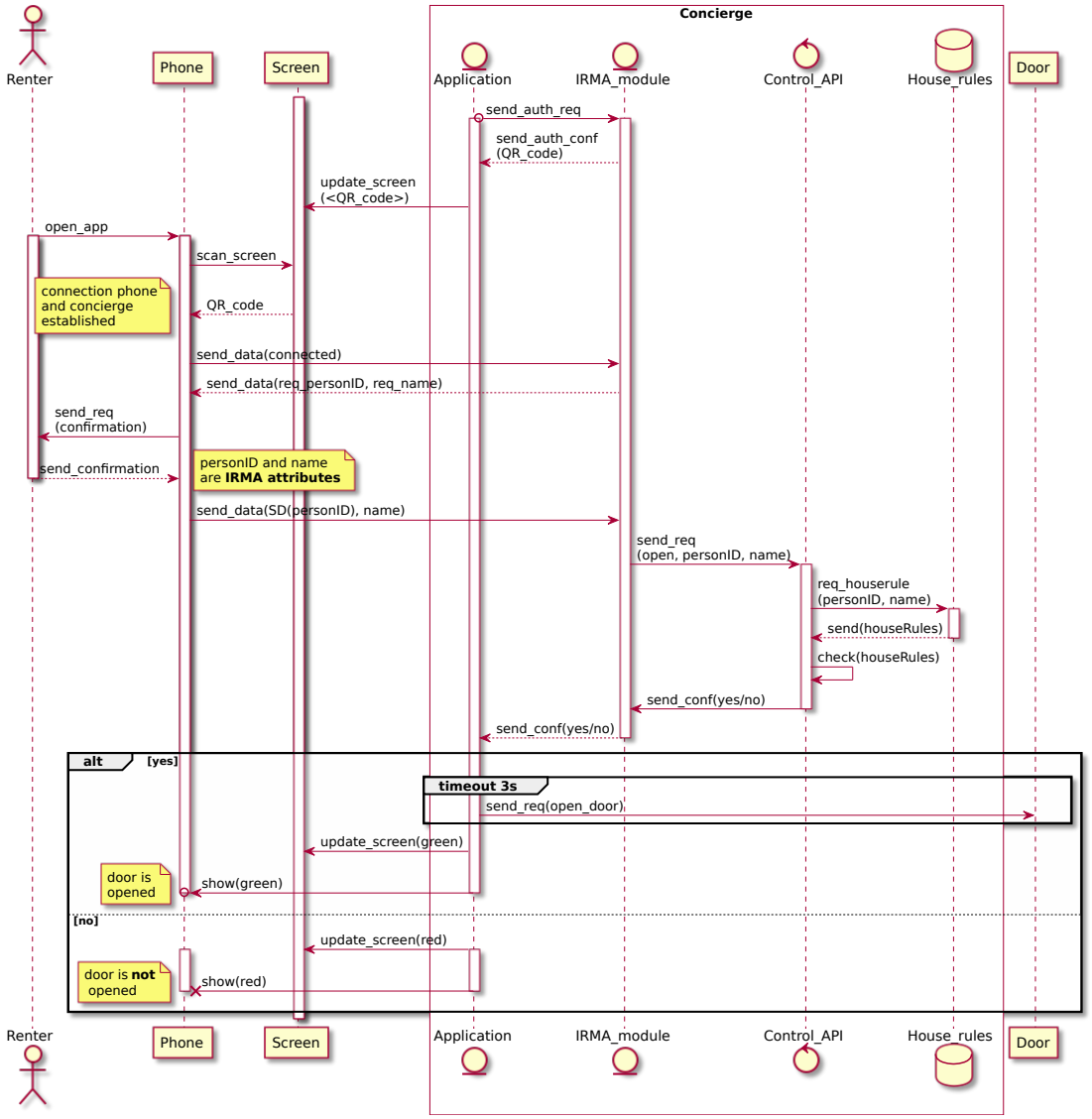


Figure 6.4: UML sequence diagram of a renter accessing the house

screen that is hanging inside the house (allowing everyone in the house to see the house rules).

