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

Short summary of the contents in English... a great guide by Kent Beck how to write good abstracts can be found here:

https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

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

1 INTRODUCTION

The Internet of Things (IoT) is a term with a wide range of interpretations [3], breafly, we can think of it as a set of devices, mainly resource constrained, that are interconnected between them in order to achieve a goal. For example, a network of lampposts 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 [23], or the multiple vulnerabilities affecting house devices, like baby monitors [29].

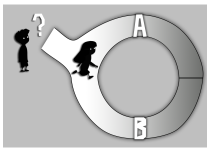
To address this problem of privacy in the Internet, a recent approach is the concept of *strong anonymity*, that conceals our personal details while letting us continue to operate online as a clearly defined individuals [14]. To achieve it, we must address a way to perform authentication and authorization 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)` [24]. The most straightforward way for the User to convince a Verifier of her list of attributes would be to simply transmit her credential 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. Thats why current solutions rely on Zero-Knowledge Proofs (ZKPs) , cryptographic methods that allow to proof knowledge of some data without disclosing it.

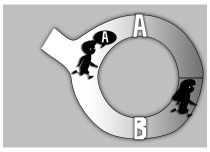
To read more about ZKPs, refer to my Mathematics thesis [31].

To understand how ZKPs work, in 1990 Guillou, Quisquater and Berson published in *How to Explain Zero-Knowledge Protocols to Your Children* [18] 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.

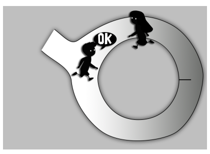


The cave [21] . Peggy takes randomly A or B. Victor awaits outside.

> 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.
>
> **P**eggy 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.
>
> 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.
>
> Victor then asks to repeat the experiment. With 20 repetitions, the chances Peggy fools Victor in all of them is only 2^{-20},



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



The cave. Peggy returns by the requested path.

> **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[1], 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."

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 [26]. This gives us a standardized language to exchange Idemix's messages between IoT devices and any other P2ABCE actor.

1.1 OUTLINE OF THIS THESIS

TODO: organización, basada en los objetivos

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

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 known 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 [17]. 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 [6] 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 [5] 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 [27].

Jan Camenisch, Markus Stadler and Anna Lysyanskaya studied in [9], [7] and [8] the cryptographic bases for signature schemes and anonymous credentials, that later became IBM's Identity Mixer protocol specification [28].

Luuk Danes in 2007 studied theoretically how Idemix's User role could be implemented using smart cards [11], 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 [16], 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 [4] and [30], 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 [33], 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 [13]. 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 execute on hardware actually used in VANET systems, although the OS may not be the same.

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 ANALYSIS AND OBJECTIVES

In this chapter we describe the project objectives, and the methodology followed during its development. In section ... TODO

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**, starting from the principal goal, dividing our work in different categories.

IDEMIX AND THE IOT This is our main objective, integrating IBM's privacy-preserving system in the IoT environment.

ANALYSIS Study the state of the art of Idemix and the IoT, analysing related projects, papers with similar approaches, and consider the best fitting solutions from where to begin.

DESIGN AND IMPLEMENTATION After studying the existing works, we must give a formal solution to be implemented. This includes the theoretical architecture, the steps to take, the software to implement, the hardware to use, and any simplifications to leave as future work.

Our **software** objectives include making it easily maintainable, structured and extensible, 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, with minimal effort, given new IoT systems or changes in the cryptographic protocols.

Our **hardware** objective is to use devices as constrained as possible. But considering our early situation in the project, we will use IoT devices that ease testing and development, having in mind the other devices, and what specific steps we should take for them in the future.

VALIDITY AND EVALUATION Deploy a Proof of Concept (PoC) in a **real scenario**, without the need of simulators, checking it works as expected, and measuring its performance.

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, analyze every part of it and categorize them. We will consider what components are mandatory to be executed in the target device that wants to act as User of Verifier, and which components the devices would actually be able to execute.

Devices with equivalent processing power to smart phones are capable of running the current Java implementations of P2ABCE with Idemix, but the most constrained IoT devices can not handle the entire system, only the mandatory cryptographic operations.

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 [33].

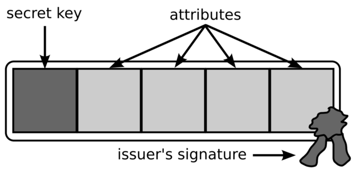


Figure 1: A first look at an attribute-based credential with four attributes.

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:

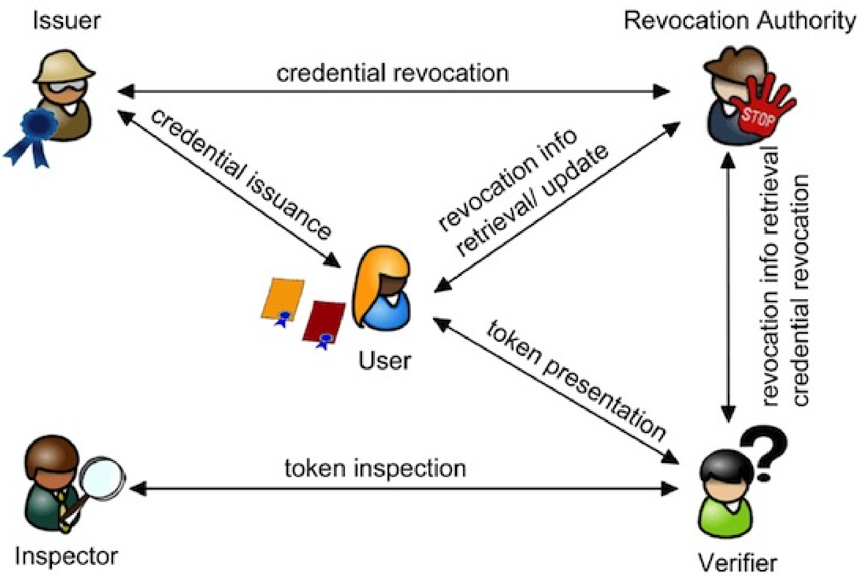


Figure 2: Entities in a P2ABC System

- **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.

- **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.

The cryptographic nature of the credential-as-container concept includes the following four security aspects [33]:

- The issuer's digital signature ensures **authenticity**: the credential originates from the issuer, and this issuer asserts that the attributes hold for the person.

- This signature also guarantees integrity: the attributes contained in the credential have not been altered since they were issued.

- A credential is **non-transferable** as it is bound to the secret key, only known by the credential owner.

- A credential **hides** its content, so it does not reveal the attributes it contains. Furthermore, a credential protects the privacy of its owner through the following cryptographic properties.

- Issuer **unlinkability** ensures that any information gathered during issuing cannot be used to link a verification of the credential to its issuance.

- **Multi-show unlinkability** guarantees that when a credential is verified multiple times, these sessions cannot be linked.

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.

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.

In the P2ABCE repository [26] 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.

3.3.1 *P2ABCE Code Structure and REST Services*

The Java code is managed by a Maven project, structured using various known design patterns, but not of our interest. 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 credentials. 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.

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 (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.

