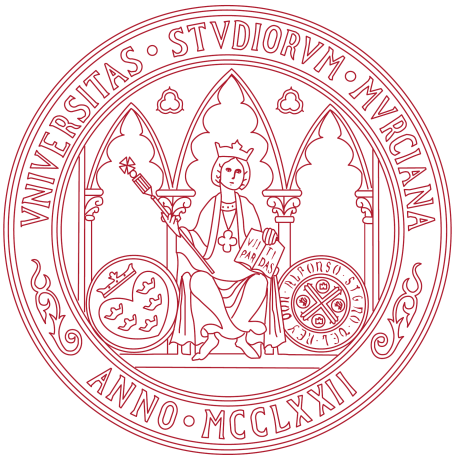
INTEGRACIÓN DE IDEMIX EN ENTORNOS DE IOT

JOSÉ LUIS CÁNOVAS SÁNCHEZ

Tutores
ANTONIO FERNANDO SKARMETA GÓMEZ
JORGE BERNAL BERNABÉ



Facultad de Ingeniería Informática
Universidad de Murcia

José Luis Cánovas Sánchez: *Integración de Idemix en entornos de IoT*

Junio 2017

ABSTRACT

As part of IBM's academic grant for *Privacy Preserving Identity Management applied to IoT*, we designed a solution to integrate IBM's Identity Mixer into the IoT ecosystem, in a privacy-preserving fashion thanks to the use of Zero-Knowledge Proof cryptography. To validate our design, we implemented a functional Proof of Concept written in C for IoT devices. The applications of such solution promise to improve both the security and privacy of IoT, which in recent years have proved to be compromised.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

IoT	Internet of Things
ZKP	Zero-Knowledge Proof
P2ABCE	Privacy-Preserving Attribute-Based Credentials Engine
PoC	Proof of Concept
APDU	Application Protocol Data Unit
BLOB	Binary Large OBject
CoAP	Constrained Application Protocol

INTRODUCCIÓN

El término Internet de las Cosas, o *Internet of Things* (IoT) en inglés, tiene un amplio rango de interpretaciones [1], pero podemos resumirlo en miles de millones de dispositivos, en su mayoría de capacidad computacional limitada, que están interconectados entre sí e Internet, a fin de realizar algún objetivo común. Por ejemplo, una red de farolas en una calle de la ciudad, donde cada farola tiene un sensor de proximidad, y se comunica con las siguientes farolas para iluminar antes la calle, mejorando la seguridad y ahorrando energía; invernaderos con riego automatizado en base a los sensores de humedad repartidos por la tierra, mejorando la calidad de la plantación, y ahorrando agua; coches comunicándose entre sí para dar aviso de accidentes a los que vienen detrás, o indicando rutas alternativas para minimizar los atascos y la contaminación dentro de la ciudad.

Muchos de estos objetivos requieren recopilar una gran cantidad de datos por medio de sensores, y gracias a organizaciones como WikiLeaks, los usuarios son cada vez más conscientes de la implicación que suponen sus datos en Internet, y demandan más seguridad y privacidad para ellos. Esto incluye tanto los datos que los usuarios comparten, sino también los datos recopilados sobre nosotros y sobre los que no tenemos un control directo. El ataque informático sufrido por Sony en 2011[1] afectó a su red de jugadores de consola, cuya información de facturación estaba guardada en sus servidores, y debido a que dicha información dependía tanto de los usuarios como de Sony para estar segura, aunque los usuarios la protegieran, se vieron comprometidos igualmente.

Con la proliferación de dispositivos IoT recopilando tanta información como son capaces con sus sensores, la cantidad de datos recopilados sobre cualquier persona puede ser enorme. Y el IoT ha demostrado no ser de confianza para la seguridad ni la privacidad, como en el reciente ataque DDoS de la *botnet* Mirai, en octubre de 2016, considerado el ataque DDoS más grande de la historia[18], o las múltiples vulnerabilidades descubiertas en aparatos del hogar, como monitores de bebé[22].

Para solucionar este problema de privacidad en Internet, aparece el concepto de *anonimidad fuerte*, que oculta nuestros datos personales, pero que nos permite poder seguir operando en Internet como individuos independientes[11]. Para conseguir esto, debemos realizar los procesos de autenticación y autorización de la manera más privada posible. Las credenciales basadas en atributos y la *selección selectiva* de

1 2011 PlayStation Network outage - https://en.wikipedia.org/wiki/2011_PlayStation_Network_outage

los mismos, permite controlar qué información revelamos a terceros, bajo un sistema criptográfico de confianza para ambas partes.

Una credencial basada en atributos puede entenderse como una firma digital, realizada por un Emisor, sobre una lista de pares atributo-valor, por ej., la lista `(nombre=Alice, apellido=Anderson, fechaNac=1977-05-10, nacionalidad=DE)`[19].

El método más directo, como Usuario, para convencer a un Verificador de que se poseen los valores firmados en el certificado, sería simplemente enviar la credencial en claro, mostrando todos los atributos. Con credenciales anónimas, el Usuario nunca transmite la credencial en sí, sino que usa una criptografía especial para convencer al Verificador de que sus atributos cumplen alguna propiedad, sin mostrarlos directamente, por ejemplo, que la diferencia entre la fecha actual y la fecha de nacimiento en el certificado es mayor a 18 años. Una ventaja directa es que el Verificador no puede reutilizar esa prueba con terceras partes, pues no posee el certificado, lo que evita el robo de información.

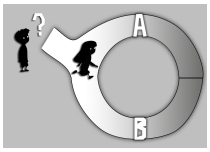
Con la criptografía simétrica o asimétrica usual utilizada en los certificados más populares, como los TLS/SSL para las webs de Internet, no es posible llevar a cabo esas *pruebas* sin una explosión combinatoria de firmas de certificados de todas las posibles pruebas a presentar. Las soluciones para la privacidad actuales utilizan las llamadas Pruebas de Conocimiento Cero (ZKP), cuya teoría estudio más en profundidad en mi TFG de Matemáticas[16], y en resumen, permiten demostrar conocimiento de algún dato secreto, sin revelar nada más que el hecho de conocer dicho dato, de ahí la parte de *conocimiento cero*.

Para explicar brevemente cómo las ZKP funcionan, en 1990 Guillou, Quisquater y Berson publicaron *How to Explain Zero-Knowledge Protocols to Your Children* [15], una historia sobre cómo Alí Babá demostró que sabía las palabras mágicas para abrir la cueva, pero sin decirle a nadie dichas palabras. A continuación presentamos una versión reducida, pero que subraya las propiedades básicas de toda ZKP.

To understand how ZKPs work, in 1990 Guillou, Quisquater and Berson published in *How to Explain Zero-Knowledge Protocols to Your Children* [15] a story about how Ali Baba proved that he knew the magic words to open the cave, but without revealing those words to anyone. Here we present a brief version highlighting the properties of ZKPs.

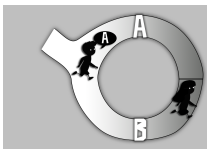
> Imaginemos una cueva, donde el camino principal se bifurca y al final de cada pasillo los caminos se vuelven a encontrar, formando una especie de anillo. En el punto en que se unen, dentro de la cueva, hay una puerta mágica con una palabra secreta, la cual permite abrirla y cruzar al otro lado.

Paula conoce la clave secreta y quiere **p**robarlo a su amigo Víctor, pero sin tener que revelársela. Paula y Víctor quedan en la entrada de la cueva con unos *walkie-talkies*, de modo que Víctor esperará fuera y Paula entrará a la cueva y tomará uno de los pasillos, que llamaremos A y B, sin decirle cuál a Víctor.



La cueva [17] . Paula entra por A o B al azar. Víctor espera fuera.

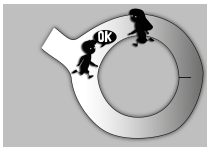
Al llegar a la puerta, Paula avisa a Víctor para que entre a la cueva y espere en la bifurcación, donde **V**íctor, para intentar **v**erificar que Paula conoce la palabra secreta, le indicará por qué pasillo quiere que vuelva, el A o el B.



La cueva. Víctor elige al azar por dónde quiere que regrese Paula.

Si Paula realmente conoce el secreto, podrá volver a la bifurcación por el pasillo solicitado, abriendo, si es preciso, la puerta. Pero en caso de no conocer la clave, al entrar por uno de los pasillos, tenía una probabilidad del 50% de adivinar cuál pediría Víctor.

Víctor no se queda contento con una sola prueba, pues Paula podría haber tenido suerte, así que la repiten hasta que se convence. Con unas 20 repeticiones, Paula tendría solo una probabilidad de 2^{-20}, prácticamente nula, de acertar todas las veces y engañar así a Víctor.



La cueva. Paula vuelve por el camino pedido.

Eva, curiosa de qué hacían Víctor y Paula en la cueva, **e**spía a Víctor durante todo el proceso. Eva no sabe si Paula y Víctor han acordado previamente qué pasillo pedir por el *walkie-talkie*, y sólo Víctor está seguro de que él los estaba eligiendo al azar y sin previo acuerdo.

Más tarde Eva habla con Víctor, que está seguro de que Paula conoce la clave, y éste querría convencer también a Eva, pero como él no conoce la clave, no puede repetir la prueba a Eva, sólo Paula puede realizarla con éxito.

Utilizando las ZKP, IBM ha desarrollado el protocolo Identity Mixer[2], Idemix para acortar, para autenticación y transferencia de atributos certificados preservando la privacidad. Idemix permite a un usuario autenticarse sin divulgar ningún dato personal. Cada usuario recibe un certificado con múltiples atributos, pero a la hora de presentarlo, pueden decidir qué atributos revelar o de cuáles dar prueba de alguna propiedad, como la de la mayoría de edad, pertenencia a un país de la Unión Europea, etc. De este modo, los datos como su fecha de nacimiento, o país específico no son recopilados y no necesitan ser protegidos, ni procesados por terceros.

"Si tus datos personales nunca son recopilados, no pueden ser robados."

2 Identity Mixer - https://www.research.ibm.com/labs/zurich/idemix/

Hasta ahora, Idemix o credenciales privadas basadas en atributos han conseguido lidiar con los escenarios típicos de Internet, donde los usuarios pueden autenticarse frente a un proveedor de servicios, demostrando que poseen un credencial válido con permisos, pero sin dar más información. Veremos en el capítulo del estado de arte que las implementaciones actuales están realizadas en Java, lo cual requiere unos recursos mínimos para su ejecución, y, hasta donde llega nuestro conocimiento, este trabajo es la primera propuesta de implementar una solución para la privacidad en IoT, basada en sistemas de credenciales anónimas como Idemix.

El objetivo de este proyecto es integrar Idemix con el Internet de las Cosas. Para ello nos apoyaremos en el proyecto europeo de ABC4Trust, Privacy-Preserving Attribute-Based Credentials Engine (P2ABCE)[3], que define una arquitectura común, lenguaje y artefactos para unificar distintas soluciones como Idemix de IBM, o U-Prove de Microsoft. Con esto todo dispositivo IoT que lo implemente será compatible con el ecosistema existente.

Una vez los dispositivos pueden ejecutar Idemix, el siguiente paso es utilizar esta herramienta en otros proyectos. Por ejemplo, en un edificio inteligente, distintas políticas de privacidad podrían combinar la protección de los datos recogidos con la seguridad, como el número de personas en el edificio, al termostato sólo le debe interesar si hay más de un mínimo para encender más aparatos de aire acondicionado, pero en caso de incendio, el cuerpo de bomberos puede tener acceso a los datos ocultos por los sensores, para conocer el paradero de las personas atrapadas. Para llegar a soluciones de ese tipo, el primer paso será integrar Idemix en sistemas de gestión de identidad de IoT existentes, como el proyecto FIWARE.

Distribución del trabajo

Este documento se estructura como sigue: En el capítulo 2 mostramos el estado del arte mediante la historia de Idemix y trabajos relacionados, analizando los aspectos más interesantes de cara al IoT; luego, en el capítulo 3 destacamos los objetivos del proyecto y analizamos en detalle las tecnologías existentes con las que trataremos; en el capítulo 4 describiremos el diseño formal de nuestra solución para IoT e Idemix, y describiremos la implementación de la prueba de concepto; tras esto, en el capítulo 5 validaremos la implementación y realizaremos tests para comprobar su viabilidad; finalmente, describimos nuestras conclusiones y líneas de trabajo futuro en el capítulo 6.

3 P2ABCEngine https://github.com/p2abcengine/p2abcengine

1 INTRODUCTION

The Internet of Things (IoT) is a term with a wide range of interpretations [1], breafly, we can think of it as billions of devices, mainly resource constrained, that are interconnected between them, and the Internet, in order to achieve a goal. For example, a network of lamposts with proximity sensors that talk to each other so they light up part of the street when a passerby walks by, but save energy when not; greenhouses with automated irrigation to balance costs and quality vegetables; or cars exchanging traffic data to reduce city pollution from traffic jams or avoid accidents.

Many of this objectives require the use of a great amount of data, and thanks to organizations like WikiLeaks, people are aware of the implications of their data on the Internet, demanding more security and privacy for it. This includes not only the data shared with others, where one must trust they will keep it safe, but it's also the data collected about us and that we don't have direct control over it. The attack Sony suffered in 2011 to PSN[1] is an example of the trust people had in Sony to store their billing information, and because the security of that information depended on both them and Sony, people found themselves compromised unable to do anything about it.

With the proliferation of IoT devices gathering as much information as they can with their sensors, the amount of data gathered about anyone can be immense. And IoT has proved to not address neither security nor privacy, with recent events like the Mirai botnet DDoS attack on October 2016, considered the biggest DDoS in history [18], or the multiple vulnerabilities affecting house devices, like baby monitors [22].

To address this problem of privacy in the Internet, a recent approach is the concept of *strong anonymity*, that conceals our personal details while letting us continue to operate online as a clearly defined individuals [11]. To achieve it, we must address a way to perform authentication and authorization in the most privacy-friendly approach. Attribute-based credentials and *selective disclosure* allow to control what information we reveal, under a trusted environment.

Intuitively, an attribute-based credential can be thought of as a digital signature by the Issuer on a list of attribute-value pairs, e.g. the list `(fname=Alice, lname=Anderson, bdate=1977-05-10, nation=DE)` [19]. The most straightforward way for the User to convince a Verifier of her list of attributes would be to simply transmit her credential

1 2011 PlayStation Network outage - https://en.wikipedia.org/wiki/2011_PlayStation_Network_outage

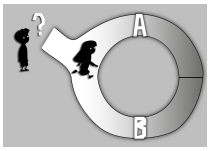
to the Verifier. With anonymous credentials, the User never transmits the credential itself, but rather uses it to convince the Verifier that her attributes satisfy certain properties – without leaking anything about the credential other than the shown properties. This has the obvious advantage that the Verifier can no longer reuse the credential to impersonate `Alice`. Another advantage is that anonymous credentials allow the User to reveal a selected subset of her attributes. Stronger even, apart from showing the exact value of an attribute, the User can even convince the Verifier that some complex predicate over the attributes holds, e.g. that her birth date was more than 18 years ago, without revealing the real date.

With usual symmetric and asymmetric cryptography it seems impossible to create such credentials, without an explosion of signatures over every possible combination. For this reason, current solutions rely on Zero-Knowledge Proofs (ZKPs) , cryptographic methods that allow to proof knowledge of some data without disclosing it.

To read more about ZKPs, refer to my Mathematics thesis [16]

To understand how ZKPs work, in 1990 Guillou, Quisquater and Berson published in *How to Explain Zero-Knowledge Protocols to Your Children* [15] a story about how Ali Baba proved that he knew the magic words to open the cave, but without revealing those words to anyone. Here we present a brief version highlighting the properties of ZKPs.

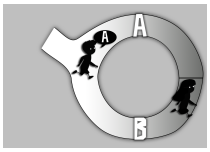
Imagine a cave, where the path forks in two passages, and at the end of each one, they join again, with the shape of a ring. In the point the paths meet, there is a magic door, that only opens when someones pronounces the magic work.



The cave [17] . Peggy takes randomly A or B. Victor awaits outside.

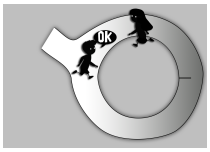
Peggy knows the secret word and wants to **p**rove it to her friend, **V**ictor, but without revealing it. Peggy and Victor meet at the entrance of the cave, then Victor awaits while Peggy goes inside the cave, taking one of the passages, that we will name A and B. Victor can't see which way Peggy went.

When Peggy arrives at the door, **V**ictor enters the cave, and when he arrive to the fork, stops and yells which path, A or B, he wants Peggy to come back, to **v**erify she knows how to open the door.



The cave. Victor chooses randomly the returning path for Peggy.

If Peggy actually knows the secret, she always can take the requested path, opening the magic door if needed. But if Peggy doesn't know the magic word, she had a chance of 50% to guess correctly what passage Victor was going to ask. That means she had a chance to fool Victor.



The cave. Peggy returns by the requested path.

> Victor then asks to repeat the experiment. With 20 repetitions, the chances Peggy fools Victor in all of them is only 2^{-20},
>
> **E**ve, curious about what Victor and Peggy were doing in the cave, **e**avesdrops Victor during the process. The problem is that Eve doesn't know if Peggy and Victor agreed on what paths to choose, because they wanted to prank her for being busybody. Only Victor is confident he is choosing the returning passage randomly.
>
> Later, Victor is convinced that the door can be opened and Peggy knows the word, but he can't prove it to Eve because he can't open the door.

Based on ZKP properties, IBM has developed the Identity Mixer[2], Idemix for short, protocol suite for privacy-preserving authentication and transfer of certified attributes. It allows user authentication without divulging any personal data. Users have a personal certificate with multiple attributes, but they can choose how many to disclose, or only give a proof of them, like being older than 18 years-old, living in a country without revealing the city, etc. Thus, no personal data is collected that needs to be protected, managed, and treated.

"If your personal data is never collected, it cannot be stolen."

So far, Idemix or privacy-ABCs have been successfully applied to deal with traditional Internet scenarios, in which users can authenticate and prove their attributes against Service provider. However, due to the reduced computational capabilities of certain IoT devices, it has not been yet considered for IoT scenarios. As we study in the state of the art chapter, current implementations are based on Java, which requires high computational and memory resources to be executed, and to the best of our knowledge, this is the first proposal that tries to apply an IoT solution for privacy-preserving authentication and authorization, based on Anonymous credential Systems, like Idemix.