For the integrity and authenticity of the house rules itself, we use IRMA signatures (as discussed before). This provides an additional layer of security that allows only people with the required attributes to add or change house rules.

The web application is connected to the house rules API. The house rules API adds house rules to the house rules database (see Figure 6.1 for an overview). In the UML diagram, we combined the web application and the house rules API, because otherwise, the house rules API would in the diagram only be relaying messages between the application and the database. However, we separated the screen from the web application/API, to make clear what is shown on the screen to the user of the application. The UML diagram is shown in Figure 6.5, and we will now provide a description of each function from this figure:

`visit_site` The occupant visits the web application portal of his house (as pointed out before, we assume that he/she is already authenticated to open this portal).

`get_house_rules` The house rules API will show all the house rules to the Occupant. In order to this, it needs to retrieve them from the database.

`send_data(houseRules)` The house rules database will return all the house rules after a request. In a real scenario, this could be a huge list. If that will be the case, we probably need to let the house rules API be more specific in which rules it requests, so that we only return the relevant part of the list of rules.

`update_screen(accessList)` The house rules API will display a list of all the access rules on the screen. This allows the occupant to edit or delete them. He/she can also add new house rules in this screen.

`edit_add_rule(houseRule)` The occupant edits or adds a new house rule to the house rules API.

`send_sig_req` This new house rule needs to be signed. For this, the house rules API will issue a signing request to the IRMA module.

`send_sig_conf(QR_code)` The IRMA module replies with a confirmation message, containing a QR code.

`update_screen(QR_code)` The house rules API displays this QR code on the screen.

`open_app` The occupant now needs to open his/her IRMA app to sign the house rule.

`scan_screen` The IRMA apps scans the QR code on the screen, which connects the IRMA app to the IRMA concierge of the house.

`send_data(connected)` The IRMA app sends a message that it is connected to the IRMA module.

`send_data(reqSig_houseRule, req_houseID, req_mainOccupant)` The IRMA module sends a signature request, containing a signature specifica-

tion (as discussed in Section 5.2.3) to the occupant's IRMA app.

`send_req(`*`confirmation`*`)` The IRMA app asks confirmation to the renter to sign the house rule with the included attributes (*houseID* and *mainOccupant*). The complete house rule is displayed on the phone's screen.

`send_confirmation` The occupant agrees with this signing request (if not, the protocol will be aborted).

`send_data(`*`houseRuleSig`*`)` The IRMA app sends an IRMA signature of the house rule to the IRMA module.

`verify(`*`houseRuleSig`*`)` The IRMA module verifies the received signature.

`send_conf(`*`validSig / (invalidSig, houseRuleSig)`*`)` The result of the verification will be sent to the house rules API by the IRMA module. If the signature is valid, the IRMA module also sends the signature itself to the house rules API.

`edit_update(`*`houseRule, houseRuleSig`*`)` If the signature is valid, then the house rules API sends the house rule and corresponding IRMA signature to the house rules database.

`send_conf(`*`updated / changed`*`)` The house rules database sends a confirmation of the update request to the house rules API. This can either be updated, if the database is updated or changed if a house rule that already exists is changed.

`update_screen(`*`green / red`*`)` The house rules API updates the screen with a green icon if the house rule that the occupant wants to add is indeed added (and the signature is valid). Otherwise, a red icon with an error message will be displayed on the screen.

`show(`*`green / red`*`)` The IRMA module also sends the result to the IRMA app on the occupant's smartphone. This is also done in the form of either a green or a red icon with an error message. This error message is only sent in this protocol (adding a house rule), and not in the previous protocol from Figure 6.4. We do this, because in the previous protocol, everyone can try to open the door, and in that case we want to be as less verbose as possible about error messages.