The project also provides a set of REST services to control each role of the P2ABCE systeam (User, Issuer, Verifier, Inspector, etc.). The methods implemented include the creation of *Software* smart cards

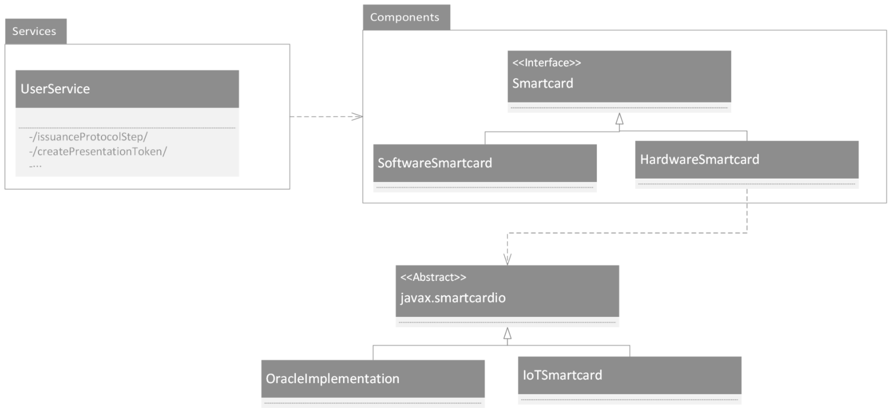


Figure 3: Basic P2ABCE structure

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, for example, but for the User actor to call the Engine through the User Service, to perform specific actions, and then the Service returns the response XML. Those XML files are the way to communicate two different actors. The transmission method to exchange the XML depends on the specific scenario.

3.3.2 *ABC4Trust Card Lite*

As a PoC the P2ABCE project includes a smart card reference implementation, the ABC4Trust Card Lite [10]. 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.

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 have in count:

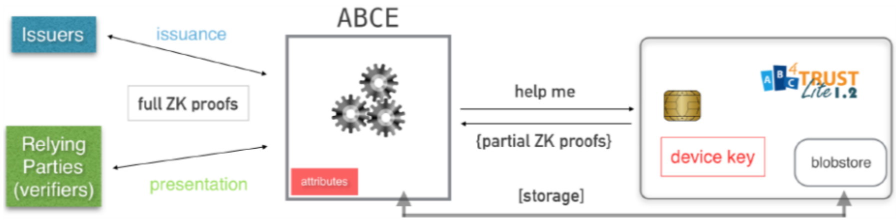


Figure 4: ABC4Trust Card Lite

The best asset of this code is that it's 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 [33]. 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, they are pointers to shared buffers with fixed sizes, avoiding the use of any dynamic memory allocation.

Although, among the many drawbacks, we could highlight the *awful* coding style, the strong dependency on MULTOS framework and many bugs found.

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, with practically no comments; and to conclude, the implementation of thirty functions called *Subroutines* at the end of the file, 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, and use some communication protocol to remotelly 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 and applies many refactoring techniques to the code, it becomes the best starting point for the IoT version, making us opt for the second option.

3.4 PRELIMINARIES ABOUT SMART CARDS

In this section we will introduce a brief description of the APDU standard protocol, and the MULTOS framework used by the ABC4Trust Card Lite.

3.4.1 *Smart Card Communication Protocol*

To communicate the smart cards and the reader the `ISO/IEC 7816-4` [15] 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.

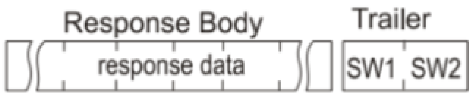


Figure 6: APDU Response

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.

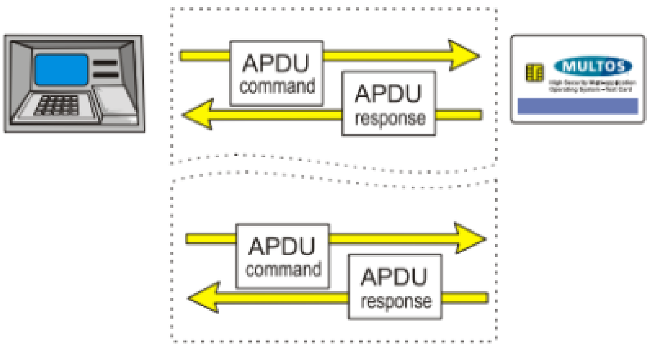


Figure 7: APDU Command-Response Dialogue

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 [20]. 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 we can see, 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.

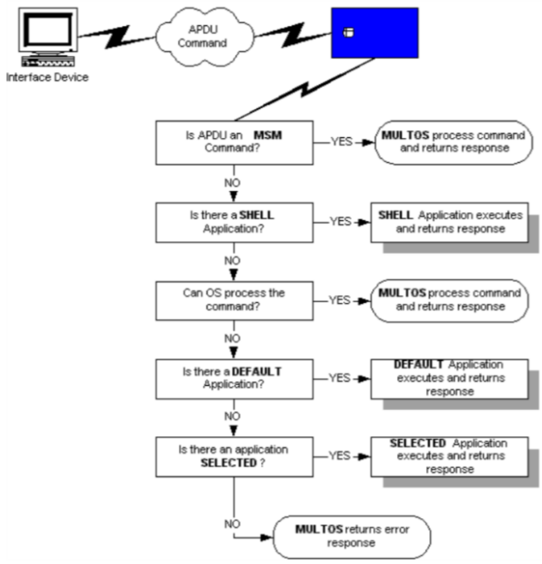


Figure 8: MULTOS Workflow

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

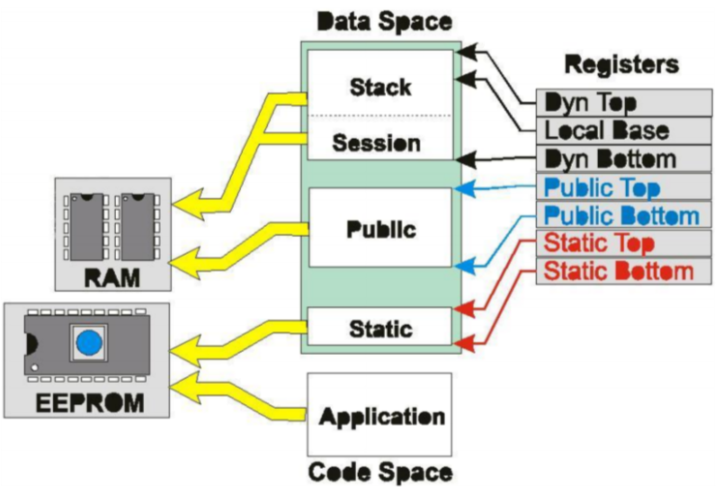


Figure 9: MULTOS Memory Layout

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

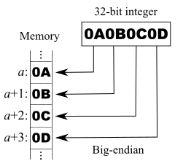
Static memory are the application variables declared after the specific *#pragma melstatic* compiler directive. These variables are stored in the non-volatile secure EEPROM, and any write is assured to be saved because they are not loaded into RAM.

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 [12] 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)

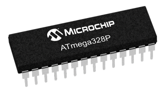
Figure 11: MULTOS Data Types

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 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. Therefore, no implementation for these tools is available, nor in C, nor in assembly code.

Nevertheless, the C-API documentation [20] provides rich description for each function.

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, Linux Embedded Development Environment [19], using the Onion Omega2 development board. Although the Omega2 uses LEDE, its microcontroller is also listed as compatible with ThreadX OS [32], a Real-Time Operative System for embedded devices, 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.