The goal of this project is to integrate Idemix with the IoT. It will be done using the ABC4Trust's Privacy-Preserving Attribute-Based Credentials Engine (P2ABCE), a framework that defines a common architecture, policy language and data artifacts for an attribute based ecosystem, cryptographically based on either IBM's Idemix or Microsoft's U-Prove [3]. This gives us a standardized language to exchange Idemix's messages between IoT devices and any other P2ABCE actor.

Once the IoT devices can execute Idemix, the new step is to take advantage of it in other deployments. In a smart building, different

2 Identity Mixer - https://www.research.ibm.com/labs/zurich/idemix/
3 P2ABCEngine https://github.com/p2abcengine/p2abcengine

privacy policies could protect sensitive data, like how many people there are, and their identities, where a thermostat would only need to know if the amount of people is between some boundaries, but in case of emergency, the police department could request all the information available. We think that the first approach to achieve these ideas is first to integrate Idemix in known IoT identity systems, like FIWARE project.

1.1 OUTLINE OF THIS THESIS

This document is structured as follows: In Chapter 2 we show a state of the art analysis through the history of Idemix and related works, analysing what is of the most interest for the IoT perspective; then, in Chapter 3 we outline the project's objectives and analyse in depth the existing solutions that we are going to deal with; in Chapter 4 we describe the formal design of the IoT and Idemix solution, and describe the PoC implementation developed; after an implementation, it is a must to validate it, as showed during the performance tests in Chapter 5; finally, our conclusions and lines for future work are described in Chapter 6.

2

STATE OF THE ART

In this project we will study Identity Mixer as an Attribute-Based Credentials (ABC) solution for privacy-preserving scenarios, but there exist other solutions competing to give the best performance and capabilities as possible. The two most notable alternatives to Idemix are Microsoft's U-Prove and Persiano's ABC systems.

Persiano and Visconti presented a non-transferable anonymous credential system that is multi-show and for which it is possible to prove properties (encoded by a linear Boolean formula) of the credentials [14]. Unfortunately, their proof system is not efficient since the step in which a user proves possession of credentials (that needs a number of modular exponentiations that is linear in the number of credentials) must be repeated times (where is the security parameter) in order to obtain a satisfying soundness.

Stefan Brands provided the first integral description of the U-Prove technology in his thesis [4] in 2000, after which he founded the company Credentica in 2002 to implement and sell this technology. Microsoft acquired Credentica in 2008 and published the U-Prove protocol specification [3] in 2010 under the Open Specification Promise4 together with open source reference software development kits (SDKs) in C# and Java. The U-Prove technology is centered around a so-called U-Prove token. This token serves as a pseudonym for the prover. It contains a number of attributes which can be selectively disclosed to a verifier. Hence the prover decides which attributes to show and which to withhold. Finally there is the token's public-key, which aggregates all information in the token, and a signature from the issuer over this public-key to ensure the authenticity [20].

Jan Camenisch, Markus Stadler and Anna Lysyanskaya studied in [7], [5] and [6] the cryptographic bases for signature schemes and anonymous credentials, that later became IBM's Identity Mixer protocol specification [21].

Luuk Danes in 2007 studied theoretically how Idemix's User role could be implemented using smart cards [9], identifying what data and operations should be kept inside the device to perform different levels of security. The User role was divided between the smart card, holding secret keys, and the Idemix terminal, that commanded operations inside the smart card, or read the keys in it to perform the instructions itself. The studied sets were:

- The smart card gives all information to the terminal.
- The smart card only keeps the master key secret.

- The smart card only gives the pseudonym with the verifier to the terminal.
- The smart card keeps everything secret.

Later, in 2008 Víctor Sucasas also studied an anonymous credential system with smart card support [13], equivalent to a basic version of Idemix, using a simulator to test the PoC and pointing out some crucial implementation details for performance. The researching tendency starts to show that smart cards are the best solution to hold safely the User's credentials.

In 2009, some Java smart card PoC for Idemix were developed in [2] and [23], but they weren't optimal and didn't include some Idemix's functionalities, like selective disclosure.

Later, in 2013, Vullers and Alpar, implemented an efficient smart card for Idemix [24], aiming to integrate it in the IRMA[1] project, and comparing the performance with U-Prove's smart cards. This new implementation was written in C, under the MULTOS platform for smart cards, and describes many decisions made during the development to improve the performance on such constrained devices. The terminal application was written in Java and used an extension of the Idemix cryptographic library to take care of the smart card specifics.

Extending the concept of smart cards, physical or logical, as holders of the credentials, the ABC4Trust's project, P2ABCE[2], was created as a unified ABC system for different cryptographic engines, currently supporting U-Prove and Idemix. The Idemix library was updated to support P2ABCE and the last version is interoperable with U-Prove. Therefore, the smart card specification from the P2ABCE project could be considered the official version to work with.

Related to the IoT, the P2ABCE project has been used to test in a VANET[3] scenario how an OBU (On Board Unit), with constrained hardware, could act as a User in an P2ABC system [10]. However, after the theoretical analysis, the paper only simulates a computer with similar performance as an OBU, without adapting the existing Java implementation of P2ABCE to a real VANET system. In our project, we can consider ourselves as part of their *future work*, because our PoC will run on hardware actually used in VANET systems, and has been implemented in C instead of Java, which is more realistic and efficient for its deployment in an OBU.

1 The IRMA project has been recently included in the Privacy by Design Foundation: https://privacybydesign.foundation/

2 Privacy-Preserving Attribute-Based Credentials Engine - https://github.com/p2abcengine/p2abcengine

3 Vehicular Ad-Hoc Network

3 OBJECTIVES AND ANALYSIS

In this chapter we describe the project objectives, and the methodology followed during its development. We continue then describing the privacy-preserving attribute based credentials and the ecosystem surrounding it, from the P2ABCE's perspective, as it is the starting point of our project. We finish with an introduction to all the technologies involved in our development, from P2ABCE's code, to how smart cards work.

3.1 PROJECT DESCRIPTION

The purpose of this project is the integration of IBM's privacy preserving solution, Idemix, in the environment of the Internet of Things. The objective is to design a general solution for the existing IoT devices and systems, without compromising any feature of Idemix, and provide a working PoC in a real IoT environment, even though it isn't the most constrained scenario. The project is aimed to be used in privacy-preserving environments, providing security for IoT, controlling what data is being disclosed and to whom.

In a smart city project, citizens' data can be privatized and at the same time continue to offer the benefits of the sensors around. Authorized personnel can disclose the information when required, like fire-fighters accessing a building's sensors to check how many people there are and what conditions they are in, in case of an emergency; but keep such invasive data private to other non-critical services, for example, only giving a proof that there are people in a floor of the building, to activate or deactivate the air-conditioning system.

We will divide the project in various objectives:

The main objective of this thesis is to integrate IBM's privacy-preserving Attribute Based System in the IoT environment. The idea is to enable IoT constrained devices to act as a autonomous entities that can authenticate and prove their attributes against a verifier entity using the Idemix protocol, thereby, allowing a privacy-preserving IdM solution for IoT.

This general objective, can be split in 6 main subobjectives:

1. Study of the Idemix ecosystem

   Study the state of the art of Idemix and the IoT, analysing the related projects and papers with similar approaches. Once we know as much as we can from the existing ecosystem, we will be able to deliver the most fitting solution to our project.

2. Design of the IoT+Idemix solution

   A formal description of the IoT architecture that will that will allow to integrate Idemix in IoT scenarios.

3. Implementation of a functional PoC

   Software objectives

   A maintainable, structured and extensible solution, from the IoT and original project perspectives, that is, the IoT solution must be interoperable with other non-IoT solutions, now and in the future, given new IoT systems or changes in the cryptographic protocols.

   Hardware objectives

   The implemented software needs to be optimized to be deployable in IoT devices as constrained as possible, and consider the multiple IoT platforms in existence.

4. Validation of the PoC

   Deploy a Proof of Concept (PoC) in a real scenario, without the need of simulators, checking that it works as expected.

5. Evaluation of performance

   This evaluation will allow to validate the feasibility of our solution and its implementation.

6. Integrate the IoT+Idemix as a security solution for other projects

   The IoT+Idemix project is a solution for the privacy threads in the IoT environment, and does not work on its own, but as part of other projects.

3.2 METHODOLOGY

Giving the vast range of IoT devices, a one-for-all solution must take in consideration multiple requirements and limitations. We will break down the original system, Idemix and P2ABCE, analyse every part of it and categorize them. We will consider what components are mandatory to be executed in the target device, and which components the devices would actually be able to execute. Given the mathematical base of the Idemix protocol and the complexity of the P2ABCE architecture and code, this may be the most challenging step, because it determines how the rest of the project will be developed.

Devices with equivalent processing power to smart phones are currently capable of running the Java implementations of P2ABCE with Idemix, but the most constrained IoT devices can not handle the entire system, only the mandatory cryptographic calculations, that is, the operations required to keep secured the privacy information in the device, their keys and certificates.

Using the technique known as *Computation Offloading* we can design an IoT architecture where the most constrained targets can keep their private keys and certificates secure within the device, and act as any other actor in the P2ABCE system. Studying the original P2ABCE architecture and implementations we will identify the key operations to be executed in the constrained device, and how to communicate efficiently during the delegation.

After the technical design, we will implement the PoC, using known software designs patterns, that will improve the maintainability of the project. Taking advantage of this practices, we can document the immediate steps for future developers, how to reimplement certain interfaces when porting the application to a new system, or where the core logic lies, to implement future protocol changes.

Finally, the PoC will be evaluated to assert that we achieved our goals, and measure its performance, judging if it can indeed be suitable for a IoT deployment.

3.3 ANALYSIS OF P2ABCE

There are several cryptographic systems for dealing with attribute-based identities. Typically these systems distinguish credentials and attributes. Informally, a credential is a cryptographic container of attributes, where an attribute has a type and a value [24].

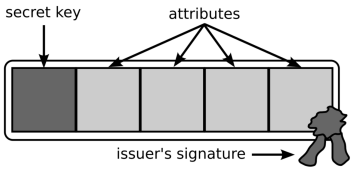


Figure 1: A first look at an attribute-based credential with four attributes.

The Privacy Preserving Attribute Based Credentials system is composed by several actors, each one of them with different roles. One entity could act with more than one role, e.g., in a M2M (Machine To Machine) scenario, a device could act as both User and Verifier to other peers; but one can assume each actor acts with one role at a time.

The roles are:

- **Issuer**
  In the ABC infrastructure, the Issuer is a trusted entity responsible for issuing credentials, vouching for the correctness of the information contained. Each Issuer generates a secret key and publishes the Issuer parameters that include the corresponding public verification key.

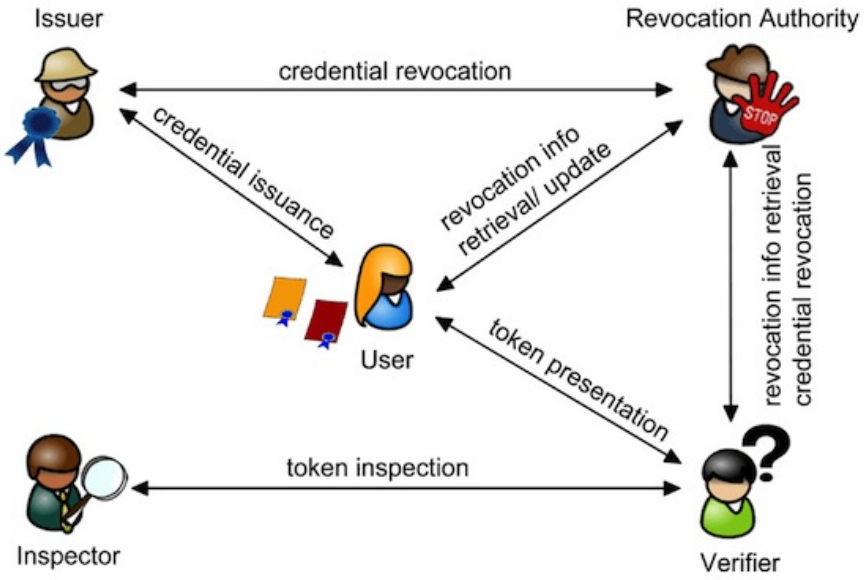


Figure 2: Entities in a P2ABC System

- **User**
  Entities that collect credentials from various Issuers. They can decide between all their credentials which attributes and values to present when making assertions about their identity to service providers.

- **Verifier**
  A service provider that protects access to a resource by imposing restrictions on the credentials that users must own and the information from these credentials that users must present in order to access the service.

- **Revocation Authority** (optional)
  This entity is responsible for revoking issued credential, preventing their further usage to generate a presentation token. Each revocation authority generates and publishes its revocation parameters.

- **Inspector** (optional)
  The Inspector's duty is to de-anonymize the user's presentation token under specific circumstances (e.g. misuse or liability). At setup, each inspector generates a private decryption key and a corresponding public encryption key. Usually, the capability of inspection should be bonded to privacy protection laws.

In our work, we aim at making IoT constrained devices capable to act as a User or Verifier in the P2ABCE architecture, given, for example, a M2M (Machine to Machine) setup, a device could identify itself

to others (User), and ask for the respective cryptographic evidence to its peer (Verifier).

The key operations of Idemix are the Issuance and Proving, but there are also others like the revocation of credentials, update of attribute values or inspection of a prove. We consider that the cryptographic details of Idemix are out of the scope of this thesis, and can be consulted in [21], but we will give a brief introduction of the Issuance and Proving interactions. Also, for those who want a little more information about the mathematical operations performed, without the deep explanations from the Idemix specification, we recommend to read the section 3.2 from [24].

The **Issuance** protocol consists on an interactive Camenisch - Lysyanskaya signature, that is, unlike usual signatures, where the Issuer alone hashes the credential (implying she knows all the values in it) and ciphers the digest, in this signature, the signing is performed between both actors.

This is possible given the nature of the CL signature, where the values of the parameters act as exponents of some public bases, and the use of ZKPs assure each party performs the correct calculations, avoiding malicious users. The User will perform the exponentiation with her secret key and some hidden attributes (the Issuer won't know their true value, only that they have some property, through a ZKP). The User then sends this new value and a ZKP that she indeed knows the exponents (the hidden values in the credential) for the public bases, without revealing said exponents. The Verifier performs the same exponentiation with the public attributes (not hidden by the User), and using her secret key, finishes the signature. The User then receives a valid CL signature for the credential, and no one except her knows the secret key and hidden values, thanks to the ZKPs.

The **Proving** protocol is simpler than the Issuance. It is usually performed in a three step protocol. First, the Verifier generates a nonce for the current interaction. Then, the User performs a non-interactive Zero-Knowledge Proof of any predicate she wants to proof. This is performed doing various ZKPs in parallel, using the same nonce from the Verifier. After the non-interactive ZKP is performed, the User sends it to the Verifier, which needs less calculations to verify the predicate, and learns nothing more than that the predicate itself is true, that is the nature of Zero-Knowledge Proofs.

Although we see both protocols are interactive, with at least 3 messages each, we mention *non-interactive* ZKPs. If we recall the cave story, Victor needed to repeat the experiment many times before believing that Peggy knew the secret. Here, using the Fiat-Shamir heuristic and the Verifier nonce, the Verifier challenges (in the story, the path to return from) are substituted by a Hash function, e.g.

SHA256, performed over some values. The User can't anticipate the digest result, so the Verifier trusts the Zero-Knowledge Proof, even when she doesn't choose the challenge. The Fiat-Shamir heuristic significantly increases the performance of these protocols, reducing the number of messages exchanged.

As we have seen, Zero-Knowledge Proofs used in Idemix allow a user to prove ownership of a credential without revealing the credential itself. Since the verifier does not see the credential, verification instances are unlinkable and they also cannot be related to the issuing procedure. The privacy of users is protected by these unlinkability properties, even if the credential issuer and all verifiers collude. We can identify the following key features of the Idemix system [10]:

- **Issuance unlinkability**: the issuer cannot link an issued credential to the presentation of such credential.
- **Multi-show unlinkability**: a credential can be used multiple times without the resulting evidence becoming linkable.
- **Selective disclosure of attributes**: allows users to prove only a subset of attributes to a verifier.
- **Predicate proof**: it consists of statements that allow to prove a property of an attribute without disclosing its actual value. Example of these statements are the logical operators > or <.
- **Proof of holdership**: a cryptographic evidence for proving ownership or possession of a credential without disclosing the attributes contained in that credential.
- **Non-transferability**: key binding can be used to bind one or more credentials of a user to the same secret.
- **Scope-exclusive pseudonyms**: a certified pseudonym unique for a specific scope and secret key, i.e. a single pseudonym can be created for each credential.
- **Carry-over attributes**: it relies on the assumption that the user already possesses a credential, from which a given attribute can be carried over into the new credential without disclosing the attribute value to the Issuer.
- **Cross-credential proofs**: it allows users to prove relations between attributes from two or more credentials without revealing them to the verifier. For instance, that the name contained on a credit card and on a passport match.
- **De-anonymization**: it is an optional feature that allows an Inspector, either alone or in cooperation with other entities, to

reveal the identity of users in cases of accountability and non-repudiation.

- **Revocation**: in case of misuse, it allows the revocation of issued credentials to (misbehaving) users. Thus, revoked credentials can no longer be used to generate Presentation Tokens.

To sum up, an attribute-based credential contains attribute-value pairs that are certified by a trusted Issuer. A credential can also specify one or more Revocation Authorities who are able to revoke the credential if necessary. Using her credentials, a User can form a presentation token that contains a subset of the certified attributes, provided that the corresponding credentials have not been revoked. Additionally, some of the attributes can be encoded in the presentation token so that they can only be retrieved by an Inspector. Receiving a presentation token from a User, a Verifier checks whether the presentation token is valid w.r.t. the relevant Issuers and Inspector's public keys and the latest revocation information. If the verification succeeds, the Verifier will be convinced that the attributes contained in the presentation token are vouched by the corresponding Issuers.

3.3.1 *P2ABCE Code Structure and REST Services*

In the P2ABCE repository[1] there is available the project's code, divided in two solutions: a complete P2ABCE implementation in Java and a MULTOS smart card implementation as PoC for the project.

