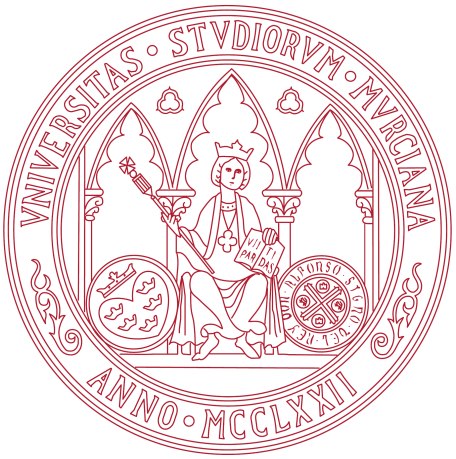
INTEGRACIÓN DE IDEMIX EN ENTORNOS DE IOT

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

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.

1939 – 2005

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

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [10]

ACKNOWLEDGMENTS

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, Henri Menke, Claus Lahiri, Clemens Niederberger, Stefano Bragaglia, Jörn Hees, and the whole LaTeX-community for support, ideas and some great software.

Regarding LyX: The LyX port was intially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and for the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

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

In recent years some new concepts have appeared in common people's vocabulary, like *machine learning*, *big data*, *artificial intelligence*, *automation*, etc., but there are two in particular that we are going to focus and try to combine: Internet of Things (IoT) and Internet Security & Privacy.

The IoT is a term with a wide range of interpretations [3], but a breaf definition could be the set of devices, mainly resource constrained, that are interconnected between them in order to achieve a goal. This includes from lampposts with proximity sensors that talk to each other in order to light up part of the street when a passerby walks by, to a sensor on your clothes that tells the washing machine how much detergent to use.

Security & Privacy, thanks to organizations like WikiLeaks, are now taken in consideration by any technology consumer, not only professionals. People are conscious about what their data can be used for, demanding more control over it.

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 [14], or like the multiple vulnerabilities affecting baby monitors [18].

A recent approach to address the problem of privacy is the *strong anonymity*, that conceals our personal details while letting us continue to operate online as a clearly defined individual [7]. One very promising way to achieve this is using Zero-Knowledge Proofs (ZKPs) , cryptographic methods that allows to proof knowledge of data without disclosing it. Furthermore, IBM has been developing a cryptographic protocol suite for privacy-preserving authentication and transfer of certified attributes based on ZKP, called Identity Mixer, Idemix for short [9].

To read more about ZKP aside the introduction done in this thesis, you can read my Mathematics thesis [19].

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 common architecture, policy language and data artifacts, but based on either IBM's Idemix or Microsoft's U-Prove [17]. This gives us a standardized language to exchange Idemix's messages between IoT devices and usual PCs.

2

STATE OF THE ART

In this chapter we present the two dimensions of this project: the IoT development state and an introduction to IBM's privacy-preserving solution, Idemix.

2.1 INTERNET OF THINGS

The development for Internet of Things depends heavily on each target device. We can differentiate two big groups: those with enough processing power to act like an usual computer, and those constrained devices that can't perform arbitrary tasks, sometimes called *embedded*.

We can consider in the first category powerful ARM devices like Raspberry Pi 3, with a 64-bit architecture, 4 CPU cores, 1GB of RAM, which can even compile its own binaries, run the *Java Virtual Machine*, etc., working in practice like any other computer. These kind of devices do not present any major difficulty in terms of research.

What we will consider to be a more *pure* IoT device will be the constrained ones, where it's not trivial to develop any algorithm and run it successfully.

Very known devices fall into this category, like Arduino, powered by Atmel's AVR ATmega328 8-bit microprocessors, with 32KB of program flash memory, 2KB of SRAM, 16MHz of CPU [2]. It seems clear that memory and computation power are a very big issue to deal with when developing to this devices.

A step above in power we can find ESP8266, the most famous Espressif's microcontroller, with built-in WiFi antenna, a Tensilica 16bit RISC microcontroller at 80MHz, 50Kb of RAM, and 1MB flash memory [6]. The possibility of direct WiFi connectivity is its best selling point, putting the *Internet* in Internet of Things.

In another level of power we have microcontrollers usually found in routers, but used in many other applications, like the On-Board Units (OBU) used in Vehicular ad hoc networks (VANET). Characteristics in this range vary around a single core 32-bit CPU, at some hundreds MHz, with tens or hundreds MB of RAM and flash memory, which places them near the first IoT mentioned category.

Although one can code in assembly language for these microcontrollers, there exist C compilers, and many frameworks to build firmware binaries: Arduino Core, Contiki, propietary SDKs, Mongoose OS, ThreadX OS (Real-Time OS), OpenWrt, LEDE, etc. Each firmware targets specific ranges of devices, depending on processing power and memory limitations. For example, Arduino and Contiki aim for microcon-

trollers like Atmel's ATmega and TI's MSP430, but can also be used in ESP8266, a more powerful microcontroller.

In particular OpenWrt and LEDE (a fork of OpenWrt) are based on Linux, with optimized library binaries, providing many packages through *opkg* [15]. To compile C/C++ code, build the firmware or packages, a complete build system and cross-compiler toolchain can be installed in a x86 host, and using Makefiles select the target hardware [16].

Devices running OpenWRT and other Linux distros are in the limit between IoT categories, but the need of a cross-compiler marks that they belong to the second category.

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

For this reason the current Proof of Concept (PoC) is developed on LEDE, Linux Embedded Development Environment [11], using the Onion Omega2 development board, a Mediatek MT688 microcontroller [13] with a 32-bit MIPS 24KEc CPU at 580MHz, 64MB of RAM and 16MB of flash and built-in WiFi. This development board uses LEDE as its firmware, but its CPU is also listed as compatible with ThreadX OS [20], a Real-Time Operative System for embedded devices.