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 to deploy containers ready to compile the 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.
- 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 [19], configured for the Omega2 target, the device chose for the PoC.

The Dockerfiles and CMake cross-compilation toolchain can be found in Appendix A.

4 DESIGN AND IMPLEMENTATION

In this chapter we will describe the process for defining how a constrained IoT device may be integrated in a system like P2ABCE. We will also describe the PoC implementation carried out to test a realistic deployment.

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

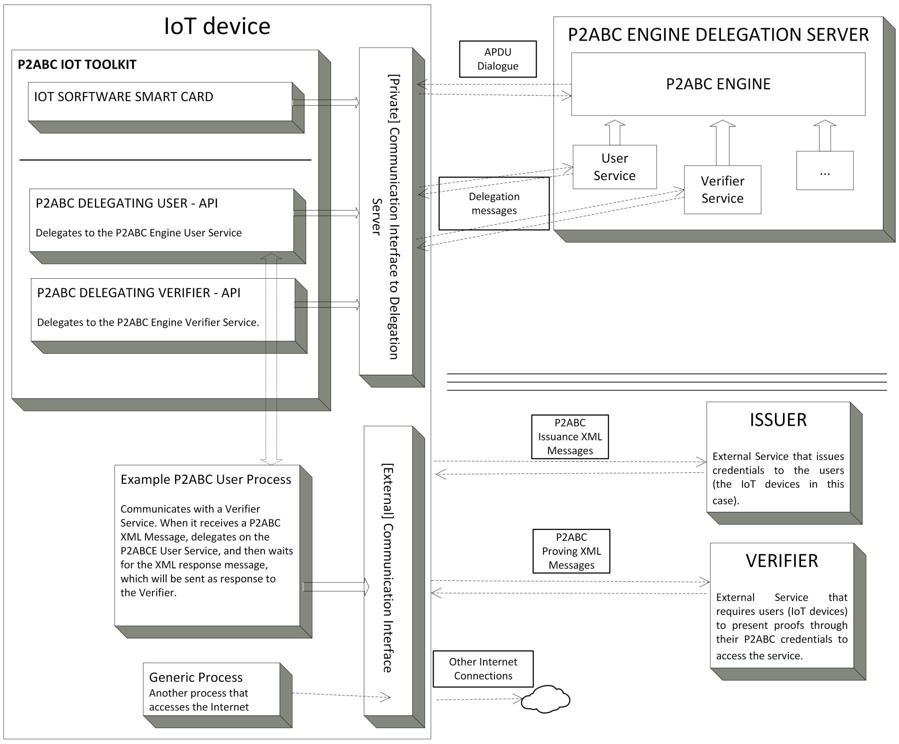


Figure 12: IoT in P2ABCE Architecture.

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 starts an interaction with another actor, e.g. an Issuer or Verifier, receiving a P2ABCE XML file.

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.

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.

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

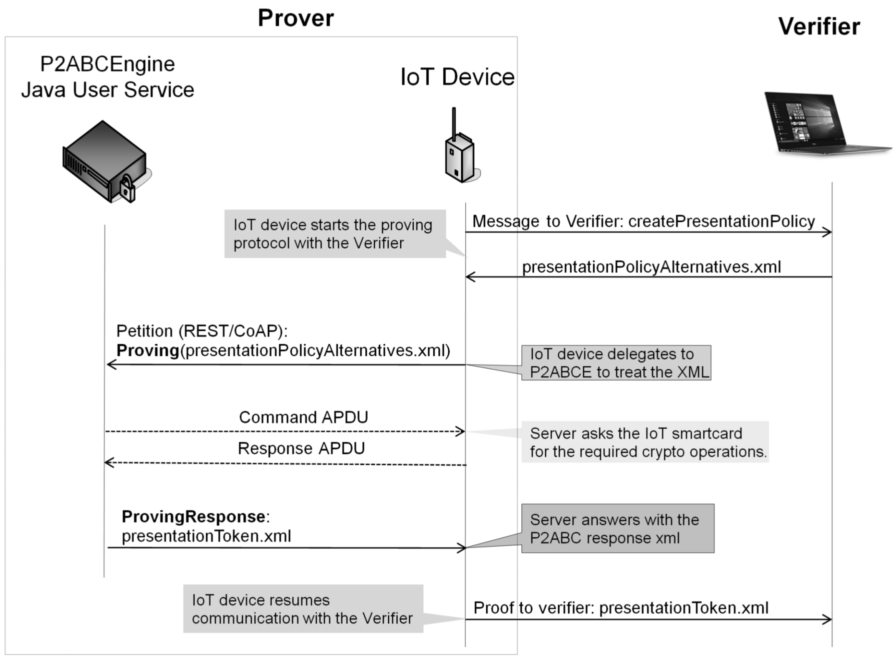


Figure 13: IoT Delegation in P2ABCE for Proving.

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.

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.

MODIFICATIONS 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[3] 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

1 Time-based One-time Password
2 Two-factor authentication
3 http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/overview.page

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

POC DELEGATION TO THE P2ABCE SERVER

We now will address each type of delegation: from the IoT device as a P2ABCE actor to the P2ABCE delegation server, and from the server to the smart card in the IoT device.

Currently P2ABCE offers multiple REST web services to run different roles in P2ABCE system: User Service, Issuer Service, Verification Service, etc. An application that integrates P2ABCE can make use of this services in the same machine or implement the functionality using the core components written in Java, the same ones the REST services use. Our PoC machine, the Omega2, can make REST calls easily, but other devices may use Constrained Application Protocol (CoAP), and in that case, the P2ABCE REST services should be adapted to offer CoAP support. The commands needed to delegate to the P2ABCE delegation server will be the same to operate with the REST services. This way, the first issue is solved.

The transmission of the messages will depend on the specific use cases, capabilities and resources available. If the delegation server is connected, for example, through RS-232 serial to the IoT device, it benefits from the same level of security as the physical device, reducing the security considerations of the delegation channel to that of the devices security, the communication is simple, and not far away from the Arduino Yun idea of combining two devices, one more powerful but to use only when needed. But if the IoT device and the delegation server are apart, or more than one IoT device delegates to it, then the transmission must be secured. They could use 6LoWPAN to talk to each other (the delegation service could be deployed in the proxy) and then secure communications with existing solutions, like with pre-shared symmetric keys, certificates for authentication and authorization, etc., it depends on each particular deployment.

At the end of the day, this is all about usual security in IoT. Many other studies focus on this matter, so we will assume it can be done, and focus on what's new, P2ABCE in IoT.

To sum up, our IoT device will act as User (Prover or Verifier) keeping its secrets in a software smart card. When it starts an interaction with other actor of the P2ABCE system (Issuer, Verifier, etc.), the IoT device will delegate with a remote call (using REST in our PoC) to a P2ABCE delegation server, attaching the XML file and the necessary information for the server to send the APDUs to the software smart card (in our PoC using TCP sockets, giving the IP and listening port).

This simple design keeps the benefit of a 100% compatible P2ABCE deployment, and the integration of IoT devices to the P2ABCE ecosystem.

In the future, more functionality currently delegated in the P2ABCE server can be implemented in the IoT device, if its resources allow it. For example, in a M2M environment, where an IoT device can act as User and Verifier, the verification consists on sending a Presentation Policy, and verifying the Presentation Token, which implies less logic than generating it as the User. Therefore, the implementation of the Verifier functionality would reduce significantly the need of a delegation server, but as we said, managing complex XML squemas is not something many IoT devices could do.