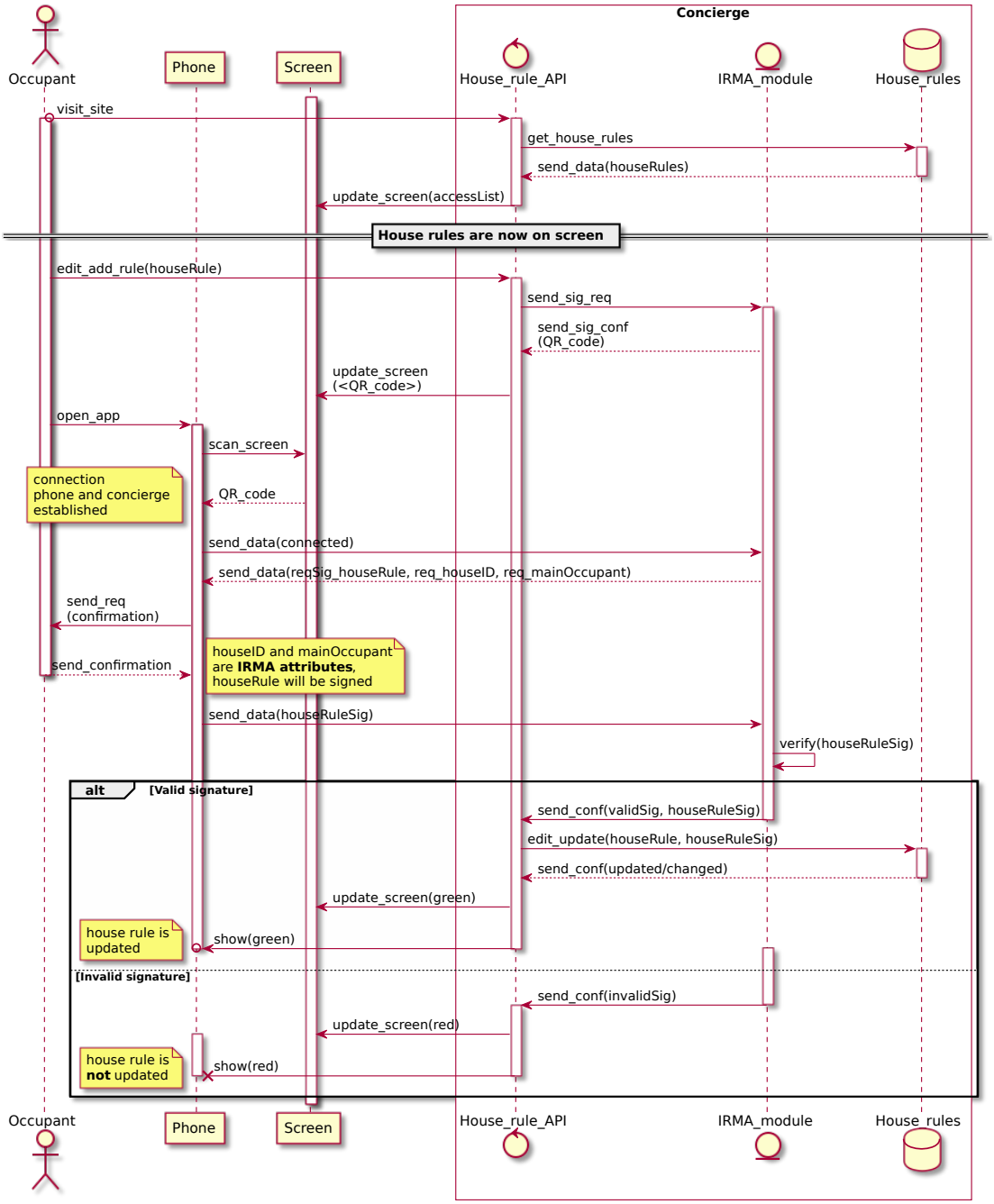


Figure 6.5: UML sequence diagram of an occupant providing a renter access to the house