The PoC will take advantage of the Linux system using 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.

2.2 IDEMIX

TODO: MOVE TO MOTIVATION The problem of Internet privacy has been approached by securing the transmission channel (e.g. SSL/TLS) and the data stored in both ends (strict access policies, local encryption, etc.). In the end, the data exists in two entities, the owner of the data and the service provider. The owner is the most interested in securing his data, and can apply as many measures as he wants, but only on his side of the table. The service provider that stores the user data needs it to provide the service, and a successful attack would reveal many users data, aside from how many measures each one used to protect it. The case of PlayStation Network outage in 2011 [1] affected 77 million accounts, with suspected credit card fraud, is an example of this kind of attacks.

Other solutions are based on minimal disclosure. Standards like OAuth offer secure delegated access to the user information and

when registering to a new service, the user can give a key to access only the data they want from another trusted service. This lets the service provider to work with the OAuth server, offering the same service as before without knowing as much data.

But this is only minimizes how many services have our data. Our OAuth provider could be attacked, revealing all our data, or our service provider, revealing now less data.

IBM proposes a step forward using Zero-Knowledge Proofs:

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

TODO: PONER DESCRIPCIÓN + FORMAL DE IDEMIX, LUEGO P2ABCE, Y METER QUE EN P2ABCE HABÍA UNA IMPLEMENTACIÓN DE SC BASADA EN C

3 OBJECTIVES AND METHODOLOGY

3.1 PROJECT DESCRIPTION

3.2 WORKING METHODOLOGY

3.3 DEVELOPMENT ENVIRONMENT

3.3.1 *Hardware*

To test our design in a realistic but easy to work with deployment, we used as the IoT device an Onion Omega2 development board, and as the delegation server a Raspberry Pi 3.

Onion Omega2

ONION OMEGA2: A device that falls inside the second category of IoT, powerful enough to run a familiar Linux environment, where we can develop and debug the first PoCs without troubling ourselves with non-related to the project problems.

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

- A reliable 3.3V with a maximum of 800mA power supply (a USB with a step-down circuit works fine), with quality soldering and wires to avoid unwanted resistances.
- A Serial to USB adapter wired to the TX and RX UART pins to use the Serial Terminal, in case WiFi doesn't work and no SSH is available, and because the connection is more reliable in case of wireless interferences.

MCU	Mediatek MT688 [13]
CPU	MIPS32 24KEc 580MHz
RAM	64MB
Storage	16MB
Firmware	LEDE (OpenWRT fork distro)
Connectivity	Wifi b/g/n
Power	3.3V 300mA

Table 1: Onion Omega2 Specifications.

Raspberry Pi 3

RASPBERRY PI 3: Another familiar environment, powerful enough to debug and hold the delegated P2ABCE Java services (User, Verifier, ...) of P2ABC with its 1GB of RAM, and with two network interfaces it's perfect to work as the gateway for the IoT devices to the Internet.

Only a microSD with enough space to burn the binary with 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 *.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 2: Raspberry Pi 3 Specifications.

3.3.2 *Software*

The development is divided between the IoT device code and the P2ABCE services.

The P2ABCE is already written in Java, and few modifications will be done to the code in comparison to the existing project size, so we will continue using Java with the P2ABCE part.

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 some can't compile C++, some may not have many common libraries, and that the memory limitations they face make practically impossible to use dynamic memory, if we want to avoid very possible execution malfunctions.

For that reason, the developed code for IoT devices must be written with standard C without using dynamic memory.

A project with thousands of lines of code can't be written in a single file. And to manage the compilation of multiple files, organized in various directories, we will use CMake.

CMake has 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 pure C without external libraries or dynamic memory, the PoC uses three major libraries:

- OpenSSL: Provides reliable and tested AES and SHA256 implementations.
- LibGMP: Provides multiprecision integer modular arithmetic.
- cJSON: Provides a JSON parser to store and read the status 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 environments:

P2ABCE ENVIRONMENT A container with OpenJDK 7 and Maven installed, with the Idemix maven plugins installed following the project instructions to use Idemix as the Engine for P2ABCE.

LEDE SDK ENVIRONMENT A container with CMake and the LEDE SDK [11] installed and configured for the Omega2 target.

The Dockerfiles can be found in the Appendix.

4

DRAFT

4.1 SMART CARD APDU

To communicate the smart cards and the reader an standardized protocol is specified in ISO/IEC 7816-4 [8].

The messages, also kown as Application Protocol Data Unit (APDU), 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 1: 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.

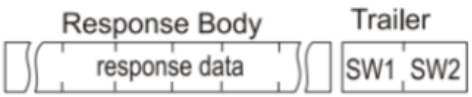


Figure 2: APDU Response

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 dynamic memory in RAM between APDU Commands, what allows to do in two or more commmands complex operations, transmit more bytes than a single APDU can admit, etc.

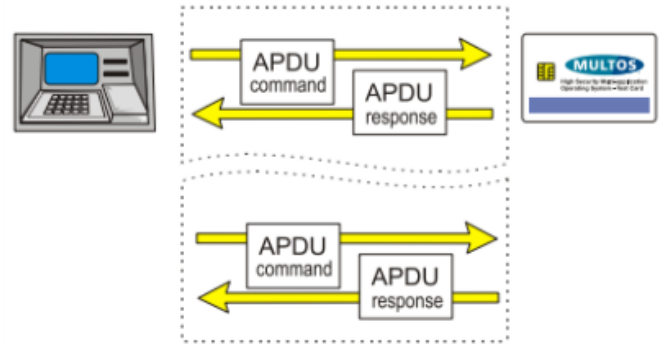


Figure 3: APDU Command-Response Dialogue

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, a new APDU Command instruction was defined so the smart card stored the payload in a buffer, until other APDU Command made use of it. To send more data in an APDU Response, the status bytes were set to: SW1=0x61 and SW2 to the remaining bytes to send. Because a smart card can't send APDU Commands, the card terminal then sends a GET RESPONSE, a special APDU Command, with Le set to the number of bytes specified in SW2. Iterating this process, the smart card could send as many bytes as it wanted.