The Java code is managed by a Maven project, structured using various known design patterns, but this code is not our main target, as it won't be executed in the device. The part we are actually interested in are the *REST Services* and their use of the *Components* classes, where the smart card's logic and use are defined.

P2ABCE project is based on the concept of smart cards, virtual or physical, to store the Idemix credentials and keys. An interface is defined to communicate with these smart cards, and then different implementations allow to use either *Software Smartcards* or *Hardware Smartcards*.

The *SoftwareSmartcard* class implements the interface in Java, suitable for applications using P2ABCE self-storing digital smart cards, like a virtual wallet. This implementation is useful, for instance, to be deployed in more capable IoT devices (that can run Java) such as smartphones or smartwatches.

The *HardwareSmartcard* class uses the standard APDU messages (3.4.1) to interact with smartcards. P2ABCE defines the necessary APDU instructions for the smart card needed to implement each method of the interface. It relies on `javax.smartcardio` abstract classes

1 https://github.com/p2abcengine/p2abcengine

(implemented by Oracle in their JRE) to communicate the smart card reader and the smart card. This way, it doesn't matter what manufacturer issues the smartcard, or if it's an Android device with NFC, if they support the APDU instructions, P2ABCE will work with them.

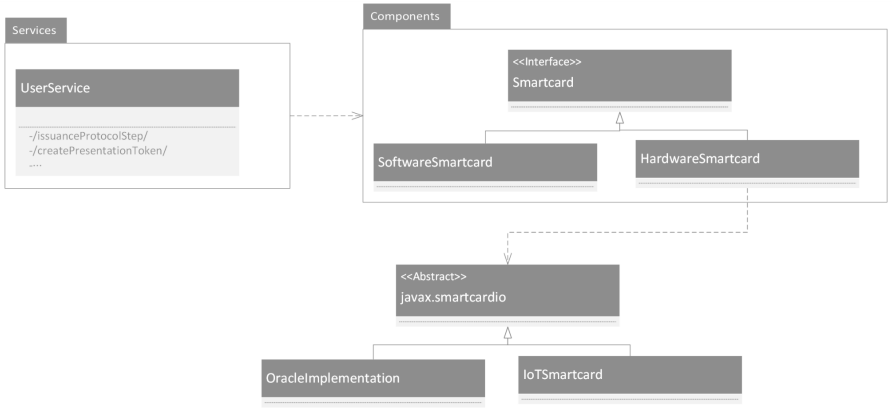


Figure 3: Basic P2ABCE structure

The project also provides a set of REST services to control each role of the P2ABCE project (User, Issuer, Verifier, Inspector, etc.). The methods implemented include the creation of *Software* smart cards within the User Service, and store them in a data base for future REST calls that may need them.

The services receive parameters like the length of the cryptographic keys, IDs, or XML files to parse. The REST API is not meant for a User Service to communicate with an Issuer Service or Verifier, but for the User actor to call the P2ABC Engine through the User Service, to perform specific actions (execute the User side of the Idemix issuance and proving protocols), and then this Service returns the Response XML. Those XML files are the way to communicate two different actors (e.g. User and Verifier). The transmission method to exchange the XML depends on the specific scenario.

This Services as *tools* for the actors are only present in the P2ABCE project, not part of the Idemix or U-Prove engines. They are useful in a desktop environment, where we have a User Service in the background, and many programs, browser plugins, etc., make use of the Engine through the REST interface, reducing the number of P2ABCE entities in execution.

3.3.2 *ABC4Trust Card Lite*

As a PoC the P2ABCE project includes a smart card reference implementation, the ABC4Trust Card Lite [8]. It supports device-bound U-Prove and Idemix, and virtually any discrete logarithm based pABC system.

Version 1.2 is written for `ML3-36K-R1` MULTOS smart cards, with approximately 64KB of EEPROM (non-volatile memory), 1KB of RAM and an Infineon SLE 78 microcontroller, a 16-bit based CPU aimed for chip cards.

The card stores the user's private key x and any Binary Large OBject (BLOB) that the P2ABCE may need (like user's credentials). Then P2ABCE delegates the cryptographic operations on the smart card, that operates with x.

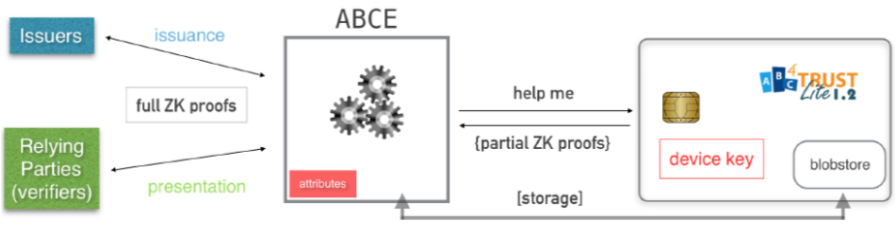


Figure 4: ABC4Trust Card Lite. *Image from ABC4Trust EU Project.*

The cryptographic operations performed by the smartcard are the modular exponentiation and addition used by ZKPs based on the discrete logarithm problem.

The code is available from the P2ABCE's repository and has some good and bad points to take into account:

The best asset of this code is that it is written in C aiming to a very constrained device, with very limited memory and similar computational power to many other IoT devices.

The code uses some good practices when programming for constrained devices, also explained in the first implementation of Idemix in smart cards [24]. Aside the implementation in assembly code of some operations to accelerate the execution, most techniques aim to reduce the memory usage. They define `union` data types to join variables never used at the same time, so they can be stored on the same data location. Instead of function parameters, most variables are made global, reducing the space on the stack used when calling other functions. If parameters are needed, there are pointers to shared buffers with fixed sizes, avoiding the use of any dynamic memory allocation.

On the other side, among the many **drawbacks**, we could highlight the *awful* coding style, the strong dependency on MULTOS framework and the fact that many bugs were found during our work.

The MULTOS C subroutines are an interface to hardware implementations for the smart cards, and it implies the ported code must be rewritten to an interoperable C code, not dependant on MULTOS specific functions.

The code is also affected by the MULTOS architecture, breaking many execution sequences, that is, the running application can stop at

any point, not returning from a function. The MULTOS OS will read the public memory to generate a Response APDU and then clean the stack memory, but our standard C application must return from every function, freeing the used memory, minimizing possible errors from bad memory management. As we will see in the next chapter, the sequence diagram shows how any execution should perform, returning to the caller, not finishing the process when an error happens.

Although we mentioned the use of pointers as a way to reduce memory usage, we all agree that pointers are *double–edged swords*, and a code with such an intense use of them is prone to undetectable errors before runtime.

The code is structured in two files, *main.h* and *main.c*, with around 550 and 5200 lines of code, respectively. The file *main.h* is mostly a reimplementation in assembly MEL code of some MULTOS functionality already offered by their API.

The *main.c* consists on nearly 600 lines of variables and data structures declarations; followed by the *main()* function, a 2600 lines long *switch-case* expression, that handles every APDU instruction, with practically no comments; and to conclude, the implementation of thirty functions called *Subroutines* at the end of the file, with around other 2000 lines of code.

At this stage, we have two options to implement our IoT device compatible with P2ABCE:

- Implement in C the `Smartcard` interface used by P2ABCE architecture, like the `SoftwareSmartcard` class, and use some communication protocol to remotely call the methods from the machine running the P2ABC Engine.
- Present the IoT device as a hardware smart card, using the APDU protocol (already defined, standard and with minimal overload). Providing a *javax.smartcardio* "IoT implementation" to communicate with the IoT device through a transmission protocol, making the already existing *HardwareSmartcard* class compatible with the new *IoTSmartcard* running in the IoT device.

Even with the problematic to maintain or even understand the code of ABC4Trust Card Lite, once one studies MULTOS framework in deep, applies many refactoring techniques to the code, consider the applied optimizations for constrained devices, and the APDU protocol compatibility to work with `HardwareSmartcard` as any other smart card, it becomes the best starting point for the IoT version, making us opt for the second option.

3.4 PRELIMINARIES ABOUT SMART CARDS

We have decided to extend this chapter in order to explain the basis of smart card technologies, instead of only referencing the corresponding documentation, involving the reader to make the effort of understanding a system that is more complex than what we actually need to know at this point.

This section is a summary of the smart card's communication protocol, the MULTOS ecosystem, and, therefore, the inherited ABC4Trust Card Lite code, and how our IoT solution will work in the inside, acting as a smart card, a port from a (poorly coded) MULTOS application.

3.4.1 *Smart Card Communication Protocol*

To communicate the smart cards and the reader the `ISO/IEC 7816-4` [12] specifies a standardized protocol .

The messages, also kown as Application Protocol Data Units (APDUs), are divided in APDU Commands and APDU Responses.

APDU Commands consist in 4 mandatory bytes (CLA, INS, P1, P2), and an optional payload.

- CLA byte: Instruction class. Denotes if the command is interindustry standard or proprietary.
- INS byte: Instruction code. Indicates the specific command.
- P1, P2 bytes: Instruction parameters.
- Lc, 0-3 bytes: Command data length.
- Command data: Lc bytes of data.
- Le, 0-3 bytes: Expected response data length.

This way, minimal number of bytes are needed to transmit commands to the smart card, allowing manufacturer's personalization of the smart card behavior and capabilities along with standard operations.

Figure 5: APDU Command

APDU Responses are generated inside the smart card, always as an answer to an APDU Command. They consist on an optional payload and two mandatory status bytes.

- Response data: At most Le bytes of data.
- SW1-SW2 bytes: Status bytes. Encode the exit status of the instruction.

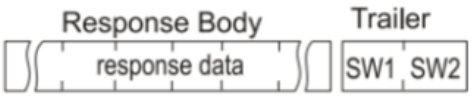


Figure 6: APDU Response

The transmission protocol varies between different types of readers and smart cards (e.g. chip, contact-less), but what is common between every smart card interaction, is the *APDU Command-Response Dialogue*. As long as the smart card has a power supply, it can maintain the memory in RAM between APDU Commands, what allows to do in two or more steps complex operations, transmit more bytes than a single APDU can admit, etc.

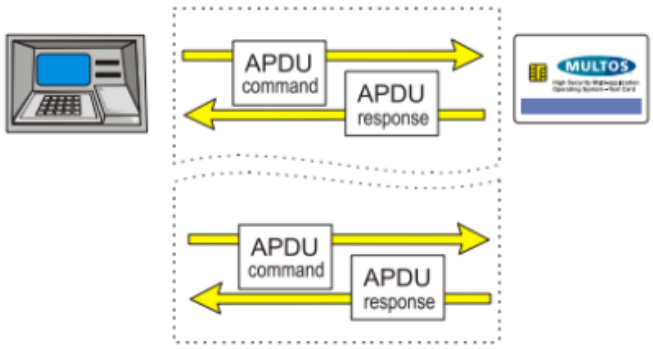


Figure 7: APDU Command-Response Dialogue

Originally, the Lc and Le bytes had only 1 byte, if present, restricting the payload data to be at most 256 bytes long. An extension to the protocol changed the meaning of a Lc or Le 0x00 byte (256 bytes long payload), so when the byte corresponding to Lc or Le started with 0x00, the next two bytes where the real length. With this, an Extended APDU lets up to 65536 bytes of data.

The problem here, is that not all readers or smart cards support extended APDUs. Originally, to send more than 256 bytes of data in an APDU Command, a *Put Data* instruction is defined, so the smart card stores the payload in a buffer, until other APDU Command indicates how to use it.

To send more data in an APDU Response, the status bytes are set to: SW1=0x61 and SW2 to the remaining bytes to send. Because a smart card can't send APDU Commands, the card terminal must send a `GET RESPONSE`, a special APDU Command, with Le set to the number of bytes specified in SW2. Iterating this process, the smart card can send as many bytes as it wants as Response.

With the introduction of Extended APDUs, this technique is no longer needed.

3.4.2 *MULTOS*

MULTOS is a multi-application smart card operative system, which provides a custom developing environment, with rich documentation[2]. MULTOS smart cards communicate like any other smart card following the standard, but internally offers a very specific architecture, affecting the way one must code applications for it.

In this section we will present the main characteristics of a MULTOS smart card that shaped the ABC4Trust Card Lite code and that we had to be aware of when adapting it to IoT devices.

MULTOS PROGRAMMING LANGUAGES A native assembly language called MEL, C and, to a lesser extent, Java, are the available languages to code for MULTOS. In our case, ABC4T Card Lite uses MEL and C.

EXECUTION MODEL Applications on a MULTOS card are executed in a virtual machine, called the Application Abstract Machine (AAM). The AAM is a stack machine that interprets instructions from the MULTOS Executable Language (MEL).

The transmission and communication process is done by MULTOS core, and it then selects, based on the CLA byte of the APDU, the application to load. This application is what most developers will only worry about, and is where their compiled `main()` function will start.

Now the developer is in charge of checking what instruction was sent, handle it with regard to his domain logic, write in the specific data space the APDU Response bytes, and call `multosExit()`, a MULTOS API function that will be in charge to send the APDU Response. In summary, our application starts with all data loaded in memory and exits without worrying how the answer is sent back.

As it can be sees, MULTOS is a comfortable framework to develop smart card applications, and now we must adapt and implement it for our IoT devices, if we want to port ABC4Trust Card Lite's code.

2 MULTOS technical library - http://www.multos.com/developer_centre/technical_library/

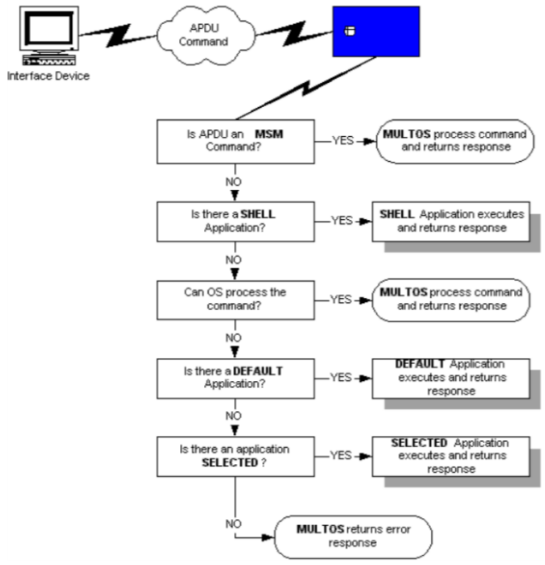


Figure 8: MULTOS Workflow

MULTOS MEMORY LAYOUT Each application in MULTOS has access to a specific memory layout, divided in different categories:

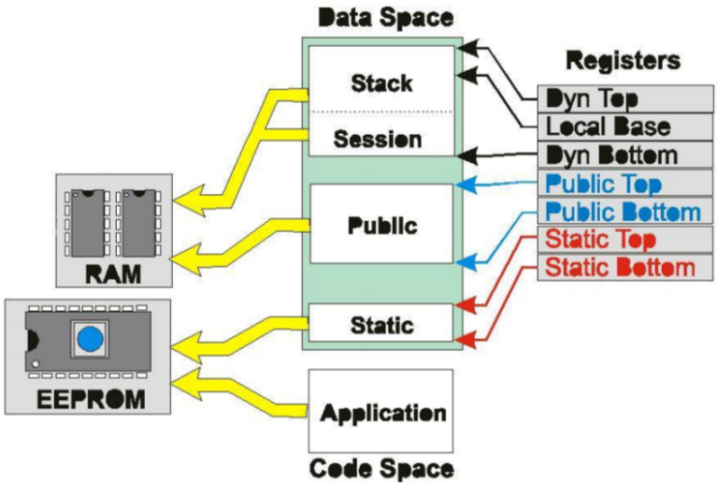


Figure 9: MULTOS Memory Layout

The Code Space is where the application code is stored. The Data Space is divided in Static memory, Public memory and Dynamic memory.

Static memory are the application variables declared after the specific `#pragma melstatic` compiler directive. These variables are stored in the non-volatile secure EEPROM, and any write is assured to be saved because they are not loaded into RAM.

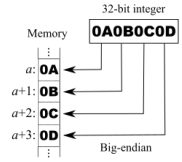
Public memory can be seen as the input/output buffer for applications and MULTOS system. The APDU header appears at the top

of Public, and command data at the bottom. The application writes then the APDU Response bytes in Public, at specific position (see Figure 10). To declare variables in this data space, the `#pragma melpublic` directive is available.

Dynamic memory works like usual program memory, with Session Data storing global variables and the Stack. The limited size of RAM in IoT devices and smart cards makes the use of dynamic memory not advisable. The compiler directive to use Session Data is `#pragma melsession`.

Address	Name	Description
PT[-1]	SW2	Byte 2 of the Status Word
PT[-2]	SW1	Byte 1 of the Status Word
PT[-4]	La	Actual length of response data
PT[-6]	Le	APDU expected length of response data
PT[-8]	Lc	APDU length of command data sent
PT[-9]	P3	If required, temporary buffer for 5th byte, if any, of APDU header
PT[-10]	P2	APDU Parameter byte 2
PT[-11]	P1	APDU Parameter byte 1
PT[-12]	INS	APDU Instruction byte
PT[-13]	CLA	APDU Class byte
PT[-14]	GetResponseSW1	Byte 1 of Status Word to be used in Get Response command
PT[-15]	GetResponseCLA	CLA to be used by Get Response command
PT[-16]	Protocol Type	Transport protocol type
PT[-17]	Protocol Flags	Bit flags indicating status of protocol values
PB[0]	Start of Data Area	Command data and response data start

Figure 10: MULTOS Public Memory Data Map



Big-Endian, Wikipedia

With regards to primitive types, to avoid confusion with their sizes, MULTOS defines and uses the following data types specified in Figure 11. It's important to notice that MULTOS is Big Endian and when storing structures there is no padding between defined variables, unlike modern compilers that perform data structure alignment [3] for performance.

Data Type	Definition
BOOL	boolean (byte)
BYTE	unsigned byte (byte)
SBYTE	signed byte (byte)
WORD	unsigned word (2 bytes)
SWORD	signed word (2 bytes)
DWORD	unsigned double word (4 bytes)
SDWORD	signed double word (4 bytes)

Figure 11: MULTOS Data Types

MULTOS STANDARD C-API A collection of more than a hundred functions are provided for arithmetic, cryptography, memory and smart card operations. The *multos.h* interface provides access to these

3 https://en.wikipedia.org/wiki/Data_structure_alignment

functions, that ultimately call their respective primitive instructions in assembly code. The primitive instructions are but a system call with an operation code, loading data in the needed registers, and hardware implemented functionality. Therefore, no implementation for these tools is available, nor in C, nor in assembly code.