4.3 IOT SMART CARD

After many design decisions in the process to adapt the original ABC4T Card Lite code to pure C, working over a more usual architecture machine, in this section, we present the current PoC code, most important decisions, workflow execution, and future work.

First, let's define what a *more usual architecture* is. If we remember the MULTOS section, the framework gives an application a very specific memory layout and entry and output points of execution, that could be seen as a single process execution machine. Many IoT devices work like a computer, with multiple processes or threads, without pre-loaded data on startup (like the APDU MULTOS loads for the application), a non-volatile memory for data and code, maybe a basic file system in this memory, and RAM with the program's stack, heap, data and code.

Our PoC is tested on a Linux system, and we will give instructions on how to adapt each part to work with other typical IoT systems. For example, other IoT devices may work like MULTOS and let access variables in non-volatile memory during execution, and in that case, the port should be changed according to these particularities.

4.3.1 *Code structure*

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

The first section is what could be called as the core of the smart card, the second one the interface for the tools the core need and may depend on the platform, and finally third party libraries, that in may be empty if the interfaces implementation doesn't need any.

In our PoC we used CMake to manage the project, due to the cross-compilation tools, integration with multiple IDEs and tests.

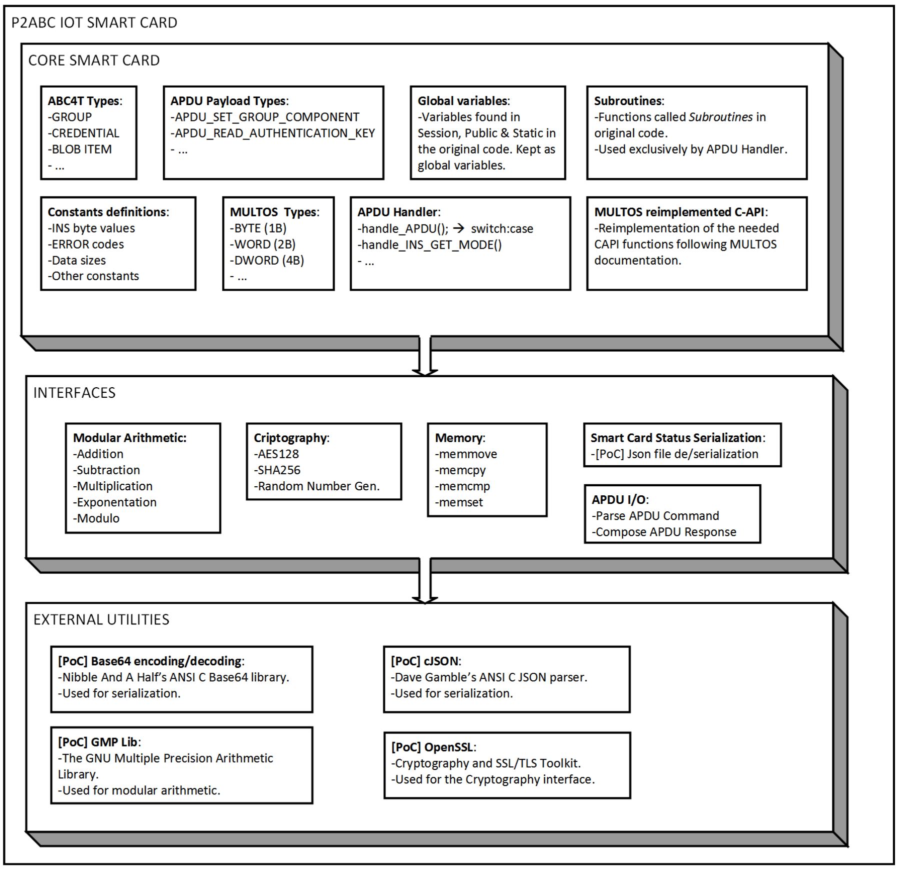


Figure 14: IoT Smart Card Code Structure.

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

Changes in the APDU protocol for P2ABCE must be done here, independently of the target platform.

After refactoring the original ABC4Trust Card's code, most of it fell in what we will call the core of the smart card.

All types and variable definitions and the APDU handling is done in this code. However, the ABC4Trust's code depended on the MULTOS C-API for the input/output of data, modular arithmetic, and even AES128 and SHA256 cryptography.

A characteristic of MULTOS C-API is that every function name starts with *multos*, but as we said, the *main.h* file implemented equivalent functions to some available in *multos.h*. Our first step was to replace the *main.h* functions for the standard ones in the C-API. Then, we implemented, following the C-API documentation, the functions from *multos.h* (only the used ones) changing their names from *multosFoo()* to *mFoo()* for readability and emphasize that they were no longer from MULTOS.

Future changes in the code may refactor it so there's no longer need for the MULTOS framework functions.

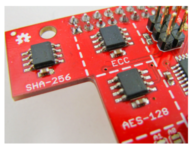
INTERFACES To implement MULTOS functions, we needed to use some libraries, so we defined a facade to isolate the implementation of the core smart card from our different options, that could vary depending on the hardware or the system used by the IoT device.

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 transformations needed.

Taking a step forward, we make the core smart card totally independent of any library, only on 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.

EXTERNAL UTILITIES If the IoT system offers well tested libraries that could aid in the interfaces implementation, or we simply found a pure C implementation for the task, 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. The last two libraries offer more functionality than we need, hence, it's desired in a production code to implement *Modular Arithmetic* and *Cryptography* interfaces with more lightweight alternatives.



Atmel's cryptography chips.

For example, Atmel's ATAES132A [2] 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. The *interfaces* and *external utilities* sections allow for a clean and fast port of the code.

4.3.2 *PoC Workflow*

In a real deployment, we would offer a full library with an API to other processes to delegate on the P2ABC Engine, that automatizes the listening and security, that we presented in the deployment diagram **TODO:ref-img** as the *P2ABCE IoT Toolkit*.

But to test the IoT smart card, we use the *curl* command for REST delegation calls and the BIOSC (Basic Input Output Smart Card) for APDU communication.

To transmit the APDU messages in our PoC we use a simple protocol, consisting in one first byte for the instruction: receive an APDU Command or close the connection. In the first case, then we read two header bytes with the length of the APDU Command or Response to receive, followed by those APDU bytes. The message is sent over TCP for a reliable transmission (concept of session, packet retransmission, reordering, etc.).

We lack of any security (authentication and 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. This belongs to *usual* security, as we already said, and for that reason it's not in the PoC.

P2ABCE REST DELEGATION The only P2ABCE Service that needed to be modified was the User Service. We added the REST call

/initIoTsmartcard/issuerParametersUid?host=&port=

where we communicate the P2ABCE server that a IoT Smart Card is accessible via *host* and *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 *IoTsmartcardio* implementation for the *HardwareSmartcard* constructor.

The rest of P2ABCE code is unchanged and will work as if a real smart card was in use. *IoTsmartcardio* implementation will transmit the APDUs through the TCP socket with the format mentioned.

Our test consists on the tutorial available in P2ABCE's repository. From the Omega2's terminal we run *curl* commands to send REST petitions. For example:

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

The IoT device will only manage XML as data files to exchange between the third party actor and the P2ABCE delegation service, without parsing them.