4.2 P2ABCE

In the P2ABCE repository [17] is available the project's code, divided in two solutions: a complete P2ABCE implementation in Java and a Multos Smartcard implementation as companion for the project.

The Java code is managed by a Maven project, structured using various known design patterns, but not of our interest. The structure we are actually interested in are the REST Services and their use of the Components classes, in which the smartcards are included.

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

The *SoftwareSmartcard* class implements the interface in Java, suitable for tests and self-stored smartcards that any application using P2ABCE may need.

The *HardwareSmartcard* class uses the standard APDU messages [TODO:ref] to interact with smartcards. P2ABCE defines for every method in the mentioned interface, the necessary APDU instructions, and currently relies on *javax.smartcardio* abstract classes (implemented by Oracle in their JRE) to communicate with the smartcard reader. This way, it doesn't matter what manufacturer issues the smartcard, or if it's an Android device, if they support the APDU API, P2ABCE will work with them.

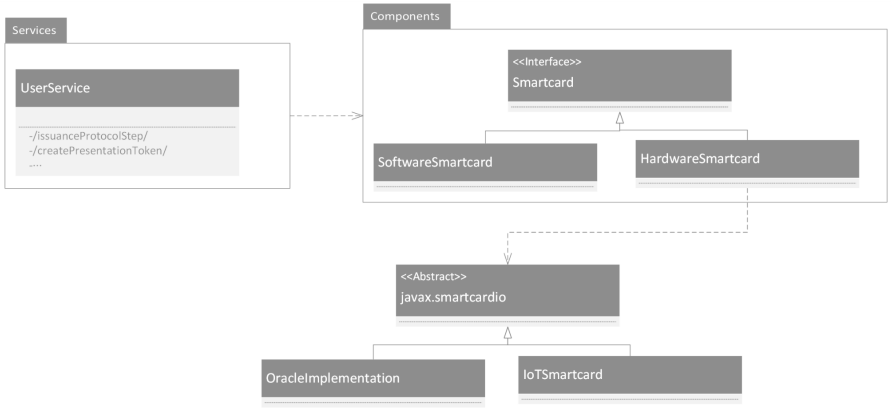


Figure 4: Basic P2ABCE structure

As a PoC the P2ABCE project includes the ABC4Trust Card Lite, an implementation for ML3-36K-R1 Multos Smartcards. The code is written in C, but is very dependent on the Multos framework, aside from numerous bugs and bad coding habits.

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, the already existing *HardwareSmartcard* class can work with the new *IoTSmartcard* in the IoT device.

4.3 MULTOS

MULTOS is a multi-application smart card operative system, which provides a custom developing environment, with rich documentation [12]. 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.

MULTOS WORKFLOW Most of 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 `main()` function will start.

The application uses then the *multos.h* file that declares multiple global variables already loaded with the needed data, including the APDU Command bytes.

Now the developer is in charge of checking what instruction was sent and if the APDU has the expected ISO Case. If everything is ok, code what needs to be run and write in specific data space the APDU Response bytes, call *multosExit()* and MULTOS will be in charge to send the APDU Response.

In summary, our application starts with all data loaded and exits without worrying how to send the answer. A very comfortable workflow that we must now implement for our IoT device if we would want ABC4T Card Lite code to work.

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

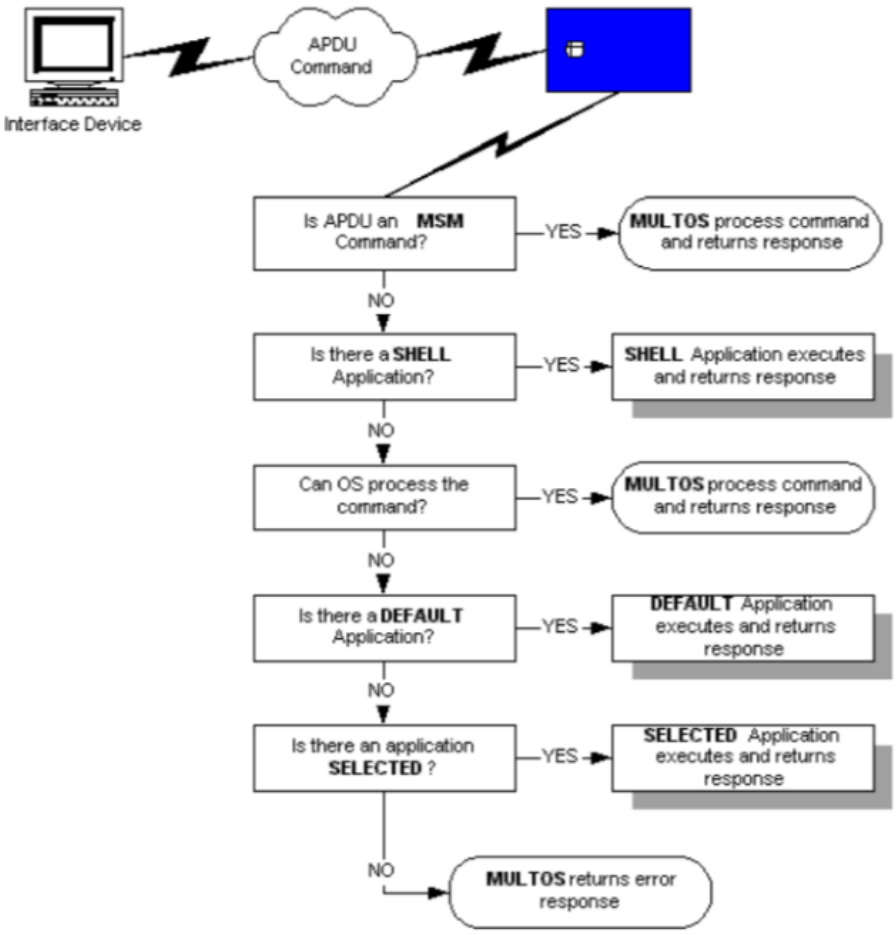


Figure 5: MULTOS workflow

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 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 7). 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*.