Nevertheless, the C-API documentation[4] provides rich description for each function.

Following this documentation, we will have to reimplement in C the same or extended functionality, trying to be as efficient as the hardware implementations in MULTOS smart cards. For example, the cryptographic operations are common in smart cards and therefore hardware supported, but a constrained IoT device does not usually include a cryptographic chip or specialized processor instructions.

3.5 DEVELOPMENT ENVIRONMENT

For almost every IoT device in the market, there exists a C compiler and many frameworks available to build firmware binaries, like Arduino Core, Contiki, Mongoose OS, ThreadX OS, OpenWrt, LEDE, proprietary SDKs, etc. Each firmware targets a specific range of devices, depending on processing power and memory limitations. For example, Arduino and Contiki aim for very constrained microcontrollers, like Atmel's ATmega or TI's MSP430, but can also be used in ESP8266, a more powerful device, with WiFi capabilities.

ATmega328P

Starting a big project development for IoT, aiming the most constrained devices may not be a good idea. The lack of usual OS tools, like POSIX, threads, or minimum I/O, like a terminal, can make debugging a tedious task. With good programming practices, one can start from the top and slowly end with more constrained devices and reliable code.

For this reason, the current PoC is developed on LEDE[5] using the Onion Omega2 development board. Although the Omega2 uses LEDE, its microcontroller is also listed as compatible with ThreadX[6], therefore, the performance measured in our PoC can be relevant to real scenarios with similar hardware.

The PoC will take advantage of the Linux system using mainly files and sockets like in any other Linux desktop distribution, so we can focus on the project itself rather than the specific platform APIs for storage and connectivity.

4 http://www.multos.com/developer_centre/technical_library/
5 Linux Embedded Development Environment - https://lede-project.org/
6 THREADX Real-Time Operative System for embedded devices - http://rtos.com/products/threadx/

The project development is divided between the IoT device code and the P2ABCE services. To ease the setup of new developing machines, we will use Docker[7] to deploy two containers, one ready to cross-compile our IoT project's C code, and the other container will compile the P2ABCE's Java code.

P2ABCE is already written in Java and uses Maven to manage dependencies. The project needs some minor changes to work with our IoT architecture. Any text editor or Java IDE is suitable for the development, because the compilation is done through the terminal, with Maven commands.

We compile the project inside a Docker container, with OpenJDK 7, Maven 3 and Idemix 3.0.36, following the project instructions to use Idemix as the Engine for P2ABCE.

We can assume all IoT devices have a C cross-compiler, some even a C++ cross-compiler. The worst case scenario is that one must write assembly code, and that code will be specific of that target, so we won't consider them. If now we focus on the most constrained devices, we could find out that for some we can't use C++, some may not have many usual libraries, moreover, the memory limitations they face make practically impossible to use dynamic memory, if we want to avoid many execution malfunctions.

For that reason, the developed code for IoT devices should be written with standard C, without using dynamic memory or third party libraries.

To manage the PoC code we chose CMake, providing many advantages over Makefiles:

- Cross-platform. It works in many systems, and more specifically, in Linux it generates Makefiles.
- Simpler syntax. Adding a library, files to compile, set definitions, etc. can be done with one CMake command, with rich documentation on the project's website.
- Cross-compilation. With only a CMAKE TOOLCHAIN file, CMake sets up automatically the cross-compilation with Makefiles and the C/C++ cross-compiler provided.

Although the ideal final code is written in pure C, without external libraries or dynamic memory, the PoC uses three major libraries:

- OpenSSL: Provides reliable and tested AES128, SHA256 and random number generator implementations.

7 https://www.docker.com/what-docker

- LibGMP: Provides multiprecision integer modular arithmetic.
- cJSON: Provides a JSON parser to store and read the status of the device, in a human readable way.

These three libraries are used to implement different interfaces in the project, and C implementations of these interfaces should replace the external libraries in the future.

Finally, we use Docker to deploy the compilation environment. Our container includes CMake and the LEDE SDK [**ledeproject**], configured for the Omega2 target, the device chose for the PoC.

4

DESIGN AND IMPLEMENTATION

In this chapter we describe the design of our proposal to integrate IoT constrained devices as part of the privacy preserving system P2ABCE. The ultimate goal is to enable constrained IoT devices to play the Idemix User role, interacting autonomously in order to authenticate and demonstrate their credential attributes in a privacy-preserving fashion.

4.1 DESIGN

In this section we will define how an IoT device may be integrated in the P2ABCE architecture, being totally compatible with any other system using P2ABCE, addressing the power and memory constrains many IoT devices face.

We decided to use P2ABCE with the Idemix as its Engine, because it is officially supported by the Idemix Library, it has the most up-to-date implementation, as we saw in the state of the art, and adds capabilities to Idemix like the Presentation Policies, or interoperability with U-Prove, not available without the P2ABCE project.

Our main goal is to make an IoT device capable to act as a User or Verifier in the P2ABCE architecture. For this, the device should be able to **communicate** with the Verifier or Prover with which it is interacting, manage the P2ABCE complex **XML schemas** transmitted, and perform the **cryptographic operations** required.

The communication between actors depends on each IoT scenario, it can be achieved with many existing standard solutions, e.g. an IP network, a Bluetooth M2M connection, RF communication, etc.

Our real concerns are, on one side, parsing the XML data, based on the P2ABCE's XML schema, that specifies the data artifacts created and exchanged during the issuance, presentation, revocation and inspection of pABCs; and on the other side, the cryptographic operations, that involve the use of secret keys, stored privately in the IoT device.

After the analysis done in the previous sections to the P2ABCE architecture, emphasizing that the logic of smart cards gathers the cryptographic operations independently from how the data is exchanged between P2ABCE actors.

Using the *computation offloading* technique to our scenario, our design consists on implementing the smart card logic inside the IoT device, keeping secure our master key and credentials, and for the rest of the P2ABCE system, if the device can not run the complete

Engine, it may delegate to a server running it, indicating how to send APDU Commands to the *IoT smart card*.

Even in the case we were to implement all P2ABCE inside an IoT device, we would have to implement the support for software smart cards, to keep the secret inside the IoT device. Therefore, we can begin implementing the smart card logic inside the IoT device, and later, if the device resources admit it, other components of the P2ABCE project.

Computation offloading is not new to IoT deployments. For example, IPv6 involves managing 128 bits per address and other headers, and many IoT scenes only need to communicate inside a private network, making only the last 64 bits in an address relevant. To reduce that overhead, instead of IPv6 they use 6LoWPAN to compress packets and use smaller address sizes. To communicate a 6LoWPAN with the Internet or other networks, the IoT devices delegate the networking workload on a proxy that can manage the 6LoWPAN and IPv6 stacks. In the scope of consumer devices, smart bands or watches can install applications, bit many of them delegate on the user's phone to accomplish their task.

Therefore, the IoT device now has a **duality** in its functions, because it is the User that starts any interaction with other actors, and it's also the smart card that a P2ABCE server must ask for cryptographic operations. It can also be seen as a **double delegation**. The IoT device delegates on the external P2ABCE server to manage the protocol, and the P2ABCE server delegates on the IoT, acting now as a smart card, for the cryptography.

4.1.1 *System architecture*

The system will be compounded by the IoT device, the P2ABCE delegation server and the third party P2ABCE actors.

- **IoT device**

  In Figure 12 shows our proposed architecture, in which the IoT device is represented with two interfaces, physical or virtual. One allows external communications to other machines, including other P2ABCE actors, that could be on the Internet, a corporate LAN, a M2M overlay network, etc. Through this interface, the P2ABCE XML messages are exchanged as in any other deployment. This allows an IoT device to interact with other actors without special adaptations to the protocol. The other interface allows a secure communication with the delegation server. Both the delegation messages and the APDU Dialogue are transmit-

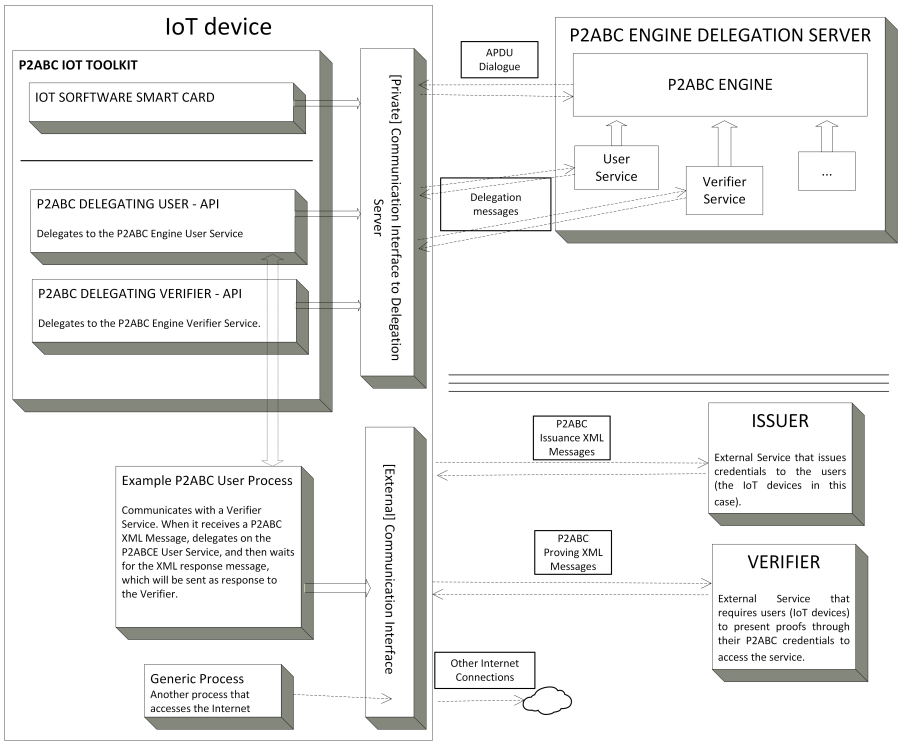


Figure 12: Proposed high level Architecture for integrating IoT devices in P2ABCE.

ted over this interface, making it a point of attack to the system, and we will talk about its security in the delegation process.

The scheme also shows the *P2ABCE IoT Toolkit*. This piece of software includes the IoT Smart Card, and the API for other processes that want to use the P2ABCE system.

The IoT Smart Card is the implementation of a software smart card, listens for APDU Commands from the secure interface and stores securely the credentials and private keys within the device's memory.

The P2ABCE API is an interface for other processes that wish to use the private-preserving environment of P2ABCE. It provides access to every operation available, hiding the delegation process. In the future, if for example the Verification Service is implemented for the IoT device, i.e., there's no need to delegate to other machine to act as a Verifier, then any program using the API won't need to change anything, the toolkit conceals the transition from delegating to native execution.

- **P2ABCE actors**

  If we recall from Section 3.3, the possible roles in the system were the Issuer, the User, the Verifier, the Revocation Authority and the Inspector. All of them use the P2ABCE XML schema in

the specification to communicate to each other. Any third party actor will be unaware of the fact that the device is a constrained IoT device that delegates on the P2ABCE server.

- **P2ABCE Delegation Server**

  The machine in charge of receiving authorized IoT devices' commands to parse the XML files exchanged and orchestrate the cryptographic operations the IoT smart card must perform.

Delegation process

Here we describe the computation offloading carried out by the IoT device. In Figure 13 we show an example of the IoT acting as a User, Proving a Presentation Policy to a third party Verifier.

1. Communication with P2ABCE actor.

   The IoT device, acting as a User, starts an interaction with another actor, e.g., against an Issuer to obtain a signed credential, or against a Verifier to demonstrate certain property of its attributes in a privacy-preserving way.

2. Delegation to the P2ABCE Server.

   Depending on what role the IoT device is acting as, it will delegate in the corresponding service, e.g. User Service. The delegation message must include the XML file, and any parameter required to accomplish the task, like the information on how to communicate with the IoT smart card (listening port, security challenge, etc.).

3. APDU Dialogue (if necessary).

   The server may need to send APDU Commands to the IoT smart card to read the credential information or perform cryptographic operations involving private keys, necessarily stored inside the IoT device.

   These APDU Commands include a list of 70 different instructions, identified by the APDU INS byte. Some of them manage the smart card, like changing the PIN code, and many are P2ABCE specific, like storing the system cryptographic parameters. Some examples:

INS	Description
0x52	SET_CREDENTIAL
	Sets a new credential inside the smart card.
0x5A	GET_CREDENTIAL_PUBLIC_KEY
	Returns a stored credential's public key.
0x60	GET_PRESENTATION_COMMITMENT
	First cryptographic operation during Proving.
0x62	GET_PRESENTATION_RESPONSE
	Second cryptographic operation during Proving.

4. Server response.

   The server may return a status code or a XML file if the first one required an answer from the IoT device, in which case, it will send as response to the third party actor, resuming the communication.

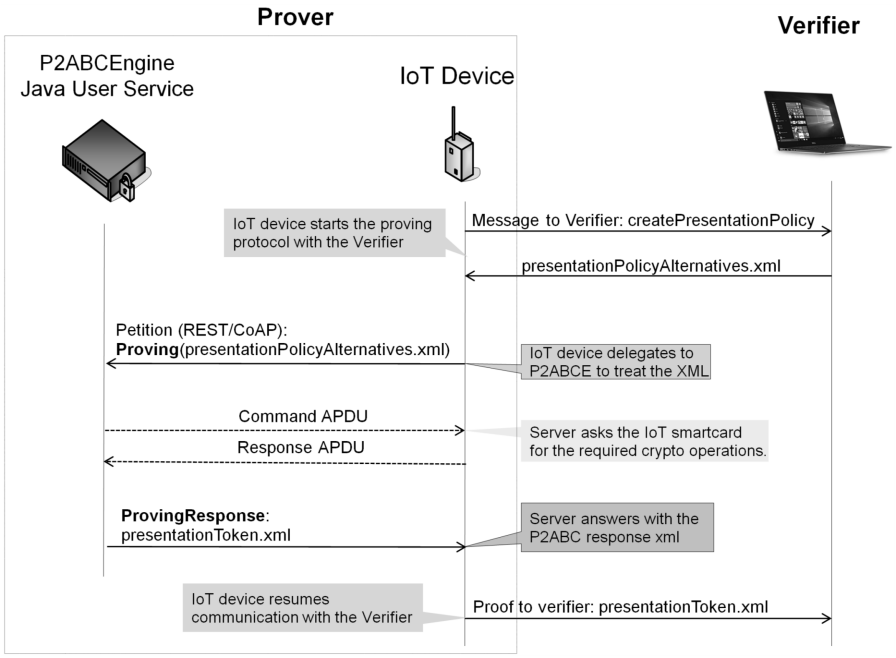


Figure 13: Designed interactions exchanged during the IoT delegation for P2ABCE Proving operation.

Transmissions over the *Server-IoT* channel must be secured in order to avoid attacks like: impersonate the P2ABCE delegation server, having access to the IoT smart card sending the APDU Commands the attacker wishes; delegate as a device on the server but giving the parameters of another device, making the delegation server send the APDU Commands to a victim IoT smart card.

We could use a corporative PKI to issue certificates to the server and devices and configure policies for access control; design a challenge-response system combined with the smart card PIN, like a password and TOTP[1] in a 2FA[2] login. We also could connect physically the delegation service through RS-232 serial to the IoT device, securing both physically as we would do with the IoT device on its own, isolating the delegation system from any network attack. This last idea is an approximation to the Arduino Yún[3], a development board that integrates two microcontrollers, one a typical Arduino with very low resources, and another one running a fork of OpenWrt. The Arduino microcontroller can control the terminal of the more powerful one, using the serial pins as commented before.

As we can see, there are many state of the art solutions for all this threads, therefore, we can assume a secure channel without mentioning a specific solution, providing freedom to choose the most fitting one in a real deployment.

NOTES FOR MORE CONSTRAINED DEVICES Our architecture is designed for devices that could in a future run a reimplemented version of P2ABCE, that means, the devices could perform more tasks than only running the smart card software and their main purpose process, e.g. recollecting sensor data. But if our target devices are so constrained that can barely run the smart card, they may not be able to handle the XML files because of memory restrictions, like a MSP430[4] running Contiki-OS, the microcontroller has between hundred of bytes to tens of kilobytes of memory, making impossible to store multiple XML files in the size range of tens of kilobytes.

In these cases, the delegation in the server goes a step forward, making the server a proxy to communicate with other P2ABCE actors, and the IoT device only acts as a smart card. The IoT device would still act as the User, or any other P2ABCE, role because it orchestrates when and how an interaction with other actor is executed, but the communications would be between the proxy and the third party actor.

4.2 PROOF OF CONCEPT IMPLEMENTATION

In this section we present the first PoC implementation, introducing the IoT system where we are going to work, then we will describe the delegation protocols, one for the computation offloading of the IoT device on the P2ABCE server, and another for the transmission

1 Time-based One-time Password
2 Two-factor authentication
3 https://www.arduino.cc/en/Main/ArduinoBoardYun
4 http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/overview.page

of APDU Commands, and finally, we will describe the IoT smart card implementation.

4.2.1 *IoT system*

We will develop our PoC in Linux based systems in order to avoid complications with firmware specific issues that we are not familiar with. Even so, the linux systems used are aimed for the IoT environment and serve as a starting point for future implementations in more constrained devices or different systems.

In our delegation server we will run Raspbian OS, a distribution based on Debian for the Raspberry Pi. We will talk more about the hardware specifications in the benchmark chapter. We chose this system because it has a great package repository support, including the Java Runtime Environment, needed to run the P2ABCE Services.

For our IoT device, we will use LEDE, Linux Embedded Development Environment, a distribution born from OpenWrt, aimed for routers and embedded chips with low resources requirements. It offers the `ash` command interpreter, or `shell`, and the `opkg` package manager, that allows an easy installation of some libraries and tools needed during the early development.

4.2.2 *PoC Delegation*

As we explained in the design section, the delegation has two steps, the IoT device calling the P2ABCE server to offload the parsing of the XML data, and the P2ABCE server sending APDU Commands to the IoT smart card in the device.