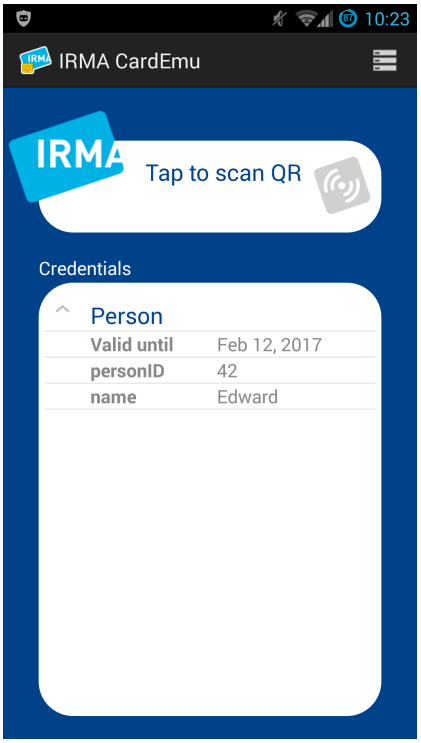


Figure 6.6: Overview of attributes of the renter that rents the house

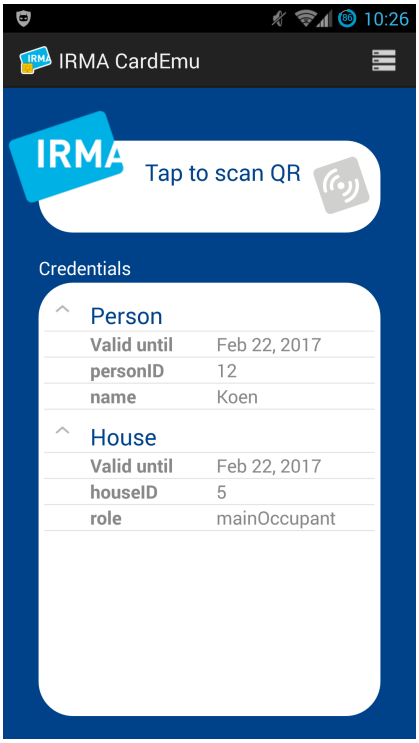


Figure 6.7: Overview of attributes of the occupant

6.2.3 Screenshots

In this section, we will illustrate the implemented parts or our PoC using screenshots. We will show the three parts, namely the attributes overview of the IRMA app, the flow of entering the house and the flow of adding and signing a new house rule.

Attributes overview

The first part is the attributes overview. All the used attributes are listed in Table 6.1. Figure 6.6 shows the attributes of the renter. He has a *personID* with value 42, and a *name* attribute with value 'Edward'. Both attributes are contained in the *Person* credential. For simplicity and because it is not used in our PoC, we did not add a *OHouse* credential to this person's IRMA token.

The second figure (Figure 6.7) shows the attributes of the occupant. The occupant has the *name* 'Koen' and the *personID* 12. The house that he will rent out to the renter has *houseID* 5 and the occupant is the main occupant of this house, as shown in the *role* attribute.

Renter enters the house

The second part is about entering a house. As we introduced in Section 6.1.2, we used an old window as door, with the Raspberry Pi touchsceen attached to this window. Figure 6.8 shows two pictures of this window. On the top it shows a picture of the complete door, and on the bottom a detailed picture of the touchscreen, where a QR code is displayed. Everyone that wants to open the door to enter the house has to scan this QR code. The pictures of the real PoC hardware are in Dutch, because Tippiq operates at this moment only in The Netherlands.

After scanning the QR code, one has to reveal his/her *personID* attribute (Figure 6.9). This attribute will be checked by the IRMA concierge using the house rules. If the person is allowed access, Figure 6.10 will appear on the touch screen.