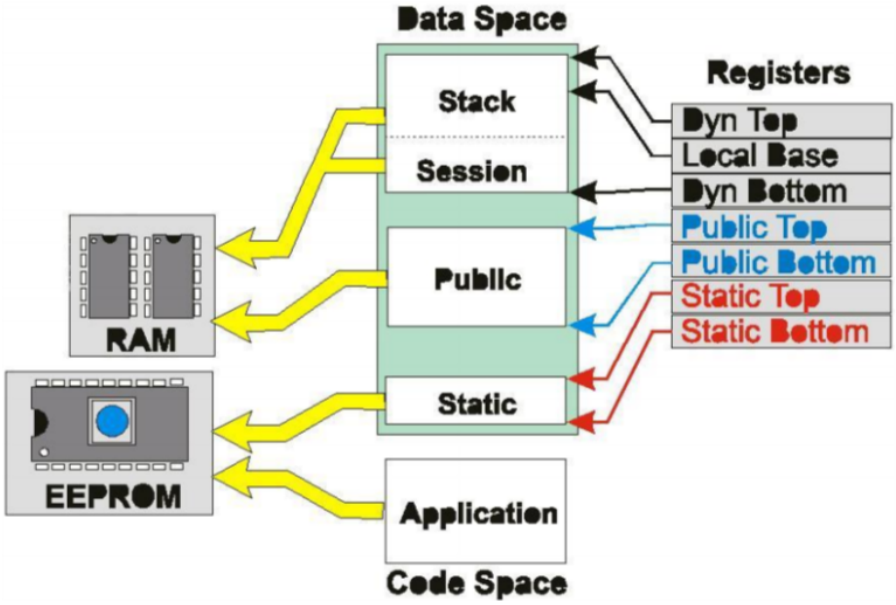
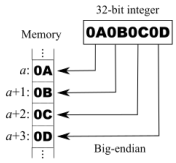


Figure 6: MULTOS Memory Layout



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 8. 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 [5] for performance.

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

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 7: MULTOS Public Memory Data Map

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 8: MULTOS Data Types

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 [12] provides rich description for each function.

4.4 ABC4TRUST CARD LITE

P2ABCE provides a smart card reference implementation, ABC4Trust Card Lite [4]. It supports device-bound U-Prove and Idemix, and virtually any discrete logarithm based pABC system.

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

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

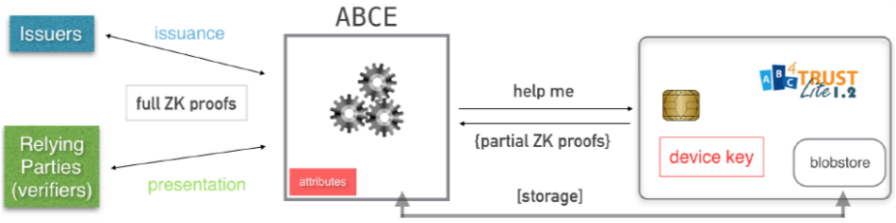


Figure 9: ABC4Trust Card Lite

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

The code is available from the P2ABCE project and has some good and bad points to have in count:

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

Some *tricks* in the code include using *union* data types for variables that will be stored on the same data location, but at different moments (e.g. depending on APDU Command INS byte), minimizing this way the use of RAM and making code readability better; or strong use of pointers and *memcpy* calls to copy structures with multiple variables as arrays of bytes.

Among the many drawbacks, we could highlight the awful coding, the strong dependency on MULTOS framework and some bugs found.

The code is structured in two files, *main.h* and *main.c*, with 557 and 5157 lines of code respectively.

The file *main.h* is mostly a reimplementation in assembly MEL of some MULTOS functionality already offered with latest *multos.h*.

The *main.c* consists on near 600 lines of variables and data structures declarations, followed by the *main()* function, a 2635 lines long *switch-case* with practically no comments, and to conclude, the implementation of thirty functions called *Subroutines* at the end of the file.

This gives an idea of the problematic to maintain or even understand the code. But once one studies MULTOS framework in deep and applies many refactoring techniques to ABC4T Card's code, this becomes the best starting point for the IoT version.

4.5 IOT AND P2ABCE

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

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

The communication is already solved by the *Internet* capabilities of IoT devices.

Our concern are the data artifacts exchanged as XML and the cryptographic operations involving secret keys that must remain private to the IoT device.

Here is where we look at the P2ABCE architecture more closely, and the concept of smart cards shows a solution for the second issue. Even in the case we were to implement all P2ABCE inside an

IoT device, we would have to implement support for software smart cards, to keep the secret inside the IoT device. We will start building the house from the ground, implementing the smart card operations inside the IoT device.

Now that in our design we have the smart card, we need to address the first point, XML squemas. We understand with *XML squemas* both managing the XML syntax and the whole process to generate a proper answer, that is, basically, the crypto engine that relies on Idemix and the smart card to hold the secret information.

Taking in consideration Idemix is currently provided in Java as a considerably big project, and version 3.0.36 still hasn't got official documentation, the task to port the Idemix crypto engine to IoT could be done in the future, if the chosen devices have enough storage and capacity. And even in that case, we would need to implement P2ABCE User and Verifier's architecture to completely free ourselves from an external P2ABCE machine.

The final port would be so big, many IoT devices would fall out of the requisites to run it, failing in our initial objective.