After a REST call to the User Service, the P2ABC Engine will talk to the IoT smart card process.

APDU TRANSMISSION WITH BIOSC Before sending the REST message, the Omega2 must be listening in the specified port. At start up, BIOSC reads from a JSON file the status of the smart card (credentials in the BLOB, private keys, PIN and PUK codes, etc.), then opens the TCP socket and listens in a loop.

When the delegation server sends an APDU Command following the simple protocol, BIOSC stores in a byte array the APDU Command bytes and calls the *handle_APDU* function in *Core Smart Card*.

The *APDU_handler* will parse the APDU bytes and check both CLA and INS bytes. For each possible INS the is a function that must always finish calling either *mExit()* or *output_large_data()*.

The *mExit* function is the reimplementation of MULTOS C-API exit functionality, that finishes the application execution returning to the MULTOS OS, that will send the APDU Response bytes to the terminal. This led in the original ABC4Trust Card's code to some tricky situations. Imagine a function that ends with an if-else expression. Many times we save a line writing something like

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

instead of the complete

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

This is a swift example of what the *multosExit* function led to, because it is called in both the *switch-case* processing the APDU instruction, and the *Subroutines*. For example, the PIN code check was programmed in a way every reason to fail finished the application execution, with different error codes. Another example, checking if a credential is stored, if not, fail and exit inside the *readBlob* subroutine.

In our standard architecture machine, if we call a subroutine, it must end, and return the control to the handling function, and so the *APDU_handler* must return to the listening loop of BIOSC.

Having all that in count, the *mExit* function is not implemented as the documentation specifies, but only saves the smart card status serializing it in the JSON file again, and once it's saved, parses the APDU Response and sends it to the socket.

The *output_large_data* function is a tool used in MULTOS smart cards that don't support Extended APDUs. The APDU Command handler saves in a buffer the data to send, and its size. Then, *output_large_data* will manage the buffer expecting future GET RESPONSE commands from the P2ABCE server.

A future change to the project may include support for Extended APDUs, avoiding the use of multiple GET RESPONSE. To do this, the *output_large_data* function and the *IoTsmartcardio* implementation of *javax.smartcardio* must support the feature.

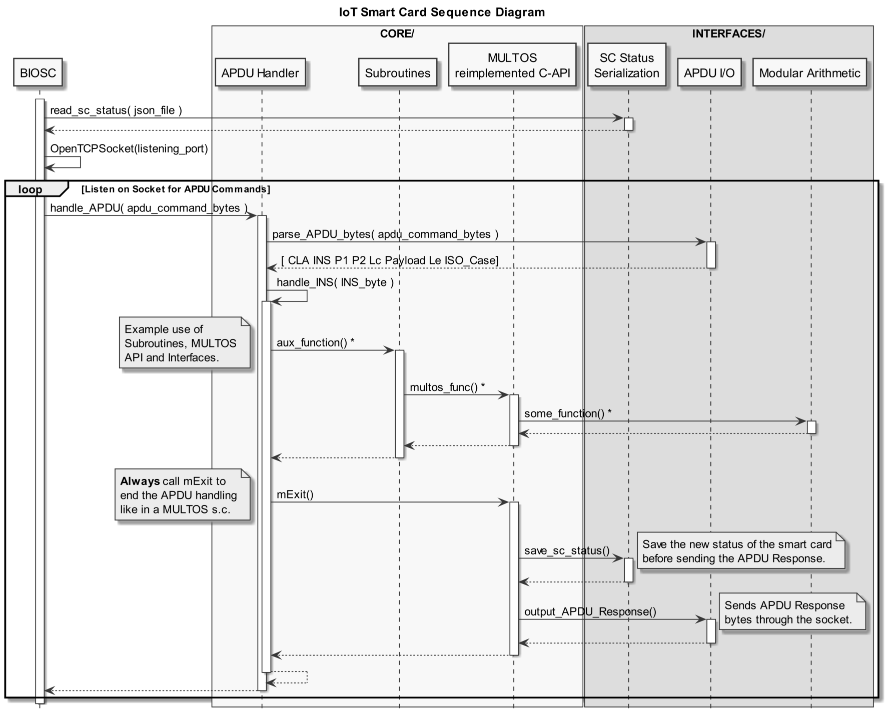


Figure 15: IoT Smart Card Sequence Diagram.

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 [25]. 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.

5.1.2 *Execution environment*

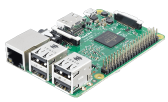
First we will execute the test in our development machine (laptop). After asserting that the services work as expected, we then run the test in a Raspberry Pi 3, exactly like in the laptop. Finally, we will deploy the IoT smart card in a Omega2 and the delegation services in the Raspberry Pi 3. After 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 environment, 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 [22]
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.

Also, we will need a cross-compiler to generate the smart card binary. Because we will use GMP and OpenSSL as shared libraries, the best option is to use the LEDE SDK. The SDK manages the available packages and generates the GCC toolchain. Also, using CMake,

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.

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 SCRIPT

To orchestrate all the services we use a simple script that performs the REST calls using *curl*. Here we perform the mentioned steps: setup of the P2ABCE system, issuance of the credential and a prove 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.

In the issuance, a two step protocol is performed by the User. The Issuer and the User send each other the XML data with the cryptographic information, and finish the protocol with a credential issued inside a smart card, software (Java) or hardware (physical or IoT).

Finally, a Verifier sends the User a *presentation policy*. The User generates a *presentation token* for the Verifier and Inspector.

There are two script versions, one where every REST call is done from the same machine, therefore avoiding the exchange of the XML files by other means (like *scp*); the other one is ready to be executed on a real setup, two scripts, one for the Omega2, where the REST calls are the delegation on the Raspberry Pi 3, and other for the PC, running the Issuer and Verifier as third party entities. The XML files generated by each service can be sent with the same protocol the IoT device and the Verifier, for example, would communicate in the real world. This way it's clearer that the *curl* commands are the delegation protocol for the IoT device, and then the IoT device can send the XML as it wishes, over the Internet, Bluetooth, NFC, etc.

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.

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

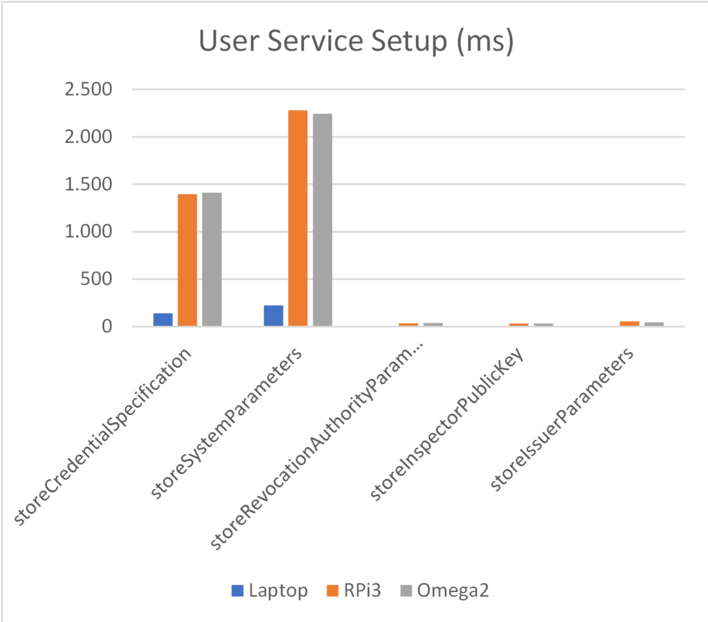
Here we create a *SoftwareSmartcard* or a *HardwareSmartcard* object that the User service will use in the following REST calls.