4.2.2.1 *PoC Delegation to the P2ABCE Server*

Currently P2ABCE offers multiple REST web services to run different roles in P2ABCE system: User Service, Issuer Service, Verification Service, etc. Any third party application that integrates P2ABCE with their system can make use of these services or implement the functionality using the library written in Java, the tool that the REST services actually use.

Our PoC machine, the Omega2, can make REST calls easily with the `curl` command, but other devices may use Constrained Application Protocol (CoAP), but in that case, the P2ABCE REST services should be adapted to offer CoAP support. The commands needed to delegate to the P2ABCE server would be the same that those defined now to operate with the REST services, but with the modifications needed to pass the parameters on how to communicate with the IoT smart card.

In this PoC, only the P2ABCE's User Service needed to be modified, because this is the role the IoT device plays. We added the following REST call:

/initIoTsmartcard/issuerParametersUid?host=&port=

where we communicate the P2ABCE server that an IoT Smart Card is accessible via the IP address *host* and TCP port *port*. Then a new *HardwareSmartcard* object is stored in the P2ABC Engine, but instead of the *javax.smartcardio* Oracle's *CardTerminal* implementation, we use our own *IoTsmartcardio* implementation for the *HardwareSmartcard* constructor, that we will discuss in the following section.

In a real deployment, we would offer a full library with an API for other processes that want to use the P2ABC Engine, which automatizes the boot of the IoT smart card and handles the security policies, like the one we presented in the deployment diagram 12 of the *P2ABCE IoT Toolkit*.

Now we show some examples of the PoC `curl` commands, run in the device's terminal, or from a shell script for automation:

```
$ curl -X POST --header 'Content-Type: text/xml' "http://
    $DelegationServerIP:9200/user/initIoTsmartcard/http%3A%2F%2
    Fticketcompany%2FMyFavoriteSoccerTeam%2Fissuance%3Aidemix?
    host=192.168.3.1&port=8888"

$ curl -X POST --header 'Content-Type: text/xml' -d
    @firstIssuanceMessage.xml "http://$DelegationServerIP:9200/
    user/issuanceProtocolStep/" > issuanceReturn.xml
```

The IoT device will only manage XML as data files to exchange between the third party actor and the P2ABCE delegation service, attaching them in the body of the REST call. In the parameters of the first call we can see the IP and port where the IoT smart card will be listening.

4.2.2.2 *APDU Dialogue Transmission*

To transmit the APDU messages in our PoC we use a simple protocol, that we will refer as BIOSC (Basic Input Output Smart Card), consisting in one first byte for the instruction:

`0x01`	APDU Command or Response: Read 2 bytes for the length and then as many bytes as said length.
`0xff`	Finish connection: close the current TCP socket and end the IoT smart card execution.

In the first instruction, we read two header bytes with the length of the APDU Command or Response to receive, followed by said APDU

bytes. The message is sent over TCP for a reliable transmission (concept of session, packet retransmission, reordering, etc.). The second instruction serves for closing the TCP socket in the IoT device when the smart card is no longer needed.

We lack any security (authentication or authorization) that a real system should implement. It is vital to authenticate the delegation service, to authorize it to make APDU Commands, and the same with the IoT device, to prevent attacks. But using this method for the transmission of the APDU messages, with only 3 bytes of overhead, helps us in the benchmarks to measure the real performance of the system.

The implementation of this simple protocol is done in Java for the P2abce delegation server and in C for the IoT smart card device.

As we mentioned in the analysis of P2ABCE's code, the `HardwareSmartcard` class implements the `Smartcard` interface using the package of abstract classes `javax.smartcardio` to communicate with the physical smart cards. The Oracle JRE implements these package for the majority of smart card manufacturers. For our PoC, we implement the `javax.smartcardio` package so it transmits the APDUs with BIOSC, making the use of a physical or IoT smart card totally transparent to the `HardwareSmartcard` class, enhancing maintainability, and following the *expert pattern* from the known GRASP guidelines.

In the PoC IoT device, we implement in `BIOS.c` the `main()` function, that reads from a Json file the previous status of the smart card, and then opens the TCP socket in the 8888 port, and loops listening for BIOSC transmissions until an `0xff` is received. When an APDU Command is sent, it calls the APDU Handler, that we will talk about in the next section.

4.2.3 *IoT Smart Card Implementation*

After many design decisions in the process to adapt the original ABC4Trust Card Lite code to pure C, working on a MIPS architecture, in this section, we present the current PoC code, most important decisions taken and the execution workflow in the form of sequence diagrams.

4.2.3.1 *Code structure*

We divide the project in three different sections with the objective of enhancing maintainability, improving future changes, ports, fixes, etc.

In Figure 14 we show the project's structure. The first section is what could be called as the core of the smart card, the second one

the interface for those tools the core needs and that may depend on the target's platform, and finally, the third party libraries, that may be empty if the interfaces implementation doesn't require any.

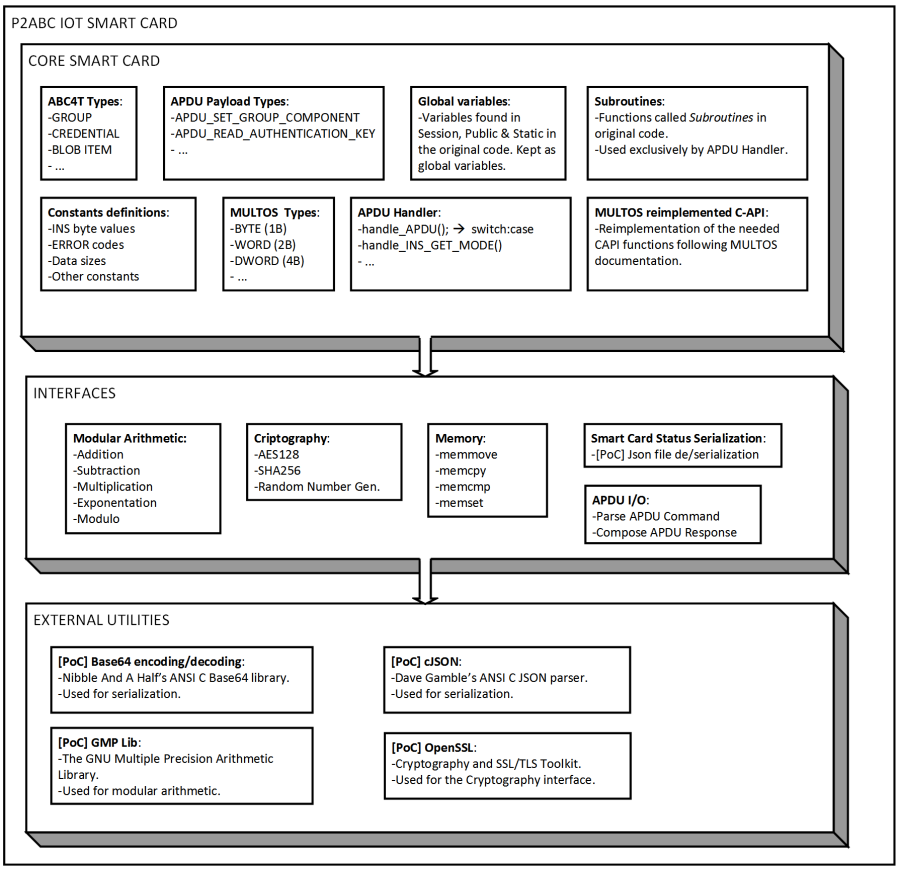


Figure 14: IoT Smart Card Code Structure.

CORE SMART CARD

The smart card logic lies in this section, the concepts of APDU Commands, what instructions are defined for P2ABCE smart cards, and how to process them and generate proper APDU Responses.

If the APDU protocol defined for P2ABCE smart cards changed, with the consequence of having to update all smart cards in use, the changes to adapt the IoT smart card should be applied in this part of the code. This also implies that if the P2ABCE Java code changes their `Smartcard` interface, something more probable, the IoT smart card will still work as intended.

Most of the original ABC4Trust Card's code has been reused in this section, e.g., all the global variables and custom data type definitions, the APDU Command handling and auxiliary subroutines.

However, the ABC4Trust's code heavily depended on the MULTOS platform for the input and output of data, modular arithmetic, mem-

ory management, and AES128 and SHA256 cryptographic operations. Therefore, we reimplement the used MULTOS functions following the documentation, adapting it in some cases.

A characteristic of MULTOS C-API is that every function name starts with "`multos`", so we changing their names from `multosFoo()` to `mFoo()` for readability and to emphasize that they were no longer MULTOS functions.

The `main.h` file, from the original ABC4Trust project, also implemented some functions that were equivalent to the ones available in `multos.h`. We replaced them for the standard functions in the C-API, checking that the MEL code did the same as the documentation specified. This last detail is very important, because in [24] they noted that MULTOS' `ModularExponentiation` function does not accept exponents larger than the modulus size, so they implemented `SpecialModularExponentiation`, dividing the exponentiation in two that MULTOS could perform. After this analysis, when those particularities appeared, our reimplementation of the MULTOS API accepts the *expanded functionality*. In total, we reduced to eighteen the number of MULTOS functions to reimplement, and with future refactoring this number may decrease.

Because of the inherited code, many particularities of the internals of the MULTOS platform had to be addressed.

One was the different memory zones, where the variables in `static` were read and written directly from the secure EEPROM, defining the status of the smart card between executions. We keep those variables in RAM, but save them in a Json file, deserializing them at the smart card boot up, and serializing always before an APDU Response is sent. From the MULTOS documentation, the `static` variables are read and written atomically, and if power source is lost during any process, memory won't get corrupted. This also means that if a MULTOS smart card sends an APDU Response, every variable that had to change, is saved in the EEPROM. Our PoC uses Json for human readability during debugging, but a real deployment should apply secure reads and writes on the file, preventing data corruption and undesired access, for example, ciphering the file with a key derived from the smart card PIN. It is also desirable another data serialization method, reducing the size of the file to store in memory constrained devices.

Another of the mentioned particularities is that the MULTOS compiler does not apply padding between variables in the data structures. For example, in an X86 machine, where a `short` is stored in 2 bytes, in the following code[5]

```
struct MyData
{
```

5 From https://en.wikipedia.org/wiki/Data_structure_alignment

```
        short Data1;
        short Data2;
        short Data3;
};
```

each member of the data structure would be 2-byte aligned. Data1 would be at offset 0, Data2 at offset 2, and Data3 at offset 4. The size of this structure would be 6 bytes. But if we defined the following structure

```
struct MixedData
{
        char Data1;
        short Data2;
        int Data3;
        char Data4;
};
```

where a `char` is 1 byte, and an `int` uses 4 bytes, although the total bytes used are 8 bytes, the compiler will align them in memory adding padding, like if we defined the following:

```
struct MixedData  /* After compilation in 32-bit x86 machine */
{
        char Data1; /* 1 byte */
        char Padding1[1]; /* 1 byte for the following 'short' to
            be aligned on a 2 byte boundary
        assuming that the address where structure begins is an
            even number */
        short Data2; /* 2 bytes */
        int Data3;  /* 4 bytes - largest structure member */
        char Data4; /* 1 byte */
        char Padding2[3]; /* 3 bytes to make total size of the
            structure 12 bytes */
};
```

In MULTOS, to reduce the memory usage, there is no padding, and this affects the inherited ABC4Trust's code because of the use of `memcpy` to copy zones of memory from one address to another. The problem is the code copies multiple variables in one `memcpy` call, because, for example, the APDU Command payload includes multiple data, and instead of copying one variable at a time, the `struct` is defined with the same order and copies everything at once.

An example taken from the parsing of instruction `SET_PROVER`:

```
typedef struct
{
        BYTE prover_id;
        unsigned int ksize;
        unsigned int csize;
        BYTE kx[MAX_SMALLINT_SIZE];
        BYTE c[HASH_SIZE];
        BYTE proofsession[PROOFSESSION_SIZE];
```

```
        BYTE proofstatus;
        BYTE cred_ids[NUM_CREDS];
        BYTE cred_ids_size; // also called 't' in the
            documentation
        BYTE exists;
} PROVER;
/*****/
PROVER provers[NUM_PROVERS];

/*****/
case INS_SET_PROVER:
  // [...]
  memcpy(&(provers[temp_prover_id-1].prover_id), &(apdu_data.
      set_prover_in.prover_id), 5); // under the hood, this also
      initializes ksize and csize
```

This is the only case where a comment showcases that it is copying more than one byte.

The temporal solution is to use `struct __attribute__((__packed__))` to ask a GCC compiler to not use padding in the structs, but this is not standard, neither a good practice. A deeper refactorization of the code is needed where the hidden copies of variables are made explicit, letting the compiler manage the memory layout.

An internal alarm should always warn us when we see raw memory copy from serialized data, that is, the APDU Payload. We mentioned in the MULTOS introduction that the platform is Big Endian. The serialized data is also typically represented in Big Endian, and our APDU Payloads do so. In our code we added a new Subroutine to check whether the device is Big or Little Endian, and to rectify the Endianness from those variables, with 2 or more bytes (the only ones affected by the Endianness), copied from or written to APDU Payloads.

Finally, we must talk about the `multosExit` function. In a MULTOS application, to finish the execution and send the APDU Response, the programmer can call in any moment to `multosExit`, and the MULTOS OS will continue the execution, not returning to the application again.

Think about how we all have, at least once, used an `if` and `return` to avoid the `else`, like in this example:

```
if(condition)
        return a;
return b;
```

The use of `multosExit` in ABC4Trust code is similar, but used in almost every function. This leads to a sequence execution with many possible interruptions, leaving a full stack behind with *un-returned* function calls, i.e., parameters stored in memory after a function call that never returns. In MULTOS, this memory won't affect future ex-

ecutions, because with the next APDU Command, the stack will be loaded with the clean application, and the Session memory stays untouched between APDU Commands, during an APDU Dialogue.

To adapt the code, the reimplemented `mExit` function can not end the program execution, because we would lose the variables in RAM needed during the APDU Dialogue, as well as that the TCP socket would close. The solution is to refactor every use of `multosExit` in a way that, if it is called from a Subroutine, the function returns a success or error code, and the only functions that can call `mExit` are the ones handling an specific instruction.

Nevertheless, there is one exception, the `output_large_data()` Subroutine, which is a tool to send large data payloads without Extended APDU support. We talked about this in 3.4.1, mentioning the special instruction `GET RESPONSE`.

Summarizing, every Subroutine must return from its execution, as well as every instruction handler, which must call either `mExit` or `output_large_data()` before returning. After the instruction has been handled, the execution returns to the listening loop of BIOSC.

INTERFACES

To reimplement some of the MULTOS functions, we needed to use some libraries, so we defined a facade to isolate the implementation of the core smart card from our different options, that could vary depending on the hardware or the system used by the IoT device.

The use of a facade lets us, for example, change the implementation of modular arithmetic with a hardware optimized version, or a future more lightweight library, or our very own software implementation using the same data types that the core uses, minimizing the data usage.

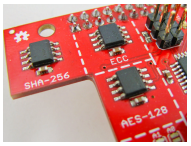
Taking a step forward, we make the core smart card totally independent of any library, only dependant of our interfaces. This means that typical C libraries, like the standard *stdlib.h*, or *string.h* are also behind the facade, in case some IoT system doesn't support them. The main goal we go after with this decision is that future developers adapting the code to a specific platform need to make no change to the *core smart card*'s code, only to the interfaces implementation.

The interfaces defined can be organized in 5 groups (see Figure 14), depending on their purpose: Modular Arithmetic, Cryptography, Memory Management, Serialization, APDU Parsing.

EXTERNAL UTILITIES

If the IoT system offers well tested libraries that could aid in the interfaces implementation, for example, a assembly optimized code for the AES128 and SHA256 operations, these third party libraries belong to this section.

In our PoC, we use two ANSI C libraries, for base64 and JSON, and two shared libraries available in as packages in LEDE, GMPLib and OpenSSL. These libraries use dynamic memory and offer more functionality than we need, although they are a good tool in this PoC, future versions should use more lightweight solutions.



Atmel's cryptography chips.

For example, Atmel's ATAES132A[6] offers a serial chip for secure key storage, AES128 execution and random number generation. Another serial chip like ESP8266 offers WiFi connectivity, typically used with Arduino, and can also perform AES encryption. For random number generation, a technique used with Contiki devices is to read from sensors aleatory data and use it as seed.

All these alternatives depend on the target device, but are all valid, and would substitute OpenSSL with only a Serial communication library, so we can talk to the dedicated chips.

The *interfaces* and *external utilities* sections allow that the project is easily ported to specific targets without modifying the smart card logic.

4.2.3.2 *Execution workflow*

The sequence diagram from Figure 15 shows the execution of the PoC IoT smart card.

The program starts with the `main` function in BIOSC, that deserializes the status from the Json file, and listens on a loop for APDU Commands from the delegation server.

Every time an APDU Command arrives, it calls the function `handle_APDU()` with the raw APDU bytes. The Handler calls the APDU I/O interface to parse the bytes, storing in global variables the APDU structure. Using a `switch-case` expression on the `INS` byte, the Handler calls an *Instruction handler* function.

Inside this function, it may call multiple functions from the Subroutines, that may call MULTOS C-API functions, Which in turn may use an interface to perform its functionality.

As we mentioned, every instruction handler must end, before the `return;` expression, with `mExit` or `output_large_data()`, that will use `mExit`. This reimplemented MULTOS function will save the current status of the smart card, with the help of the serialization interface, and once the file is saved, it outputs the APDU Response back to the delegation server.

After returning from `mExit`, the instruction handler and the APDU handler functions, the program listens again from the socket, until an `0xff` byte is received.