After this analysis of the P2ABCE, we conclude that the mandatory requisite for any IoT device that wants to work with this system and keep its private keys in it, is to implement the smart card functionality, and delegate the rest of the operations on a machine capable of running P2ABCE, until that functionality is implemented for the IoT device.

This architecture is not really such an original idea. For example, IPv6 involves managing 128 bits per address and large headers, and many use cases only need IoT devices to communicate inside a private network. That's why many of them use 6LoWPAN to compress packets or use smaller address sizes. To communicate a 6LoWPAN with the Internet, a proxy is needed to transform 6LoWPAN packets to IPv6.

Therefore, the IoT device now has a **duality** in its functions, because it is the User that starts any interaction with other systems, and it's also the smart card that P2ABCE delegates for crypto 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.

We find here two challenges: how to delegate from the IoT device to the P2ABCE delegation server, and how to transmit them and the APDUs to the IoT device.

Currently P2ABCE offers various 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 rewritten 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 case, capabilities and resources available. If the delegation server is connected, for example, through RS-232 serial with the IoT device, and physically inaccessible, in the same way an IoT device on its own would be protected, 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 will 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 delega-

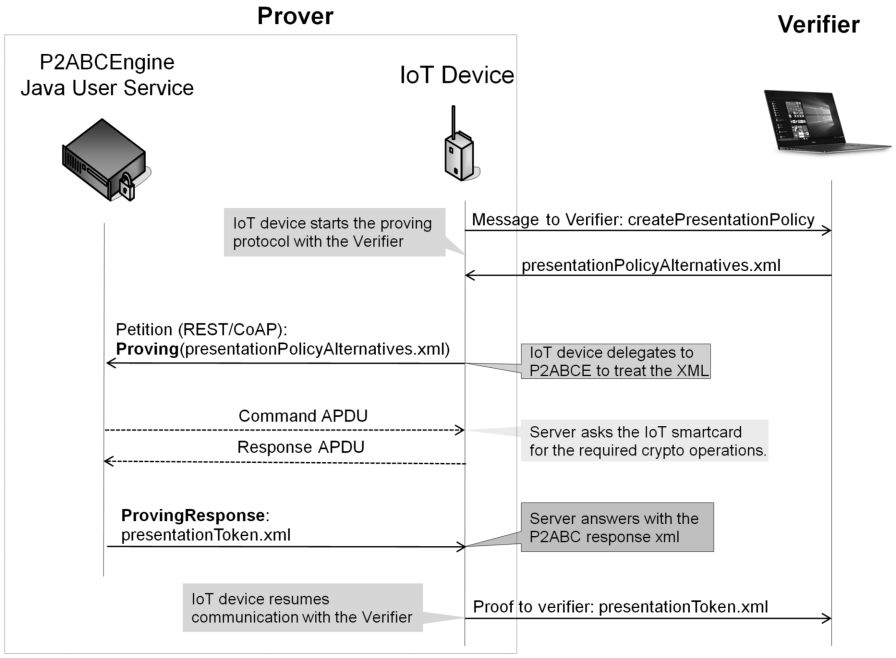


Figure 10: IoT Delegation in P2ABCE for Proving.

tion server, but as we said, managing complex XML squemas is not something many IoT devices could do.

4.6 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.

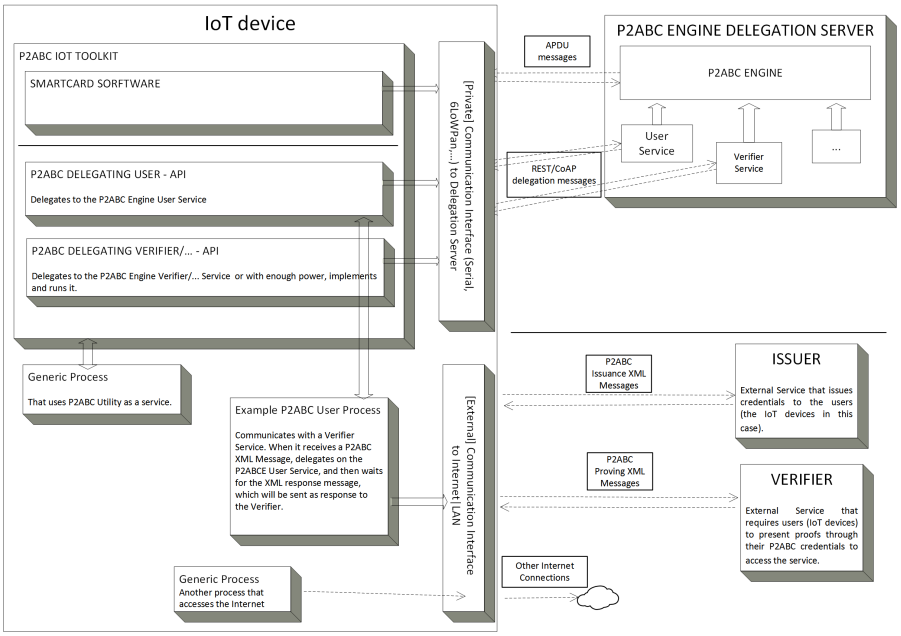


Figure 11: IoT in P2ABCE deployment diagram.

4.6.1 *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.