Occupant adds house rule

The third and final part is the management portal, where the occupant can add new house rules. We implemented a very basic version of this portal in the form of a web application. When the occupant visits this application, he/she sees an overview of the house rules. Figure 6.11 shows a screenshot of this overview, with one enabled house rule that allows the renter Edward access to the house.

After changing or adding a house rule, the new rule has to be signed. For this, we need to scan a QR code, as shown in Figure 6.12. After scanning, the occupant has to agree on his/her phone to sign this house rule. The complete house rule is displayed on the phone, so the occupant knows what he/she is signing. Figure 6.13 shows an example confirmation dialog.

If the house rule is signed correctly, the screen from Figure 6.14 will be displayed and the house rule is added to the database.

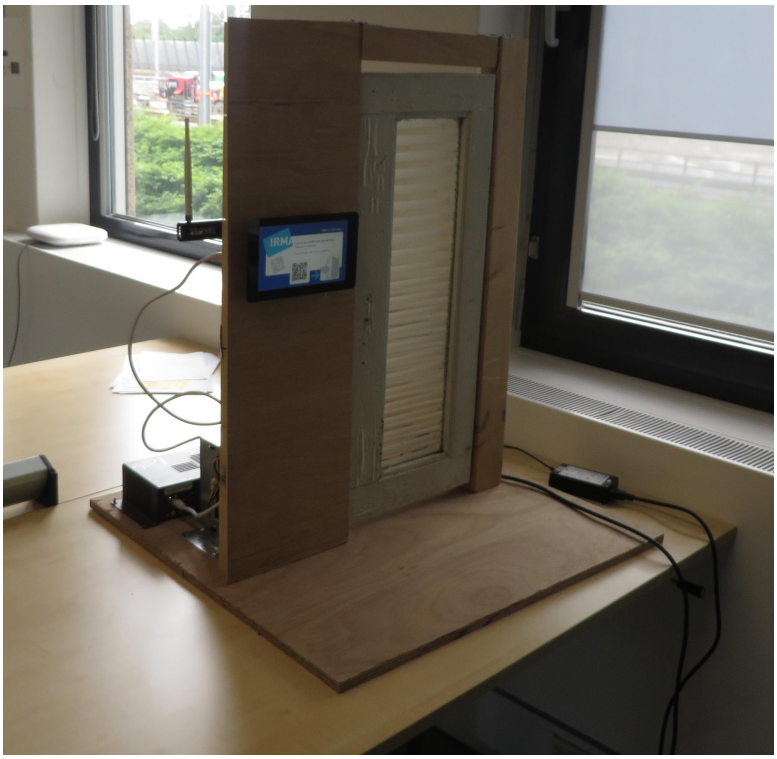


Figure 6.8: Our PoC with on the top a picture of the complete setup with door and touchscreen, and on the bottom a detailed picture of the touchscreen