6 ATAES132A 32K AES Serial EEPROM Datasheet - http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8914-CryptoAuth-ATAES132A-Datasheet.pdf

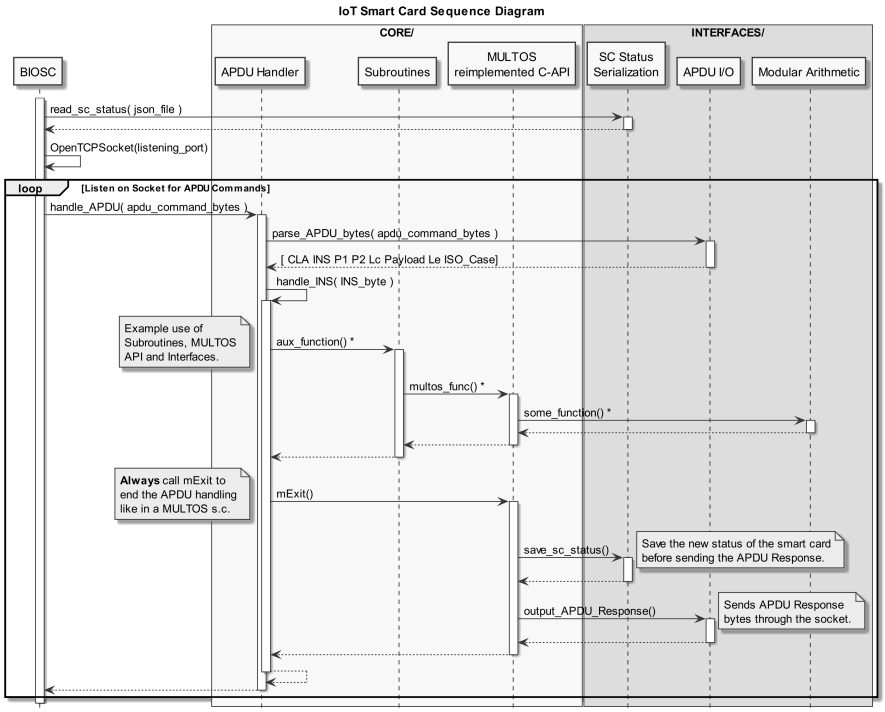


Figure 15: IoT Smart Card Sequence Diagram.

5 VALIDATION AND PERFORMANCE EVALUATION

In this chapter, we will describe the deployment of three testing scenarios: a laptop, a Raspberry Pi 3, and a Omega2 IoT device with the Raspberry Pi 3 as the delegation server. We will measure and compare the results to determine if the proposed solution is feasible or must be submitted to revision.

5.1 TESTBED DESCRIPTION

First, we shall describe the example Attribute Based Credential system in use. Then, the hardware we will use in our benchmarking.

5.1.1 *P2ABCE setting*

To test the correct execution of the *IoT smart card*, we will use the ABC system from the tutorial in the P2ABCE Wiki[1]. It is based on a soccer club, which wishes to issue VIP-tickets for a match. The VIP-member number in the ticket is inspectable for a lottery, ie. after the game, a random presentation token is inspected and the winning member is notified.

First the various entities are **setup**, where several artifacts are generated and distributed. Then a ticket credential containing the following attributes is issued:

```
First name: John
Last name: Dow
Birthday: 1985-05-05Z
Member number: 23784638726
Matchday: 2013-08-07Z
```

During **issuance**, a *scope exclusive pseudonym* is established and the newly issued credential is bound to this pseudonym. This ensures that the ticket credential can not be used without the smart card.

Then **presentation** is performed. The *presentation policy* specifies that the member number is inspectable and a predicate ensures that the matchday is in fact $2013-08-07Z$. This last part ensures that a ticket issued for another match can not be used.

The ticket holder was lucky and his presentation token was chosen in the lottery. The presentation token is therefore inspected.

1 https://github.com/p2abcengine/p2abcengine/wiki/

5.1.2 *Execution environment*

First we will execute the test in our development machine (laptop). After asserting that the services work as expected, we then run the test in a Raspberry Pi 3, exactly like in the laptop. Finally, we will deploy the IoT smart card in a Omega2 and the delegation services in the Raspberry Pi 3. After every test, we checked that the issuing and proving were successful, in case a cryptographic error appeared in the implementation.

Lets have a closer look at the hardware of each device:

DELL XPS 15

DEVELOPMENT LAPTOP Our device is a DELL XPS 15, with a Core i7-6700HQ at 3.5GHz quad core processor and 32GB of DDR4 RAM, running Ubuntu 16.10.

This is a powerful machine that can simulate the performance of many servers and clients that would implement P2ABCE in a real environment, giving a reference point for performance comparisons.

Raspberry Pi 3

RASPBERRY PI 3 A familiar environment, powerful enough to debug and hold the delegated P2ABCE Java services of P2ABCE with its 1GB of RAM, and with two network interfaces, perfect to work as the gateway for the IoT devices to the Internet.

Only a microSD with enough space to burn the OS is needed to plug&play with the Raspberry Pi. We use Raspbian, a stable Debian based distro, recommended by the Raspberry Pi designers, and ready to use with the P2ABCE compiled *self-contained .jar* services.

CPU	ARMv8 64bit quad-core @1.2GHz
RAM	1GB
Storage	microSD
Firmware	Raspbian (Debian based distro)
Connectivity	Wifi n + Ethernet
Power	5V 2A

Table 1: Raspberry Pi 3 Specifications.

Onion Omega2

ONION OMEGA2 A device that falls inside the category of IoT, powerful enough to run a Linux embedded environment, such as LEDE, where we can develop and debug the first PoC without troubling ourselves with problems not related to the project itself.

Nonetheless, the Omega2 needs fine tunning to start operating, and basic knowledge of electronics to make it work. The two main things to begin with Omega2 are:

MCU	Mediatek MT688[2]
CPU	MIPS32 24KEc 580MHz
RAM	64MB
Storage	16MB
Firmware	LEDE (OpenWRT fork distro)
Connectivity	Wifi b/g/n
Power	3.3V 300mA

Table 2: Onion Omega2 Specifications.

- A reliable 3.3V with a maximum of 800mA power supply, e.g. a USB2.0 with a step-down circuit, with quality soldering and wires to avoid unwanted resistances. The Omega2 will usually use up to 350mA, when the WiFi module is booting up. The mean consumption is about 200mA.
- A Serial to USB adapter wired to the TX and RX UART pins to use the Serial Terminal, to avoid the use of SSH over WiFi.

A downside of using LEDE and the Omega2 is the lack of hardware acceleration for cryptographic operations, unlike the MULTOS applications, because the smart cards hardware and MULTOS API include support for such common operations in smart cards.

THE NETWORK In our third scenario, the Raspberry Pi 3 and the Omega2 will talk to each other over TCP. This implies possible network delays depending on the quality of the connection. The Raspberry Pi 3 is connected over Ethernet to a switch with WiFi access point. The Omega2 is connected over WiFi n to said AP. To ensure the delay wasn't significant, we measured 6000 APDU messages, and the results show that the mean transmission time is less than half a millisecond per APDU. As we will see in the results section, this network time is negligible.

FUTURE WORK FOR TESTS The lack of a physical MULTOS smart card precludes us to load and test the ABC4Trust Card Lite's code and measure the time P2ABCE would need when using the *HardwareSmartcard* class. This would be really interesting because for a single method from the *Smartcard* interface, *HardwareSmartcard* implementation needs to send multiple APDU Commands, but *SoftwareSmartcard* can perform the operations immediately, with the full computer's resources. Also, physical smart cards benefit from hardware acceleration in most of the cryptographic operations, unlike our software implementations in the PoC.

5.2 THE TEST CODE

In this section we present the scripts and binaries used during the tests.

There are three pieces of software that conform the test: the P2ABCE services, the IoT smart card, and shell scripts automatizing the REST calls, from the terminal.

P2ABCE SERVICES

This is a common part to our three sets. The services are compiled in a self-contained Jetty web server, or in WAR format, ready to be deployed in a server like Tomcat. We use the same JAR files with the embedded Jetty web server for the PC and Raspberry Pi.

We modified the User Service code to measure the execution elapsed time for each REST method. We don't measure the time the web server spends processing the HTTP protocol and deciding which Java method to call.

IOT SMART CARD

Our C implementation of the P2ABCE smart card, compiled for the Omega2, with BIOSC listening on port 8888 for the APDU messages.

We tested the execution in two modes, a full logging where every step was printed in terminal, and another one with no logging. With the first mode, we can check a proper execution, every byte exchanged, and with the second one, we measure the execution without unnecessary I/O.

SHELL SCRIPTS

To orchestrate all the services we use shell scripts that execute the REST calls using `curl`. Here we perform the mentioned steps: setup of the P2ABCE system, issuance of the credential and proving for the presentation policy.

In the setup, the system parameters are generated, indicating key sizes of 1024 bits, the ones currently supported by the ABC4Trust and IoT smart cards. Then the system parameters and public keys from the services (issuer, inspector, revocation authority) are also exchanged and stored in each Service.

During the issuance and proving, the User and the Issuer and Verifier exchange multiple XML files, with the cryptographic information of each step of the Idemix protocol.

We offer two versions for the scripts. The first one is a single shell script file, to be executed from a single terminal. The advantage of this version is that, because every XML file is stored in the machine running the script, we don't have to transfer these files by other means, e.g. `scp`. An excerpt of this script showcases how we receive an XML file from the User Service, e.g. `secondIssuanceMessage.xml`, and send

it to the Issuer Service, referencing the file stored in the script's working directory:

```
[...]
# First issuance protocol step - UI (first step for the user).
echo "Second issuance protocol step (first step for the user)"
curl -X POST --header 'Content-Type: text/xml' -d
    @uiIssuanceReturn.xml 'http://localhost:9200/user/
    issuanceProtocolStepUi/' > secondIssuanceMessage.xml

# Second issuance protocol step (second step for the issuer).
echo "Second issuance protocol step (second step for the issuer)"
curl -X POST --header 'Content-Type: text/xml' -d
    @secondIssuanceMessage.xml 'http://localhost:9100/issuer/
    issuanceProtocolStep/' > thirdIssuanceMessageAndBoolean.xml
[...]
```

The second script version is intended for a more realistic execution. In a real scenario, the User and the Issuer, for example, would call the REST methods of their respective Services, but not the other actor's Service, exchanging the XML files through other means. Instead of one script running in a single machine, orchestrating the Omega2, Raspberry Pi 3 and laptop, we offer three scripts, one per machine, with *pauses* where one device must send an XML file to another machine. An excerpt from the Omega2's script shows the same step as above, but instead of calling the Issuer's REST Service, waits until we send the corresponding files.

```
# First issuance protocol step - UI (first step for the user).
echo "Second issuance protocol step (first step for the user)"
curl -X POST --header 'Content-Type: text/xml' -d
    @uiIssuanceReturn.xml "http://$Rpi3IP:9200/user/
    issuanceProtocolStepUi/" > secondIssuanceMessage.xml

read -p "Send => secondIssuanceMessage.xml <= back to the Issuer.
     Then press any key to continue... " -n1 -s

read -p "Receive => thirdIssuanceMessageAndBoolean.xml <= from
    the Issuer. Then press any key to continue... " -n1 -s
```

The first version is recommended for tests, and the second version only for instructive purposes.

5.3 RESULTS

After 20 executions for each scenario (laptop, RPi3, Omega2+RPi3), we take the means and compare each step of the testbed.

It is worth noting that during the test, the measured use of the CPU showed that P2ABCE does not benefit of parallelization, therefore, it only uses one of the four cores in the laptop and Raspberry Pi 3.

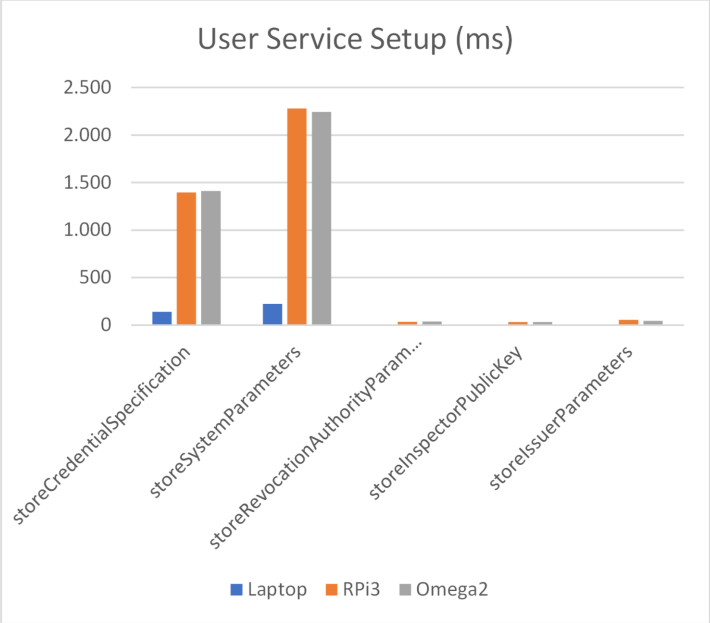
To test the network, we sent six thousand APDUs to the Omega2, but instead of calling the *APDU handler*, the Omega2 responded with the same bytes back. This way, the Omega2 only performed the simple BIOSC protocol, reading from and writing to the TCP socket. The APDUs had multiple sizes, taken from the most common APDUs logged in during a successful execution. The test showed that our network speed was around 0.014 ms per byte.

THE SETUP

The first step of our testbed. The Omega2 doesn't intervene until the creation of the smart card, therefore, the times measured in the second and third scenarios are practically identical.

	storeCredentialS	storeSystemPara	storeRevocation	storeInspectorPu	storeIssuerParar
Laptop (ms)	139.23	222.08	5.05	5.62	5.75
RPi3 (ms)	1395.12	2278.85	35.38	33.76	56.10
Omega2 (ms)	1412.29	2244.54	38.61	33.59	44.77
Laptop over RPi3	10.02	10.26	7.01	6.01	9.75

(a) Times and relative speedup



(b) Comparison graph

Figure 16: Setup times (milliseconds)

As we can see in Figure 16, the laptop is about ten times faster than Raspberry Pi 3, but considering that the highest time is less than two and a half seconds, and that the setup is done only once, this isn't a worrisome problem.

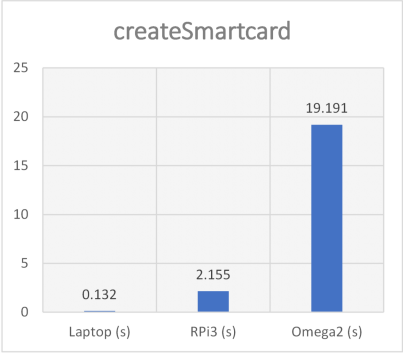
CREATION OF THE SMART CARD

Here we create a *SoftwareSmartcard* or a *HardwareSmartcard* object that the User service will use in the following REST calls.

The REST method to create a *SoftwareSmartcard* is `/createSmartcard`, and to create a *HardwareSmartcard*, using the *IoTsmartcardio* implementation, we use `/initIoTsmartcard`. This operation is done only once per device, and includes commands from the creation of the PIN and PUK of the smart card, to storing the system parameters of P2ABCE, equivalent to the previous setup step.

	createSmartcard
Laptop (s)	0.132
RPi3 (s)	2.155
Omega2 (s)	19.191
Laptop over RPi3	16.36
RPi3 over Omega2	8.91
Laptop over Omega2	145.68

(a) Times and relative speedup



(b) Comparison graph

Figure 17: Create smart card times (seconds)

From Figure 17 we see that the RPi3 is about 16 times slower than the laptop in the creation of the *SoftwareSmartcard*, but almost 9 times faster than the setup of the smart card in the Omega2 using APDUs. This gives us that the laptop is 145 times faster than the combination of RPi3 and Omega2 in our IoT deployment. But looking at the times, this process lasts up to 20 seconds, making it something feasible.

This is the first interaction between the RPi3 and the IoT smart card running in the Omega2. To setup the smart card **30 APDU Commands**, and their respective Responses, are exchanged, as shown in Appendix A, Figure 23, with a total of 1109 bytes. From our network benchmark, using TCP sockets, the delay in the transmission is only around 15 and 20 ms, negligible, as we said, compared to the almost 20 seconds the operation lasts.

ISSUANCE OF THE CREDENTIAL

Our credential will have 5 attributes and key sizes of 1024 bits, as specified during the setup process.

The issuance is done in three steps for the User service, shown in Figure 18 with a red note showing the start of each step for the User delegation, in green for the interactions between Issuer and User, and the darker red is the Identity service, choosing the first available identity to use. The arrows in the figure show the REST calls performed during the test, where the IoT device acts as User, the RPi3 hosts the P2ABCE delegation services and the laptop is the Issuer.

The three delegation steps and the REST method called are:

First issuance protocol step	/issuanceProtocolStep
Second issuance protocol step *(end of first step for the User)*	/issuanceProtocolStepUi
Third issuance protocol step *(second step for the User)*	/issuanceProtocolStep

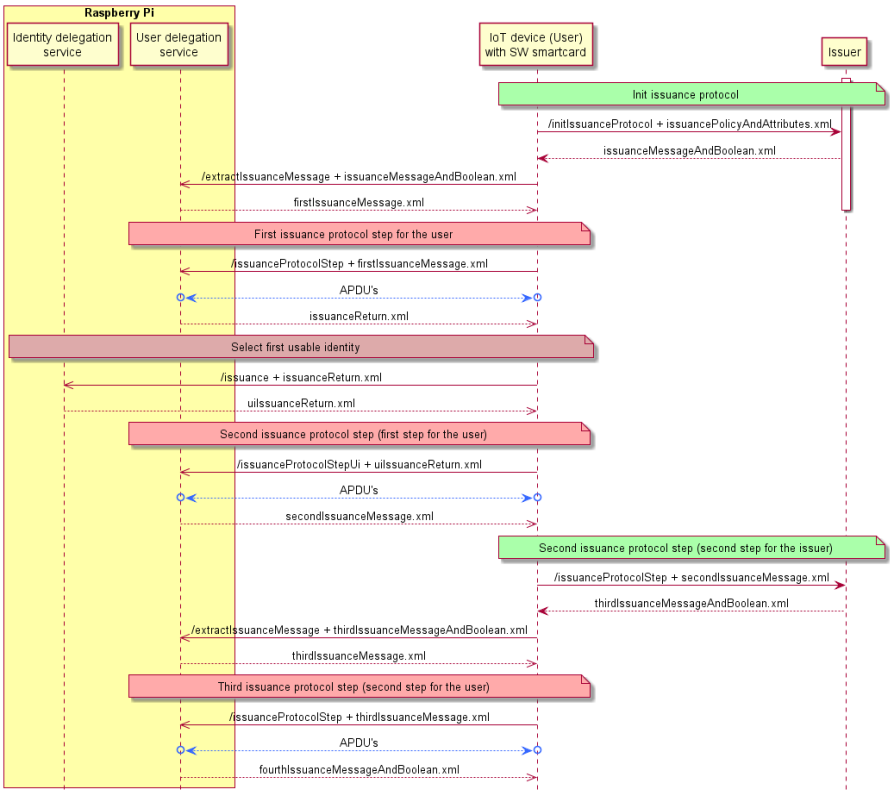


Figure 18: Issuance interaction.