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 equiv-

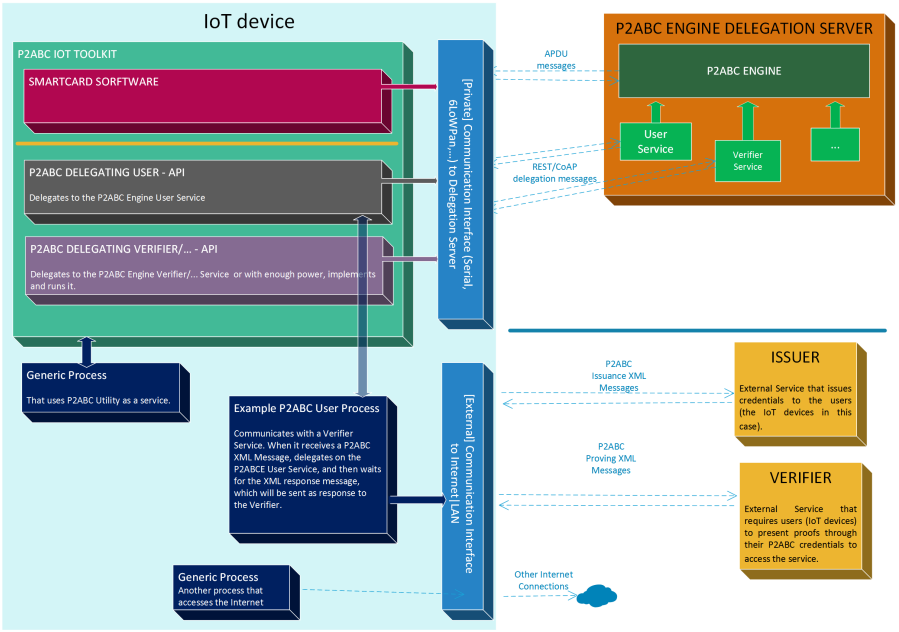


Figure 12: IoT in P2ABCE deployment diagram.

alent 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.

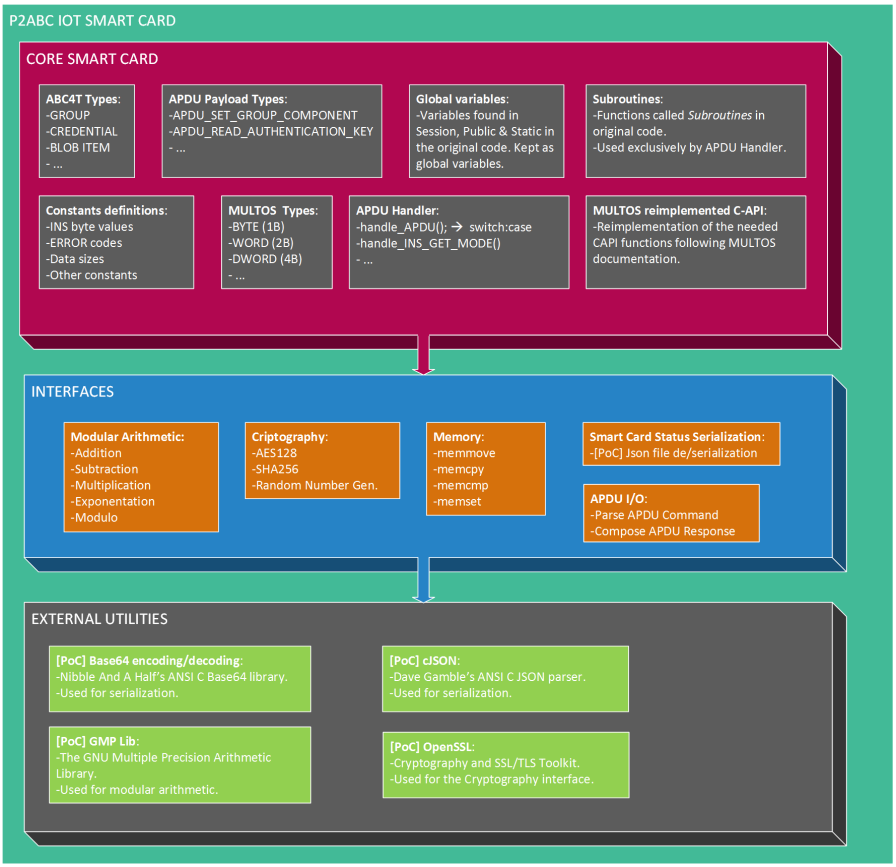
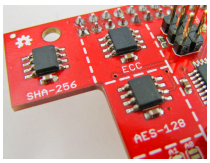


Figure 13: IoT Smart Card Code Structure.

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 [**ATAES132A**] 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.