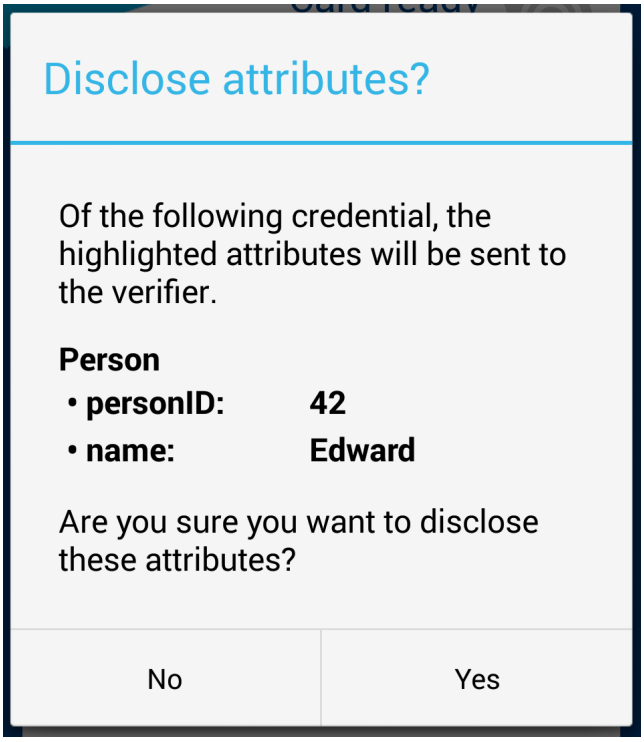


Figure 6.9: A person has to reveal his/her *personID* attribute, in order to open the door.

Figure 6.10: If there is a matching house rule containing the revealed *personID* attribute, then this screen will be displayed and the door will be opened.

Figure 6.11: The management portal with one house rule for the house with *houseID* 5

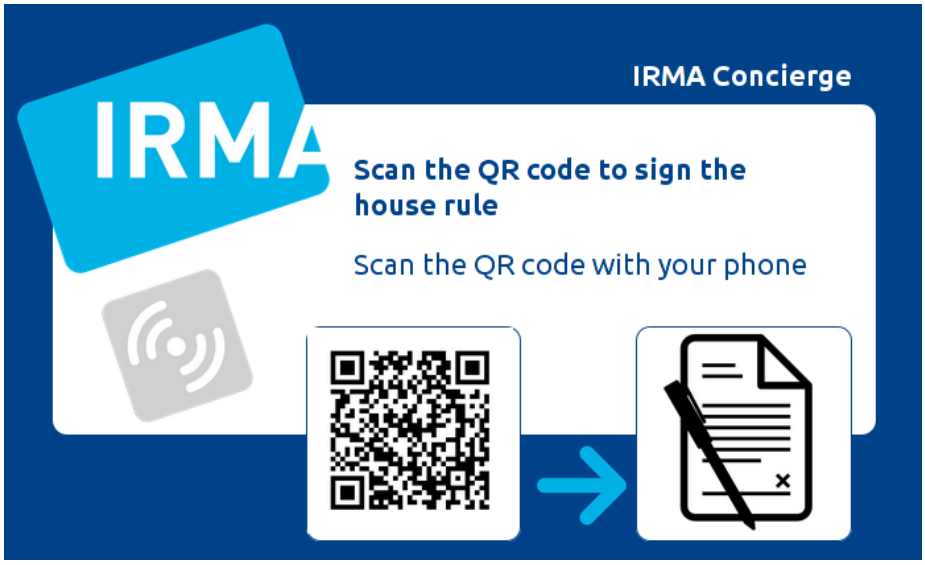


Figure 6.12: This QR code needs to be scanned to sign the new house rule.

Figure 6.13: Before the house rule is signed, it is displayed on the occupant's screen. The occupant also sees with which attributes the house rule will be signed. The occupant has to agree in order to sign this message. The message is a Dutch translation of the house rule that we introduced in Section 6.2.1.