As we can see, the three REST calls to the delegation User service involve communication with the smart card. We show in Appendix A, Figure 24, the APDU Commands used for each REST call in the issuance. There are 45 APDU Commands in total, 3197 bytes exchanged, that would introduce a latency of 45ms in the network, negligible.

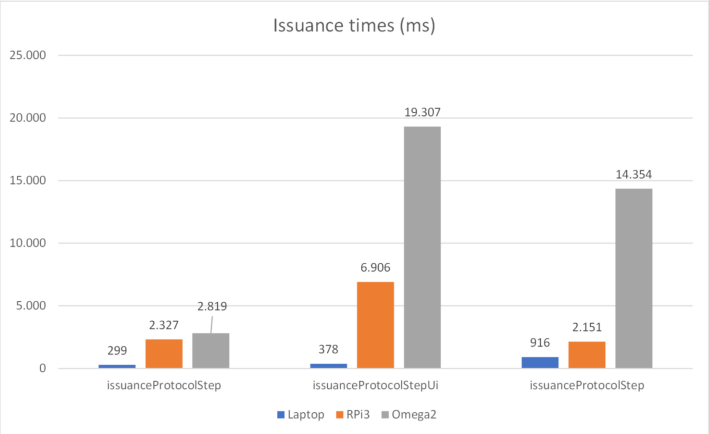
In Figure 19 we have the times spent in each REST call. The laptop shows again to be many times faster than the other two scenarios, but the times are again feasible even for the IoT environment.

Lets compare the Raspberry Pi 3 and the Omega2 executions. There is a correlation between the number of APDU Commands needed in each step with the increment in time when using the IoT smart card, that is, how much the delegation server.

The first one only involved one APDU, with 33 bytes total (Command and Response), and times are almost identical. The second call

	issuanceProtocolStep	issuanceProtocolStepUi	issuanceProtocolStep
Laptop	298.79	377.84	916.32
RPi3	2327.18	6905.93	2150.90
Omega2	2818.64	19307.44	14354.24
Laptop over RPi3	7.79	18.28	2.35
RPi3 over Omega2	1.21	2.80	6.67
Laptop over Omega2	9.43	51.10	15.67

(a) Times (ms) and relative speedup



(b) Comparison graph

Figure 19: Issuance times (milliseconds)

needed 34 APDUs, with 1623 bytes, and the increase in time is around tree times slower than the RPi3 on its own. The third call used 20 APDUs, 1541 bytes, and makes the IoT scenario almost 7 times slower.

The analysis shows where there are more cryptographic operations involving the Omega2, and because the amount of data exchanged is minimal, the difference in processing power between Omega2 and Raspberry Pi 3 is clear.

PRESENTATION TOKEN

The final step of the test involves a Prove, or Presentation in P2ABCE, where the Verifier sends the User or Prover the Presentation Policy, and the User answers with the Presentation Token, without more steps. In Figure 20, using the same colors as in the Issuance interaction, we can see the delegation messages done by the Omega2.

To ensure that all the process was successful, it's enough to check if the Verifier and the Inspector returned XML files, accepting the prove, or an error code. Of course, every execution measured in the test was successful.

In Appendix A, Figure 25, we provide the APDU Commands for each step, 28 in total, with 1939 bytes, giving us about 27ms of delay in the network transmission.

Again, as shown in Figure 21, there is a correlation between the number of APDU Commands used, the work the IoT smart card must perform, and the time measured. The 20 APDU Commands in the

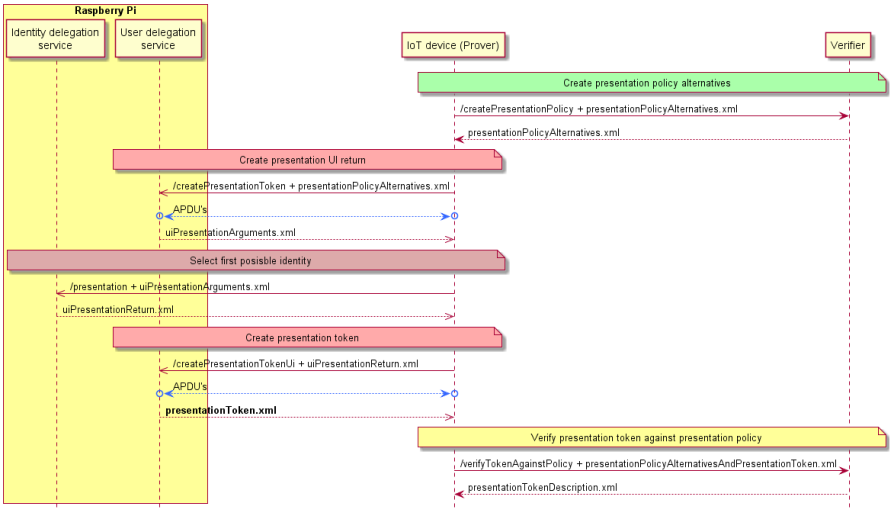


Figure 20: Proving interaction.

first call make the IoT deployment almost 8 times slower than the Raspberry Pi 3; but with only 8 APDU Commands, the second one is less than 1.5 times slower.

Nonetheless, it's significant the difference in performance between the laptop and the Raspberry Pi 3 in the last REST call, more than 40 times slower, even using the *SoftwareSmartcard*.

Unlike the previous steps, the Presentation or Proving is done more than once, being the key feature of ZKP protocols. The laptop performs a prove in less than one second, the RPi3 needs 15 seconds, but our P2ABCE IoT deployment needs 15 seconds for the first step, and 18s for the second step, 33 seconds total to generate a Presentation Token.

MEMORY USAGE ON THE OMEGA2

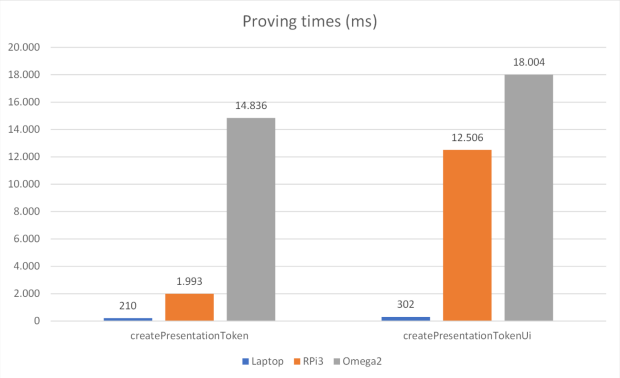
Using the tool *time -v* we can get a lot of useful information about a program, once it finishes. In our case, the binary with BIOSC and the smart card logic starts as an empty smart card, goes through the described process, and then we can stop it, as the User Service won't use it anymore.

After another round of tests, now using *time -v*, the field named *Maximum resident set size (kbytes)* shows the **maximum** size of RAM used by the process since its launch. In our case, this involves the use of static memory for the *global variables* of the smart card logic, and the dynamic memory used by the third party libraries, like GMPlib, OpenSSL and cJSON.

GMP and OpenSSL always allocate the data in their own ADT, what involves copying the arrays of bytes representing the big modular integers from the cryptographic operations. cJSON, used in the

	createPresentationToken	createPresentationTokenUi
Laptop	209.78	301.66
RPi3	1993.11	12505.80
Omega2	14836.33	18003.75
Laptop over RPi3	9.50	41.46
RPi3 over Omega2	7.44	1.44
Laptop over Omega2	70.72	59.68

(a) Times (ms) and relative speedup



(b) Comparison graph

Figure 21: Proving times (milliseconds)

serialization of the smart card for storage, and debug being human readable, stores a copy of every saved variable in the JSON tree structure, then creates a string (array of char) with the JSON, that the user can write to a file.

Understanding the many bad uses of memory done in this PoC is important for future improvements and ports. A custom modular library using the same array of bytes that the smart card logic, a binary serialization, and many improvements, are our future work.

With all that said, the mean of the maximum memory usage measured is 6569.6 kbytes. Compared to the 64MB of RAM available in the Omega2, our PoC could be executed in more constrained devices, given the system is compatible.

6 CONCLUSIONS AND FUTURE WORK

To finish this document, we sum up some conclusions from the work done, and results obtained. We will also enumerate some future lines of research.

6.1 CONCLUSIONS

In the memory of this project we try to show the work done from the beginning, but we only showed our right decisions, and the information that is significant for the final result. The truth is that, aside from the information included in this paper, we have worked with other systems that ended up discarded. This isn't a negative aspect, because if we didn't, for example, study the Contiki OS, Cooja simulator and the hardware used, we could not be sure the development of the first PoC for that system would be impossible in the time given.

With regard to the work presented, the flexibility of the computation offloading technique, identifying the key operations that can be delegated, and those ones that can't, has allowed us to define a general solution for the vast world of the Internet of Things. The IoT devices can operate as individual actors in the P2ABCE ecosystem, and when in need of performing computation offloading, the delegation server can also be a device considered into the IoT class.

During the development, we had to investigate a lot of concepts related to IoT, smart cards, and even the insides of P2ABCE's code, to fix many existing bugs in the original project and minimize the amount of changes it had to undergo, in order to work with the IoT devices. In the implementation chapter we give guidelines to port MULTOS applications, considering the particularities we encountered, that's why we can't consider it an *instruction manual to port MULTOS apps*, but an interesting reading on how to confront a similar project. For example, if we wanted the Idemix implementation from [24], previous to P2ABCE, in a IoT device, we could apply almost the same steps, obtaining a core functionality of Idemix for constrained devices in C.

Our PoC implementation demonstrates that this project is actually feasible, not by performing a simulation of an IoT device, like in [10]. However, the use of third party libraries and no hardware acceleration support, makes the PoC too slow for certain cases, e.g., a Real-Time system requires almost immediate operations, like a car warning that another one is approaching too fast. There is a long path of

research before we can see this design in production, as many decisions depend on the specific deployment in consideration.

Recalling the objectives listed in Section 3.1, we think we accomplish them, except the implementation of a PoC in the most constrained device possible, where we used LEDE to ease our development, and the last objective, where we could have given a use to the PoC, and we leave as part of the future work. In spite of this negative auto-critic, we have to acknowledge that this is the first privacy-preserving ABC implementation for IoT, designed for extensibility, interoperability and maintenance.

Finally, we are very thankful to IBM's grant for the *Privacy Preserving Identity Management applied to IoT* project, which allowed us to get this far. Also, this paper will be presented for publication in a JCR magazine.

6.2 FUTURE WORK

Due to the nature of the project, there exist many options to continue researching in this area. We list below a collection of the tasks we couldn't accomplish but would have been our next steps:

- Second PoC for Arduino systems

  We have already developed a working PoC for Linux based systems, and from our objective of using the most constrained devices available, we can follow an iterative process, building for devices a little more constrained than the last one. The next natural step, after building with LEDE/OpenWrt, is to aim for Arduino.

  The critical changes to the actual PoC are:

  The transmission channel, if the device has no TCP/IP stack, it could use a Serial connection to transmit BIOSC and delegation messages.

  The use of a Json file to store the smart card status depends on whether the device has enough space or a file system. The current Json parser also uses too much dynamic memory. At this stage the serialization should be performed in a binary format, in a file or writing variables directly to the EEPROM memory.

  GMPLib and OpenSSL libraries must be replaced by others with similar capabilities, or special implementations for the particular device in use.

  All these changes affect only the *external utilities* and *interfaces* sections of the code, leaving the *core* untouched, because the smart card logic doesn't change.

- Implement more P2ABCE functionality in the device

  If instead of aiming for more constrained devices, we work with similar devices to the Omega2, the computational power of the device could assume all the functionality of P2ABCE, at least for some roles. In a M2M (machine to machine) environment, the devices could act as Provers and Verifiers in the same interaction, and the verifying logic could be implemented entirely in the device (generate a Presentation Policy and checking a prove in a Presentation Token is valid), reducing the need of a delegation server.

- Improve the *core smart card* code

  From the simple refactoring of removing the MULTOS reimplemented API for direct calls to the *interfaces*, to identify all the `memcpy` copies of multiple variables with one call, removing the compiler directive for not using memory padding in the defined structures, letting the compiler manage the memory automatically.

- Use cryptographic hardware

  Atmel offers many solutions for secure memory and cryptographic operations chips, with serial interfaces for the IoT devices. We could benefit from this chips by storing the secret keys of the credentials inside the secure memory, and speed up the SHA256 and AES128 calculations, currently software implemented and slow compared to any hardware implementation.

- Support Extended APDUs

  Currently, the APDU instruction handlers use the GET RESPONSE command to send large APDU Response Payloads, but we can reduce the number of APDUs exchanged supporting the Extended APDU format. Because the P2ABCE already sent some Extended APDUs, the APDU I/O interface in the PoC has support for Extended APDU Commands, but not for Extended Responses.

- Integrate the IoT+Idemix solution in FIWARE

  Our sixth objective, that we couldn't get to design in depth. The original idea was to integrate in a FIWARE[1] environment the Fiware Keyrock and Idemix Issuer, so both, users and IoT devices' identities are managed in the IdM[2] system. The Issuer generates credentials based on the user's attributes held in the Keyrock, and IoT devices can authenticate against Fi-ware services (using Idemix), or authenticate against other IoT devices (M2M).

1 FIWARE https://www.fiware.org/
2 Identity Management - KeyRock https://catalogue.fiware.org/enablers/identity-management-keyrock

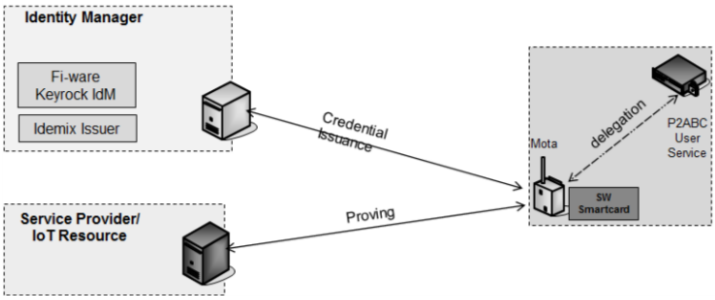


Figure 22: IoT+Idemix Fi-Ware integration.

- Research P2ABCE policies for IoT applications

  The next step, after integrating Idemix with an Identity Management system, should be to design more complex applications to IoT scenarios. We talked about a smart building in the introduction, but the straightforward proofs we can present depend on the signed values in the credentials. The Zero-Knowledge Proofs cryptography is already implemented, we only have to extend the protocols to benefit from them.

  For example, a combination of authentication and arbitrary proof could be achieved by a combined proof of knowledge of a signed credential (current authentication system) and said arbitrary proof, although the values used in it aren't actually signed by the Issuer, the first proof gives credibility.

  Another solution for the same problem, similar to a PKI scheme, the IoT device receives a signed credential including an attribute which is his own public key. The IoT device can issue new credentials with the parameters it needs for the arbitrary proofs. The Verifier then certifies the new credential is signed by the IoT device's public key.

APPENDIX

A

TEST: APDU COMMANDS EXCHANGED

In this appendix we provide 3 sequence diagrams highlighting the APDU Commands exchanged during our testbed.

The first one is the setup of the IoT smart card, storing the system parameters needed to work in the deployed P2ABCE system. The first Command is called `isAndroid` because the `HardwareSmartcard` class distinguishes Android phones acting as smart cards from MULTOS smart cards, in order to use Extended APDUs. Then configures the smart card, setting the PIN ("`1234`" by default) and PUK. Finally, it copies the cryptographic system parameters to the smart card with multiple `SET` instructions.

The second image shows the issuance of the credential, divided in the three REST calls needed during the delegation. If we recall Idemix's issuance protocol, the User performed, during his first step, an exponentiation with his secret key and a ZKP, which can be identified by the `COMMITMENT` and `ISSUANCE RESPONSE` commands. During the second step, the User only has to verify the Issuer's ZKP and store the credential if the signature is valid. Because the P2ABC Engine can perform Verification of ZKP without the need of secret keys stored in the smart card, the APDU Commands in this phase are only for storing the credential information inside the IoT smart card's BLOB[1] database, with the `STORE BLOB` instruction.

The last diagram represents the proving for the Presentation Policy received from a Verifier, using two REST calls. During the first REST call, the Service stores data for the proving in the smart card, but before starting the proving itself, the device must choose an identity for the proving. This is done with the Identity Service, that in the PoC chooses the first identity available. After this, the second REST call starts the proving, creating the *commitment* and *responses* that integrate any ZKP, and depends on the device's secret keys[2].