The REST method to create a *SoftwareSmartcard* is `/createSmartcard`, and to create a *HardwareSmartcard*, using the *IoTsmartcardio* implementation, we use `/initIoTsmartcard`.

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.

	storeCredentialS	storeSystemPara	storeRevocation	storeInspectorP	storeIssuerParam
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)

Again, 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.

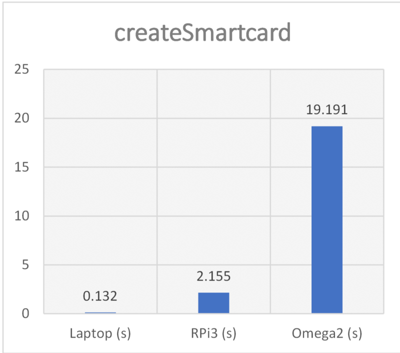
This is the first interaction between the RPi3 and the IoT smart card running in the Omega2. To setup the smart card **30 APDU Commands**, and their respective Responses, are exchanged, as shown in Appendix B, Figure 22, with a total of 1109 bytes. From our network benchmark, using TCP sockets, the delay in the transmission is only around 15 and 20 ms, negligible, as we said, compared to the almost 20 seconds the operation lasts.

ISSUING OF THE CREDENTIAL

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.

	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)

The three delegation steps and the REST method called are:

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

As we can see, the three REST calls to the delegation User service involve communication with the smart card. We show in Appendix B, Figure 23, 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 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

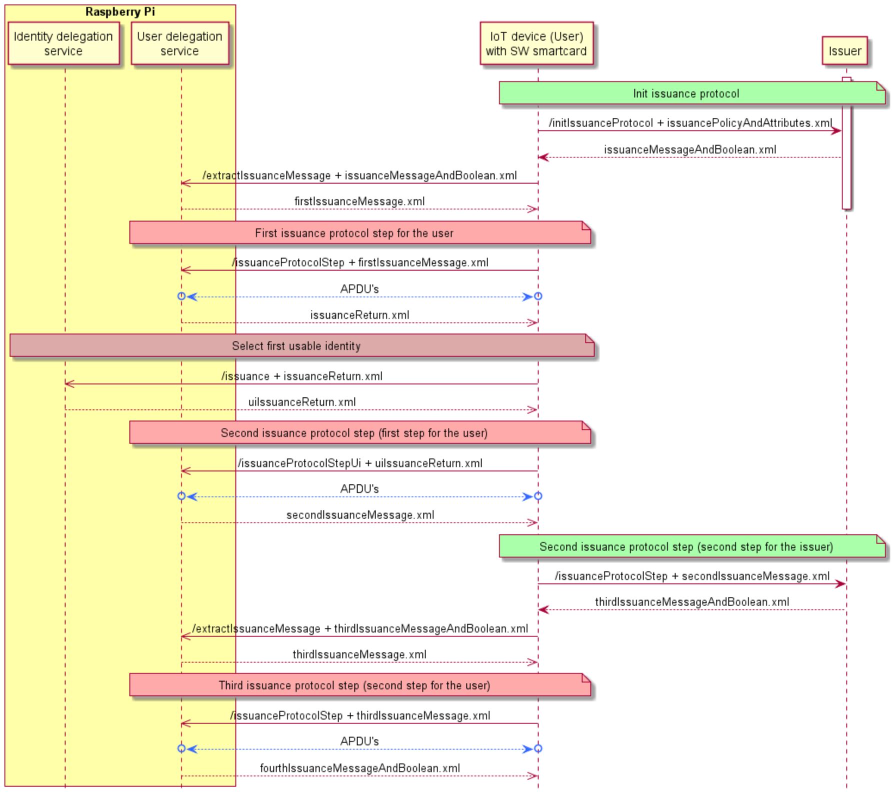


Figure 18: Issuance interaction.

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.

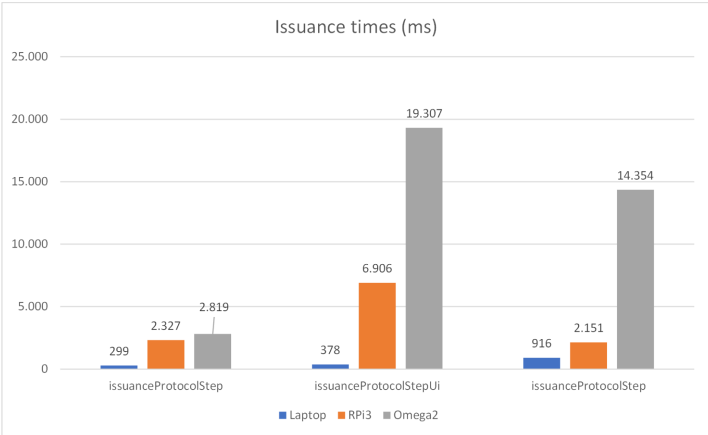
In Appendix B, Figure 24, we provide the APDU Commands for each step, 28 in total, with 1939 bytes, giving us about 27ms of delay in the network transmission.

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 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*.

	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)

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 serialization of the smart card for storage, and debug being human

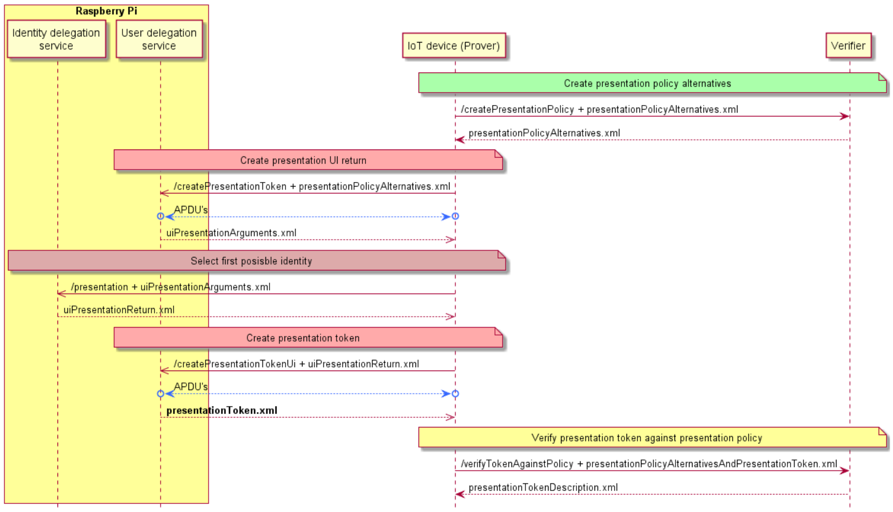


Figure 20: Proving interaction.