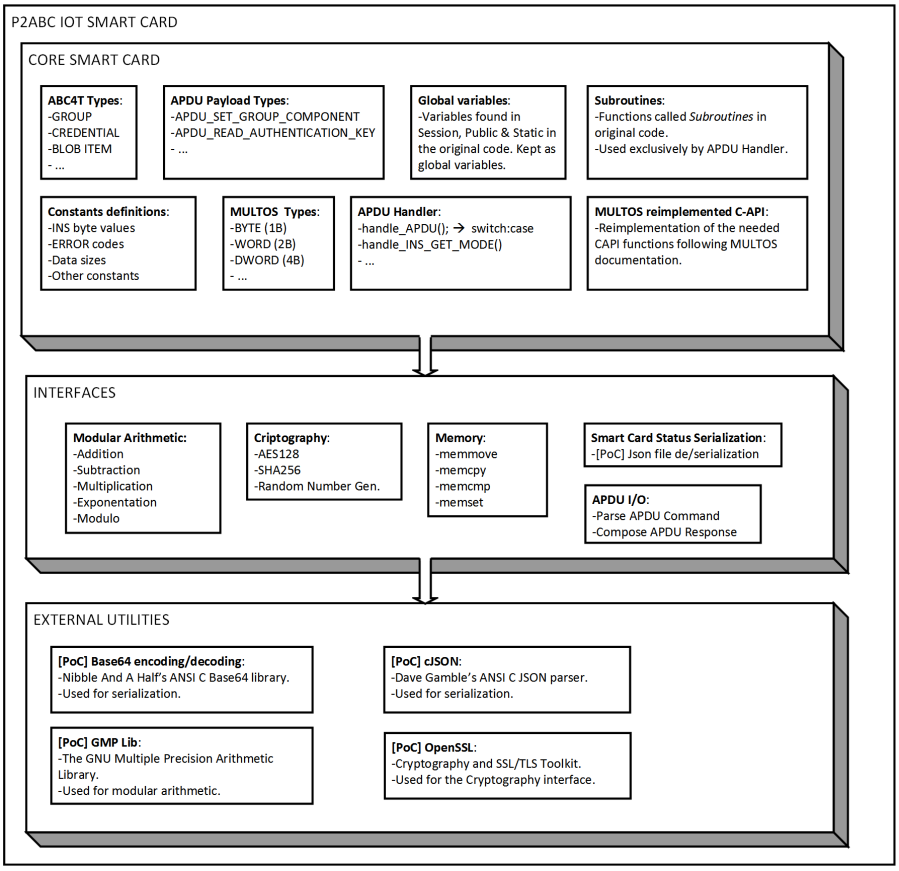


Figure 14: IoT Smart Card Code Structure.

4.6.2 *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.

4.7 PERFORMANCE EVALUATION

4.7.1 *Testbed description*

4.7.1.1 *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 [**p2abcurlwiki**]. It is based on a soccer club which wishes to issue VIP-tickets to a soccer 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.

4.7.1.2 *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 3: Raspberry Pi 3 Specifications.

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.

Onion Omega2

MCU	Mediatek MT688 [13]
CPU	MIPS32 24KEc 580MHz
RAM	64MB
Storage	16MB
Firmware	LEDE (OpenWRT fork distro)
Connectivity	Wifi b/g/n
Power	3.3V 300mA

Table 4: Onion Omega2 Specifications.

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:

- 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 IN TESTS The lack of a physical MULTOS smart card precludes us to load and test the ABC4Trust code and measure the time P2ABCE needed using the *HardwareSmartcard* class. This would be really interesting because for a single method from the *Smartcard* interface, *HardwareSmartcard* needs to send various APDUs, but *SoftwareSmartcard* can perform the operations immediately.

4.7.2 *The test code*

There are three pieces of software that conform the test: the P2ABCE services, the IoT smart card, and a shell script automatizing the REST calls to all the services.

P2ABCE SERVICES This is the 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 Jetty web server for the PC and Raspberry Pi.

We modified the User Service code to measure the execution time for each REST method. The time stamp starts inside the method and ends before it's *return* instruction. This way, we filter the time the 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.

4.7.3 *Results*

After 15 executions for each scenario (laptop, RPi3, Omega2), we take the means and compare each step.

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

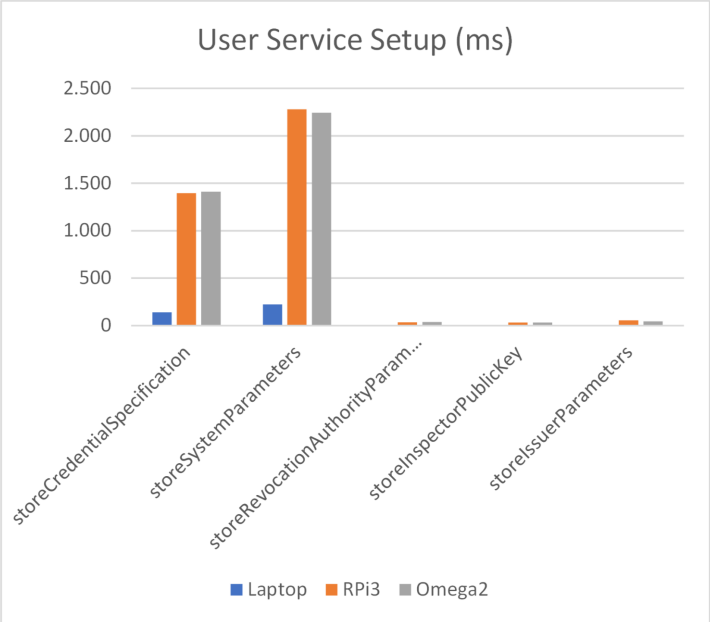
THE NETWORK To test the network we sent six thousand APDUs to the Omega2, but instead of calling the *APDU handler*, the Omega2 responded with the same bytes back. This way, the Omega2 only performed the simple BIOSC protocol, reading and writing the TCP socket. The APDUs had multiple sizes, taken from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

THE SETUP Because the Omega2 doesn't intervene until the creation of the smart card, the times measured in the second and third scenarios are practically identical.

As we can see, 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.

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

(a) Times and relative speedup



(b) Comparison graph

Figure 15: Setup times (milliseconds)

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`.

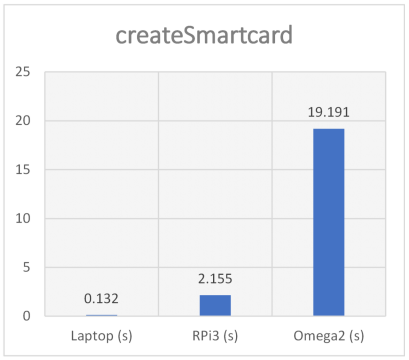
From Figure 16 we see that the RPi3 is about 16 times slower than the laptop in the creation of the *SoftwareSmartcard*, but almost 9 times faster than the setup of the smart card in the Omega2 using APDUs. This gives us that the laptop is 145 times faster than the combination of RPi3 and Omega2 in our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

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 Omega2 IoT smart card, **using 30 APDU Commands, and Responses**, as shown in Figure 17, with a total of 1109 bytes exchanged. From our network benchmark, using TCP sockets, the delay in the transmission is only

	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 16: Create smart card times (seconds)

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.

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 Figure 19 the APDU Commands for each REST call. There are 45 APDU Commands in total, 3197 bytes exchanged, that would introduce a latency of 45ms in the network, negligible.

In Figure 20 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 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 21, 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 Figure 22 the APDU Commands for each step are shown, 28 in total, 1939 bytes, 27ms approx. in the network.