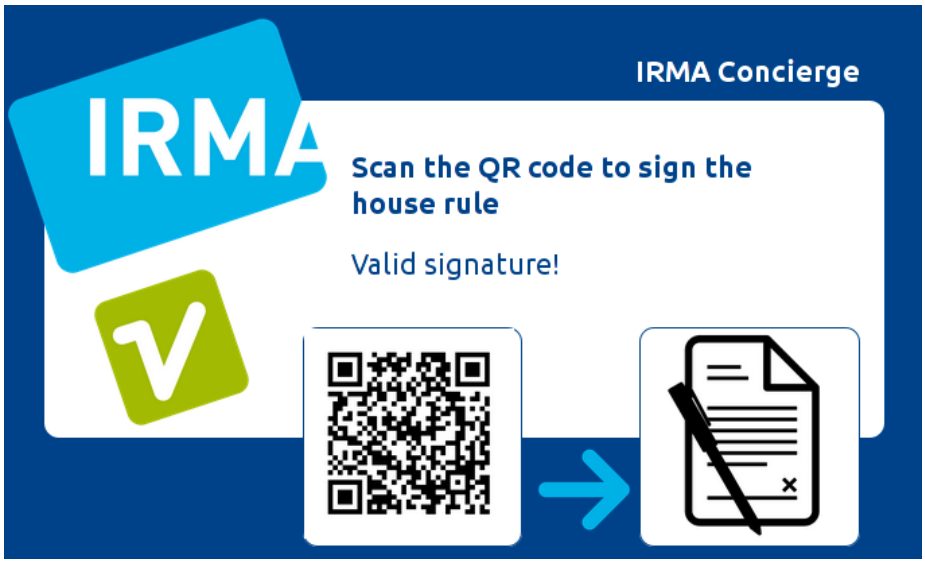


Figure 6.14: If the house rule is signed correctly, it will be added to the database and this screen will be displayed.

6.3 Possible improvements

In this PoC, we demonstrated how IRMA can work in an IoT context, where a renter is provided access to the house by disclosing an IRMA attribute, which is in real-time checked with the house rules by the IRMA concierge. While there are many improvements possible, we will point out a few important ones that we missed in our PoC.

One important improvement is an audit log. It is important that an occupant can check who at a certain timestamp has accessed the house. In the case of problems, this could really help to solve conflicts.

Another important part we mentioned earlier is the integration with Airbnb. If we could cooperate with them (or a similar party), then we could build a complete protocol that contains all the actions.

At this moment, access control relies solely on the *personID* attribute, which is rather limited. For instance, occupants with the *houseID* attribute of the the house should also be allowed access. We could even extend this by allowing 'access classes' of persons access which can combine our PoC with the scenarios of Chapter 4. In this way, we can for instance allow firefighters and house cleaners to open the door when certain conditions are met.

Finally, the UI can be improved and extended. An option to digitally 'ring' the bell could for instance be added to the touchscreen. Also, the management portal should be improved and for instance be converted into a standalone smartphone application.

6.4 Source code

This proof of concept is still built upon the old IRMA protocol stack (using a deprecated protocol based on smart card APDUs). While we implemented IRMA signatures on both the old IRMA protocol and the modern IRMA protocol, we did not do this with our PoC.

Because we do not want anyone to use this old protocol stack and code, we decided to not publish the source code of our PoC on Github. Therefore, we have made all the source code only available in a ZIP file at the following link: `https://vps.koenvaningen.nl/files/2016/08/64azeQUHHfso.zip`. That link is only provided for reference. The code at that link should never be used in production. If someone wants to use IRMA signatures, we recommend him/her to look at our IRMA signature implementation for the modern IRMA protocol. In Section 5.2.8, we provide links to the source code of that implementation.

Chapter 7

Conclusions

In this thesis, we looked into how one can use and adapt the IRMA technology for an IoT and house context. We did this by adding a central gateway to the house, called the IRMA concierge. This device can contain its own IRMA attributes. Instead of only giving 'real' persons attributes, we extended IRMA by also giving an IoT device that represents a home IRMA attributes. This device is therefore multi-functional. It acts as both an IRMA user, by disclosing attributes to other parties, as well as an IRMA verifier, by verifying attributes from other parties.

To allow fine-grained access control to the house itself, the devices in the house and the data in the house, we used Tippiq house rules. With several scenarios, we showed in Chapter 4 that this concept can work in different concrete IoT scenarios if we combine house rules with IRMA technology where all the parties (IRMA concierge, occupants and other third parties) possess an IRMA token with IRMA attributes. The identity of all these parties is built up using only IRMA attributes, which allows them to reveal only the required attributes needed for a particular action. We saw that it can be really useful to provide a house with IRMA attributes, which for instance allow external parties to verify an address of a house. Also, occupants can be coupled to a house by providing them the same *houseID* attribute that we also provided to the IRMA concierge of their house.