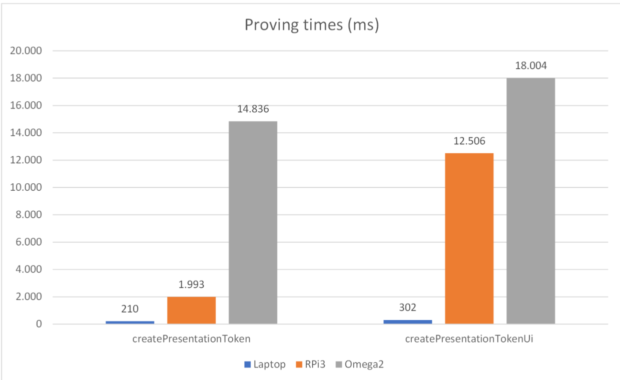
readable, stores a copy of every saved variable in the JSON tree structure, then creates a string (array of char) with the JSON, that the user can write to a file.

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.

	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)

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 that could start from the work done in this project.

6.1 CONCLUSIONS

The designed solution for the integration of Idemix and the Internet of Things provides new possibilities for the IoT security field, mainly in those scenarios where people's data is more vulnerable. However, there is a long path of research before we can see this in production. Many decisions depend on the specific deployment in course, and our solution tries to ease the best it all that future process.

With regards to the designed architecture, the flexibility of the Computation Offloading technique, identifying the key operations that can't be delegated, and those ones that can, allows 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 Idemix ecosystem. When in need of performing offloading, the delegation server also falls into the IoT class of devices, a great benefit for any real project, having as many options as possible, the solution delivered can be adapted to most of requirements. Nevertheless, being so open to any solution leaves us with a lot of work to do, researching what are the best options, comparing benefits and drawbacks.

Our PoC implementation demonstrates that this project is actually feasible, not by performing a simulation of an IoT device, like in [13]. During its 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.

Although there are many other techniques for the delegation server to send commands to the IoT device that knows the secret keys, like RPC

6.2 FUTURE WORK

APPENDIX

A

DOCKER AND CMAKE FILES

Listing 1: Dockerfile for P2ABCE

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_VERSION 1)

# specify the cross compiler
SET(CMAKE_C_COMPILER   /lede/staging_dir/toolchain-
    mipsel_24kc_gcc-5.4.0_musl/bin/mipsel-openwrt-linux-gcc)
SET(CMAKE_CXX_COMPILER /lede/staging_dir/toolchain-
    mipsel_24kc_gcc-5.4.0_musl/bin/mipsel-openwrt-linux-g++)

# where is the target environment
SET(CMAKE_FIND_ROOT_PATH  /lede/staging_dir/toolchain-
    mipsel_24kc_gcc-5.4.0_musl)

# search for programs in the build host directories
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
# for libraries and headers in the target directories
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

# Use the commands:
# mkdir build
# cd build
# cmake -DCMAKE_TOOLCHAIN_FILE=Toolchain-omega2-mipsel.cmake ..
# make
```

Listing 2: Dockerfile for P2ABCE

```
FROM openjdk:7

COPY idemix-3.0.36-binaries/ /idemix/

RUN apt-get update && apt-get install -y maven && rm -rf /var/lib
    /apt/lists/*

RUN     cd /idemix/com/ibm/zurich/idmx/com.ibm.zurich.idmx.3_x_x
    /3.0.36 && \
        mvn install:install-file     \
        -Dfile=com.ibm.zurich.idmx.3_x_x-3.0.36.jar   \
        -DpomFile=com.ibm.zurich.idmx.3_x_x-3.0.36.pom   && \

        cd /idemix/com/ibm/zurich/idmx/com.ibm.zurich.idmx.
            interfaces/3.0.36  && \
```

```
        mvn install:install-file      \
        -Dfile=com.ibm.zurich.idmx.interfaces-3.0.36.jar  \
        -DpomFile=com.ibm.zurich.idmx.interfaces-3.0.36.pom  && \

        cd /idemix/com/ibm/zurich/idmx/pom/3.0.36     && \
        mvn install:install-file          \
        -Dfile=pom-3.0.36.pom          \
        -DpomFile=pom-3.0.36.pom


# docker run -it --name p2abce_dev_env -v /p2abcengine/source/
    code/:/usr/src/mymaven -w /usr/src/mymaven/Code/core-abce
    p2abce_env bash
# docker start p2abce_dev_env
# docker attach p2abce_dev_env
```

Listing 3: Dockerfile for Omega2 SDK

```
# Linux dev environment for LEDE project
# Based on borromeotlhs' dockerfile

FROM ubuntu:14.04

LABEL maintainer="joseluis.canovas@outlook.com"
LABEL description="Dockerfile for Onion Omega2 SDK environment.
    It can be modified for Omega2+."

RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get
    install -y \
subversion g++ zlib1g-dev build-essential git python libncurses5-
    dev gawk gettext unzip file libssl-dev wget \
&& rm -rf /var/lib/apt/lists/*

# Install CMake 3.7.2
RUN mkdir ~/temp &&\
cd ~/temp &&\
wget https://cmake.org/files/v3.7/cmake-3.7.2.tar.gz  &&\
tar xzvf cmake-3.7.2.tar.gz &&\
cd cmake-3.7.2/ &&\
./bootstrap &&\
make  &&\
make install &&\
rm -rf ~/temp


RUN git clone https://github.com/lede-project/source.git lede

RUN adduser omega &&  echo 'omega:omega' | chpasswd   && chown -R
     omega:omega lede
WORKDIR lede
```

```
USER omega

RUN ./scripts/feeds update -a  && ./scripts/feeds install -a


# Set SDK environment for Omega2
# For Omega2+ change the third echo line with: (notice the 'p'
    for plus)
#  echo "CONFIG_TARGET_ramips_mt7688_DEVICE_omega2p=y" > .config
    && \

RUN echo "CONFIG_TARGET_ramips=y" > .config  && \
echo "CONFIG_TARGET_ramips_mt7688=y" >> .config  && \
echo "CONFIG_TARGET_ramips_mt7688_DEVICE_omega2=y" >> .config &&
    \
make defconfig

RUN make tools/install
RUN make toolchain/install

ENV PATH "$PATH:/lede/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0
    _musl/bin"
ENV STAGING_DIR "/lede/staging_dir"
ENV CFLAGS=-I/lede/staging_dir/target-mipsel_24kc_musl/usr/
    include
ENV LDFLAGS=-L/lede/staging_dir/target-mipsel_24kc_musl/usr/lib