1 (**B**inary **L**arge **Ob**jects

2 We, again, refer the reader who wants to know more about ZKPs to [16].

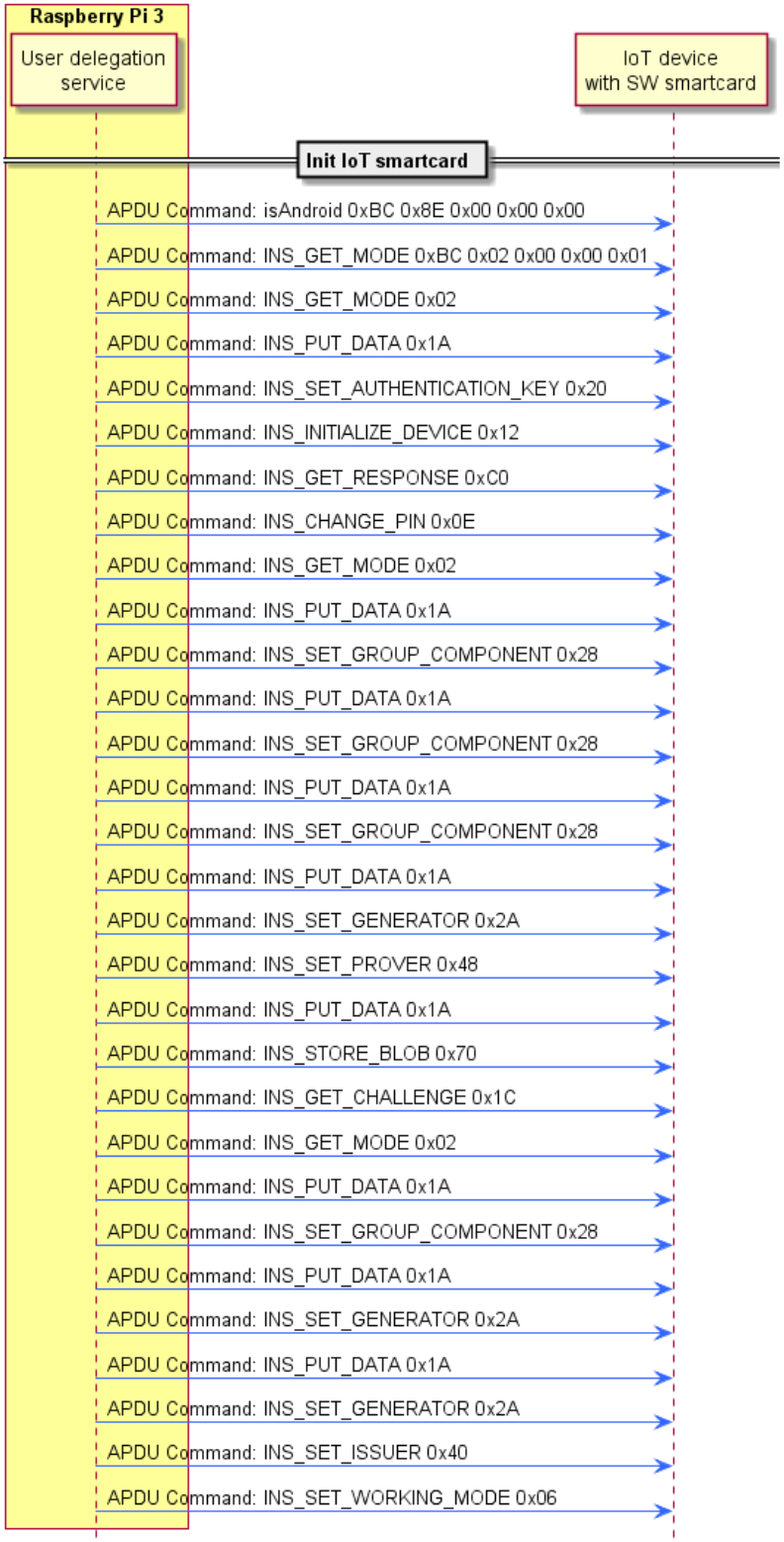


Figure 23: Init IoT Smart Card APDU Commands exchanged.

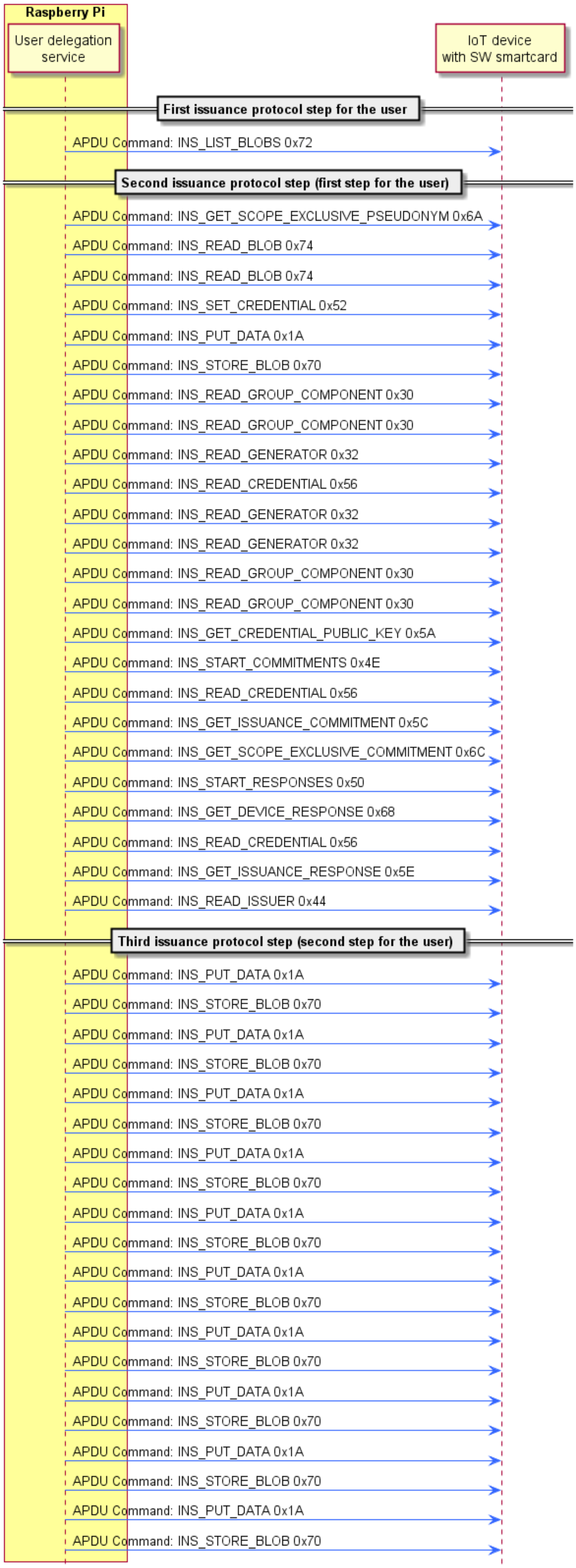


Figure 24: Issuance APDU Commands.

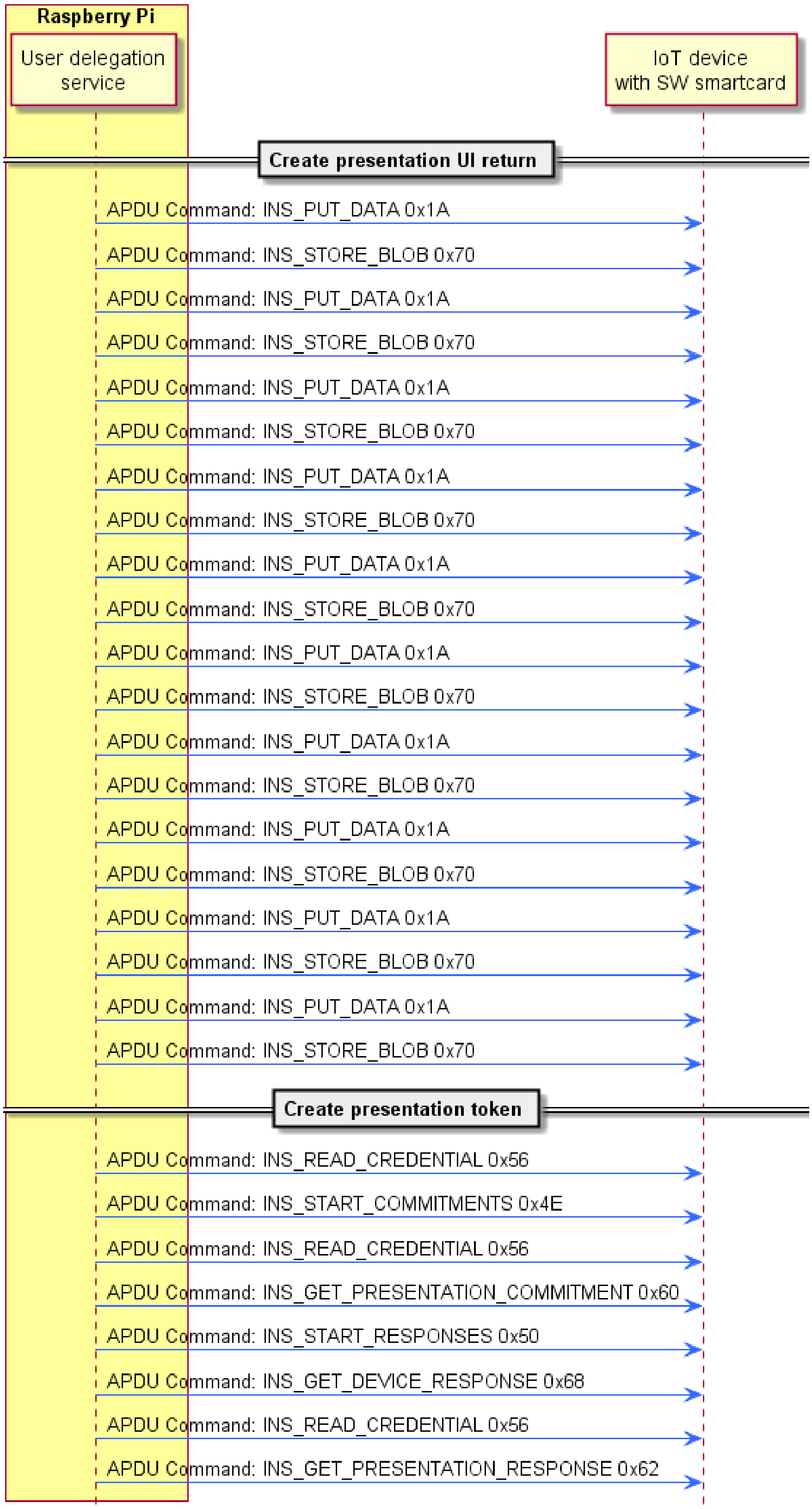


Figure 25: Proving APDU Commands.

B

ATTACHED CODE STRUCTURE

Given the size of the project, we identify four main directory structures to arrange the code.

First, the Dockerfiles, scripts that automatize the creation of Docker images. These images are like virtual machine snapshots, although Docker is not a virtualization system, and we use the images to create containers, that are like new virtual machines started from the snapshots. We use two Dockerfiles, one for the P2ABCE compilation environment, with Java and Maven, and another one for building the Omega2 binaries, using the LEDE SDK.

```
Docker/
├── P2ABCE/
│   ├── Dockerfile
│   └── idemix-3.0.36-binaries/
└── LEDE SDK/
    └── Dockerfile
```

The original P2ABCE project is of considerable size, but we list here the most interesting directories that we had to modify to make the project compatible with IoT smart cards.

```
P2ABCE/
└── Code/
    └── core-abce/
        ├── abce-components/
        │   └── src/main/java/eu/abc4trust/smartcard/
        │       ├── IoTsmartcardio/
        │       │   ├── util/
        │       │   │   └── IoTsmartcardConnection.java
        │       │   ├── IoTCard.java
        │       │   ├── IoTCardChannel.java
        │       │   └── IoTCardTerminal.java
        │       ├── HardwareSmartcard.java
        │       └── SoftwareSmartcard.java
        └── abce-services/
            └── src/main/java/eu/abc4trust/services/
                ├── UserService.java
                ├── VerificationService.java
                └── IssuanceService.java
```

The IoT smart card toolkit is the core of this project. Managed with CMake, we see the `CMakeLists.txt` files, including the `Toolchain-omega2-mipsel.cmake` file with the cross-compiler information. We divide the project be-

tween the *source* C files, and the header C files. Inside these directories, the organization is as stated in the implementation chapter: *core* or *common*, *util interfaces* and *external utilities*.

```
p2abce_iot_toolkit/
├── CMakeLists.txt
├── Toolchain-omega2-mipsel.cmake
├── util_sc_tests/
└── util_sc/
    ├── CMakeLists.txt
    ├── BIOSC.c
    ├── p2abc_iot_toolkit_src/
    │   ├── smartcard_common/
    │   │   ├── APDU_handler.c
    │   │   ├── global_vars.c
    │   │   ├── m_adapted_API.c
    │   │   └── subroutines.c
    │   ├── smartcard_external_utilities/
    │   │   └── cJSON.c
    │   └── smartcard_utils_interface/
    │       ├── APDU_IO_util.c
    │       ├── arithmetic_util.c
    │       ├── crypto_util.c
    │       ├── serialize_util.c
    │       └── system_funcs.c
    └── p2abc_iot_toolkit_include/
        ├── error_codes.h
        ├── macrologger.h
        ├── smartcard_common/
        │   ├── APDU_handler.h
        │   ├── APDU_types.h
        │   ├── abc4T_types.h
        │   ├── defs_consts.h
        │   ├── defs_errs.h
        │   ├── defs_ins.h
        │   ├── defs_types.h
        │   ├── global_vars.h
        │   ├── m_adapted_API.h
        │   └── subroutines.h
        ├── smartcard_external_utilities/
        │   ├── base64.h
        │   └── cJSON.h
        └── smartcard_utils_interface/
            ├── APDU_IO_util.h
            ├── arithmetic_util.h
            ├── crypto_util.h
            ├── serialize_util.h
            └── system_funcs.h
```

Finally, the test scripts, including a Python file to test the BIOSC protocol, a directory with the testbed scripts (one file to orchestrate every device from one terminal), and a more didactic version with a script per device.

```
scripts/
├── testBIOSC.py
├── tests/.....................................One test script
│   ├── setupTest.sh
│   ├── testNoIoT.sh
│   └── IoTtest.sh
└── distributed-PoC/...........................Distributed test
    ├── Omega2/
    ├── RaspberryPi3/
    └── ThirdPartyServer/
```

BIBLIOGRAPHY

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey." In: *Computer Networks* 54.15 (2010). http://www.sciencedirect.com/science/article/pii/S1389128610001568, pp. 2787 –2805. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2010.05.010.

[2] P. Bichsel, J. Camenisch, T. Grob, and V. Shoup. *Anonymous credentials on a standard Java Card. CCS*. 2009.

[3] S. Brands and C. Paquin. *U-Prove cryptographic specification v1.0. Tech. rep.* Tech. rep. Microsoft, Mar. 2010.

[4] S.A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy. MIT Press*. Aug. 2000.

[5] Jan Camenisch and Anna Lysyanskaya. "An Eficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation." In: *Advances in Cryptology EUROCRYPT 2001* (2001).

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

[7] Jan Camenisch and Markus Stadler. "Eficient Group Signature Schemes for Large Groups." In: *Advances in Cryptology CRYPTO 97* (1997).

[8] Pascal Paillier CryptoExperts. *ABC4Trust on Smart Cards. Embedding Privacy-ABCs on Smart Cards*. Summit Event, Brussels, Jan. 2015.

[9] Luuk Danes. "Smart Card Integration in the Pseudonym System Idemix." MA thesis. Faculty of Mathematics. University of Groningen, 2007.

[10] J. M. de Fuentes; L. González-Manzano; J. Serna-Olvera and F. Veseli. "Assessment of attribute-based credentials for privacy-preserving road traffic services in smart cities." In: *Personal and Ubiquitous Computing Journal, Special Issue on Security and Privacy for Smart Cities* (Feb. 2017). http://www.seg.inf.uc3m.es/~jfuentes/papers/PrivacyABC-VANET.pdf.

[11] Tom Henriksson. "How strong anonymity will finally fix the privacy problem." In: *VentureBeat* (Oct. 2016). https://venturebeat.com/2016/10/08/how-strong-anonymity-will-finally-fix-the-privacy-problem/.

[12] *ISO/IEC 7816-4:2005 Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*. https://www.iso.org/standard/36134.html.

[13] Víctor Sucasas Iglesias. "IMPLEMENTATION OF AN ANONYMOUS CREDENTIAL PROTOCOL." MA thesis. Escuela Técnica Superior de Ingenieros de Telecomunicación. Universidad de Vigo, 2008-2009.

[14] Ari Juels (eds.) Jack R. Selby (auth.) *Financial Cryptography: 8th International Conference, Revised Papers*. 1st ed. Lecture Notes in Computer Science 3110. http://gen.lib.rus.ec/book/index.php?md5=24CD58AE88D5564A03C2E44B96732298. Springer-Verlag Berlin Heidelberg, 2004, pp. 196–211.

[15] Louis C. Guillou Jean-Jacques Quisquater and Thomas A. Berson. "How to Explain Zero-Knowledge Protocols to Your Children." In: *Advances in Cryptology - CRYPTO '89* (1990). Proceedings 435: 628-631. http://pages.cs.wisc.edu/~mkowalcz/628.pdf.

[16] José Luis Cánovas Sánchez. Tutores: A. J. Pallarés y Leandro Marín. "Pruebas de Conocimiento Cero y sus Aplicaciones." http://www.jlcs.es/ZKP.pdf. 2017.

[17] Adrián Sánchez Martínez. *La cueva.* Image.

[18] R. Thandeeswaran N. Jeyanthi. *Security Breaches and Threat Prevention in the Internet of Things*. IGI Global, 2017.

[19] Gregory Neven. *A Quick Introduction to Anonymous Credentials*. IBM Zürich Research Laboratory, Aug. 2008.

[20] Zhiyun Qian, Z. Morley Mao, Ammar Rayes, David Jaffe (auth.), Muttukrishnan Rajarajan, Fred Piper, Haining Wang, and George Kesidis (eds.) *Security and Privacy in Communication Networks: 7th International ICST Conference, SecureComm 2011, London, UK, September 7-9, 2011, Revised Selected Papers*. 1st ed. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 96. Springer-Verlag Berlin Heidelberg, 2012, pp. 243–260. URL: http://gen.lib.rus.ec/book/index.php?md5=A7667DB3EE5B36AAA6E93AE07E888D8D.

[21] *Specification of the Identity Mixer Cryptographic Library (v2.3.43)*. Tech. rep. IBM Research, Jan. 2013.

[22] Mark Stanislav and Tod Beardsley. *HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities*. Tech. rep. Rapid7, 2015.

[23] M. Sterckx, B. Gierlichs, B. Preneel, and I. Verbauwhede. *Efficient implementation of anonymous credentials on Java Card smart cards. In: WIFS*. 2009.

[24] Pim Vullers and Gergely Alpar. "Efficient selective disclosure on smart cards using idemix." In: *IFIP Working Conference on Policies and Research in Identity Management*. Springer. 2013, pp. 53–67.

DECLARACIÓN DE ORIGINALIDAD

Yo, José Luis Cánovas Sánchez, autor del TFG INTEGRACIÓN DE IDEMIX EN ENTORNOS DE IOT, bajo la tutela de los profesores Antonio Fernando Skarmeta Gómez y Jorge Bernal Bernabé, declaro que el trabajo que presento es original, en el sentido de que ha puesto el mayor empeño en citar debidamente todas las fuentes utilizadas.

Murcia, Junio 2017

José Luis Cánovas Sánchez