Since Tippiq house rules form the basis for access control in the house, we want to be sure that the integrity and authenticity of these rules are guaranteed. For this, we sign them digitally. Since we build the identities of all the parties with only IRMA attributes, we decided to use these attributes to sign house rules. This is done with IRMA signatures, a concept from the literature that we described in Section 3.4. We showed that besides signing house rules, IRMA signatures can be used for more scenarios. For instance, signing a contract and allowing external parties access to data and to attributes from the IRMA concierge.

IRMA signatures only existed in the literature. Because we wanted to use them in our scenarios, we decided to propose a possible specification for them that is compatible with the current IRMA verification and issuance protocols. We also implemented this specification, and described the details of our specification and implementation in Chapter 5. We saw that we can make an important distinction between IRMA signatures with *self-selected* attributes and IRMA signatures with *pre-selected* attributes. We

also had to make sure that a nonce from an IRMA signature session can never be used in an IRMA verification/authentication session and the other way around.

Based on the scenarios from Chapter 4 and using our implementation of IRMA signatures from Chapter 5, we implemented a proof of concept that integrates all the concepts into a practical service in Chapter 6. We divided the complete application in separate components, which allows an easy integration with the Tippiq project. With this proof of concept, we proved that IRMA can really work in a realtime IoT scenario using Tippiq house rules. It showed that an attribute-based identification and authentication system like IRMA is very flexible and that it can be used in a broader setting than only identifying people in a privacy-friendly way.

Bibliography

[1] IBM Research Zürich Security Team. Specification of the identity mixer cryptographic library, version 2.3.4, 2012.

[2] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang. Smart grid - the new and improved power grid: A survey. *Communications Surveys & Tutorials, IEEE*, 14(4):944–980, 2012.

[3] Ulrich Greveler, Peter Glösekötterz, Benjamin Justusy, and Dennis Loehr. Multimedia content identification through smart meter power usage profiles. In *Proceedings of the International Conference on Information and Knowledge Engineering (IKE)*, 2012.

[4] Porkodi and Velumani Bhuvaneswari. The Internet of Things (IoT) applications and communication enabling technology standards: An overview. In *Intelligent Computing Applications (ICICA), 2014 International Conference on*, pages 324–329, 2014.

[5] Brinda Hampiholi, Gergely Alpár, Fabian van den Broek, and Bart Jacobs. Towards practical attribute-based signatures. In *Security, Privacy, and Applied Cryptography Engineering*, pages 310–328. Springer, 2015.

[6] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[7] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology–CRYPTO'86*, pages 186–194. Springer, 1986.

[8] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *International Conference on Security in Communication Networks*, pages 268–289. Springer, 2002.

[9] Guo Shaniqng and Zeng Yingpei. Attribute-based signature scheme. In *Information Security and Assurance, 2008. ISA 2008. International Conference on*, pages 509–511. IEEE, 2008.

[10] Audun Josang, Dean Povey, and Anthony Ho. What you see is not always what you sign. *Proceedings of the AUUG2002*, 2002.

[11] Dan-Sabin Popescu. Hiding malicious content in PDF documents. *arXiv preprint arXiv:1201.0397*, 2012.

[12] Burton S Kaliski Jr. A layman's guide to a subset of ASN.1, BER, and DER. 1993.

[13] Wouter Lueks, Gergely Alpár, Jaap-Henk Hoepman, and Pim Vullers. Fast revocation of attribute-based credentials for both users and verifiers. In *ICT Systems Security and Privacy Protection*, pages 463–478. Springer, 2015.

[14] PUB Fips. 186-2. digital signature standard (DSS). *National Institute of Standards and Technology (NIST)*, 2000.

[15] Jakob Jonsson and Burt Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, RFC Editor, 2003.

[16] Russ Housley. Cryptographic Message Syntax. RFC 5652, RFC Editor, 2009.