# Use this command to run with shared directory:
# docker run -it --name omega2-sdk-app -v /my_host_dir:/remote
    omega2-sdk bash
```

B 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 second one is the issuance of the credential, divided in the three REST calls needed in the delegation. The last one is the proving for the Presentation Policy from the Verifier, using two REST calls.

Figure 22: Init IoT Smart Card APDU Commands exchanged.

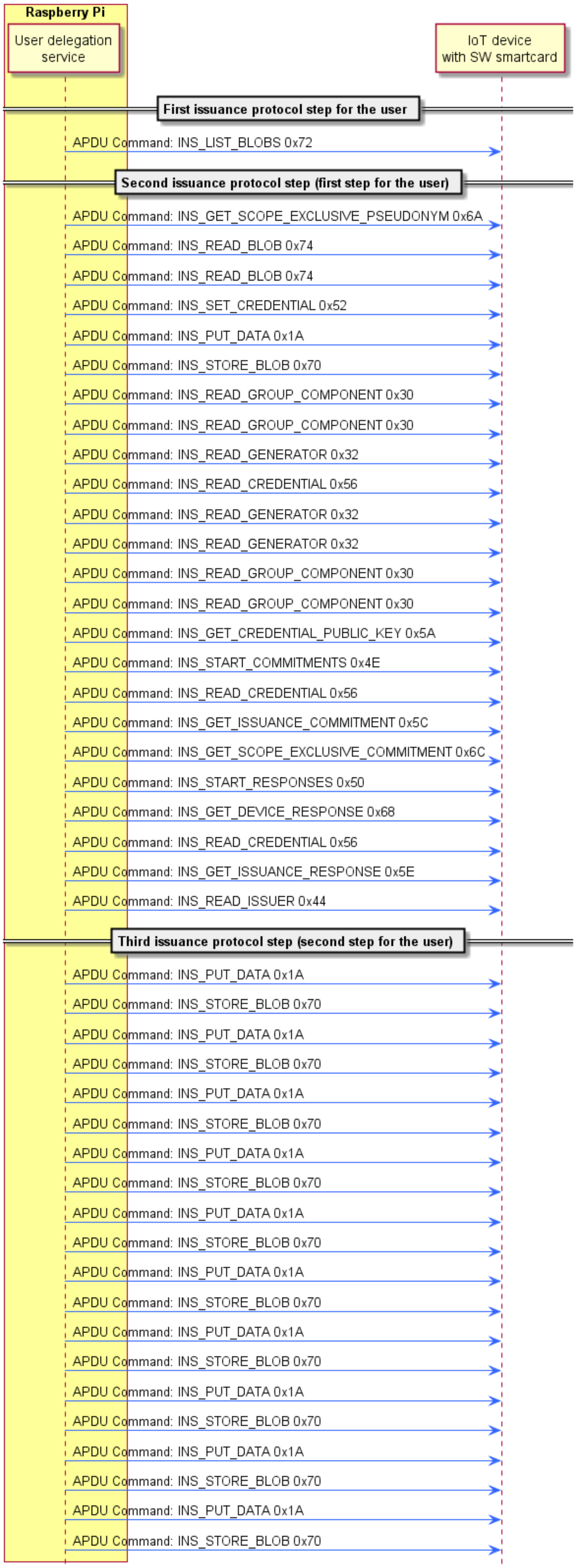


Figure 23: Issuance APDU Commands.

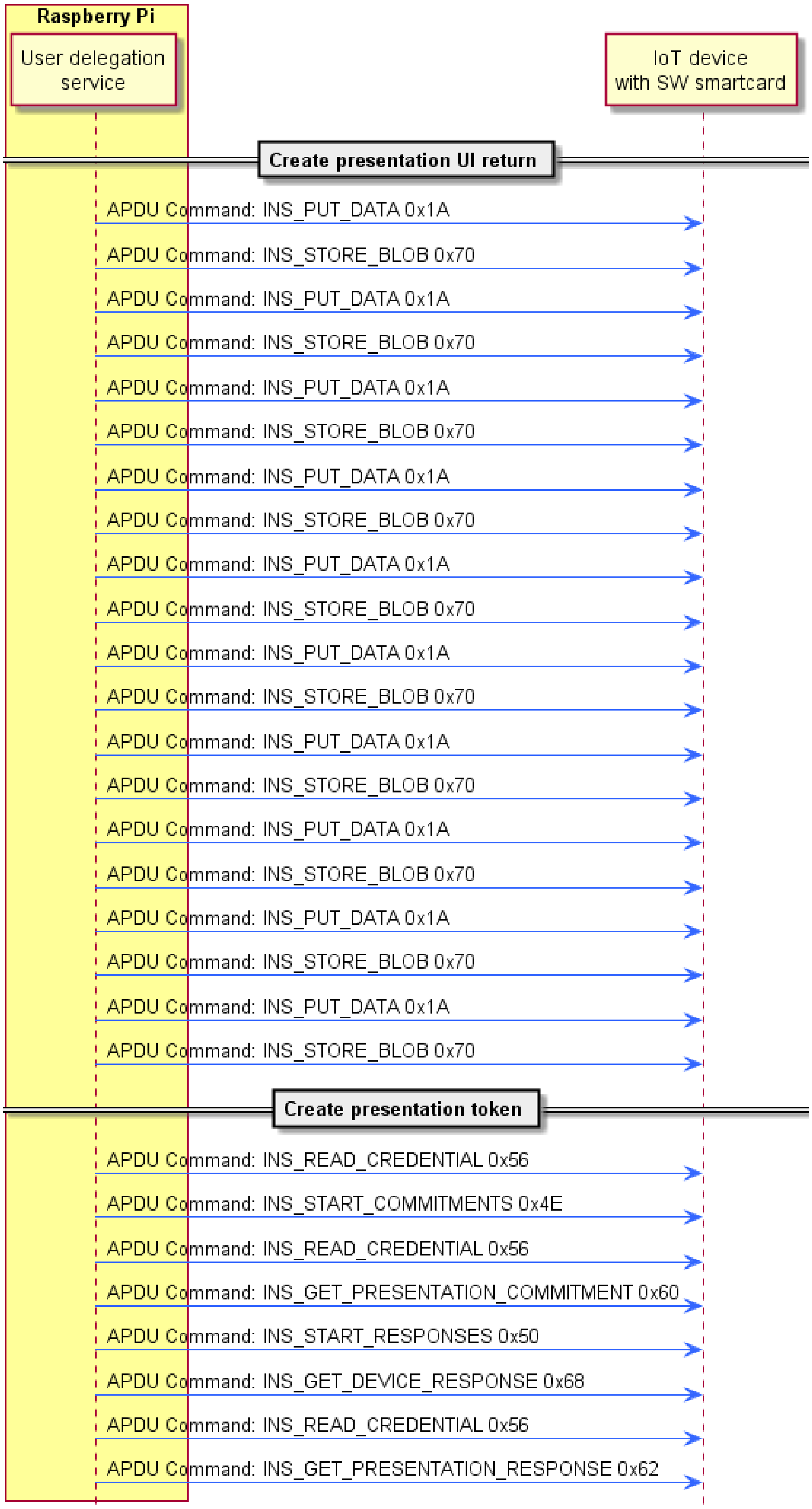


Figure 24: Proving APDU Commands.

C

GUIDE ON HOW TO CONTINUE THE PROJECT DEVELOPMENT

BIBLIOGRAPHY

[1] *2011 PlayStation Network outage*. https://en.wikipedia.org/wiki/2011_PlayStation_Network_outage.

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

[3] 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.

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

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

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

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

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

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

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

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

[12] *Data structure alignment*. https://en.wikipedia.org/wiki/Data_structure_alignment.

[13] 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.

[14] 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/.

[15] *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.

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

[17] 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. ISBN: 3540224203,9783540224204,978354027809

[18] 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.

[19] *Linux Embedded Development Environment*. https://lede-project.org/.

[20] *MULTOS technical library*. http://www.multos.com/developer_centre/technical_library/.

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

[22] *MediaTek MT7688 Datasheet*. https://goo.gl/kqD2gF.

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

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

[25] *P2ABCEngine Wiki*. https://github.com/p2abcengine/p2abcengine/wiki/.

[26] *P2ABCEngine*. https://github.com/p2abcengine/p2abcengine.

[27] 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.

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

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

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

[31] José Luis Cánovas Sánchez. "Pruebas de Conocimiento Cero y sus Aplicaciones."

[32] *THREADX OS Real-Time OS*. http://rtos.com/products/threadx/.

[33] 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