Again, as shown in Figure 23, 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*.

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 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.

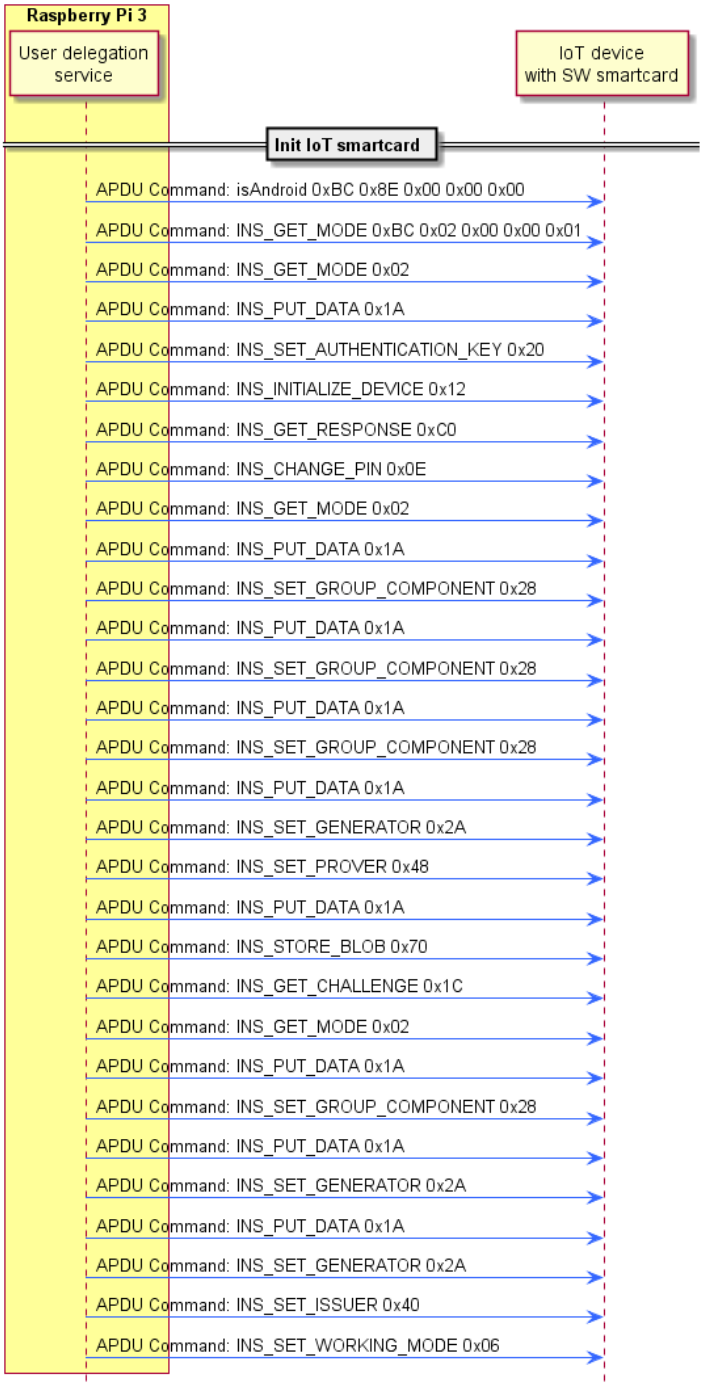


Figure 17: Init IoT Smart Card APDU Commands exchanged.

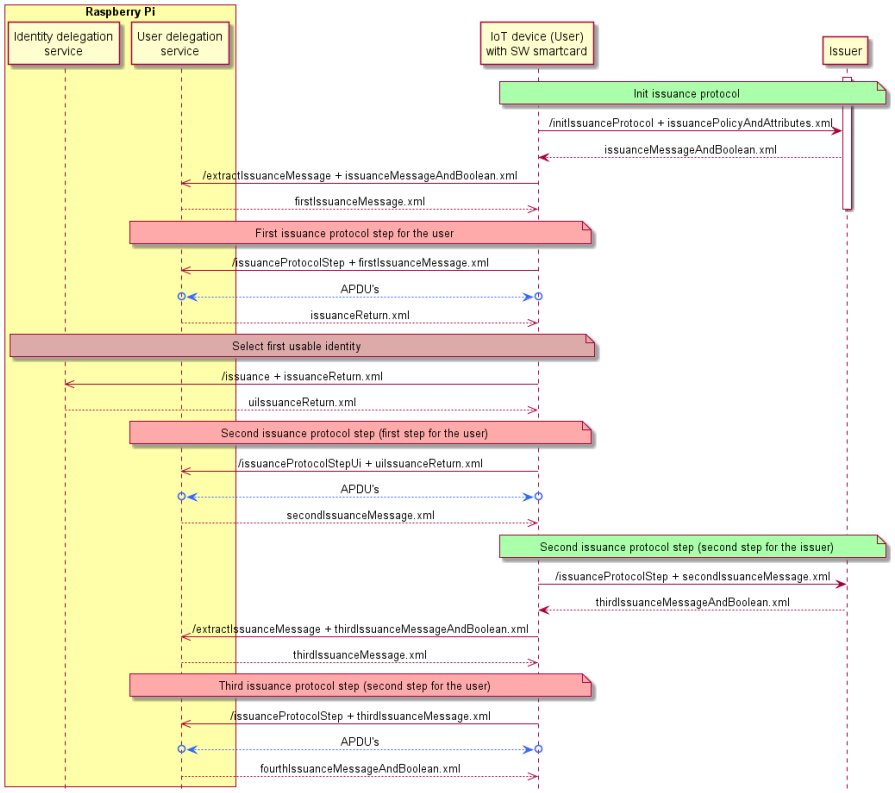


Figure 18: Issuance interaction.

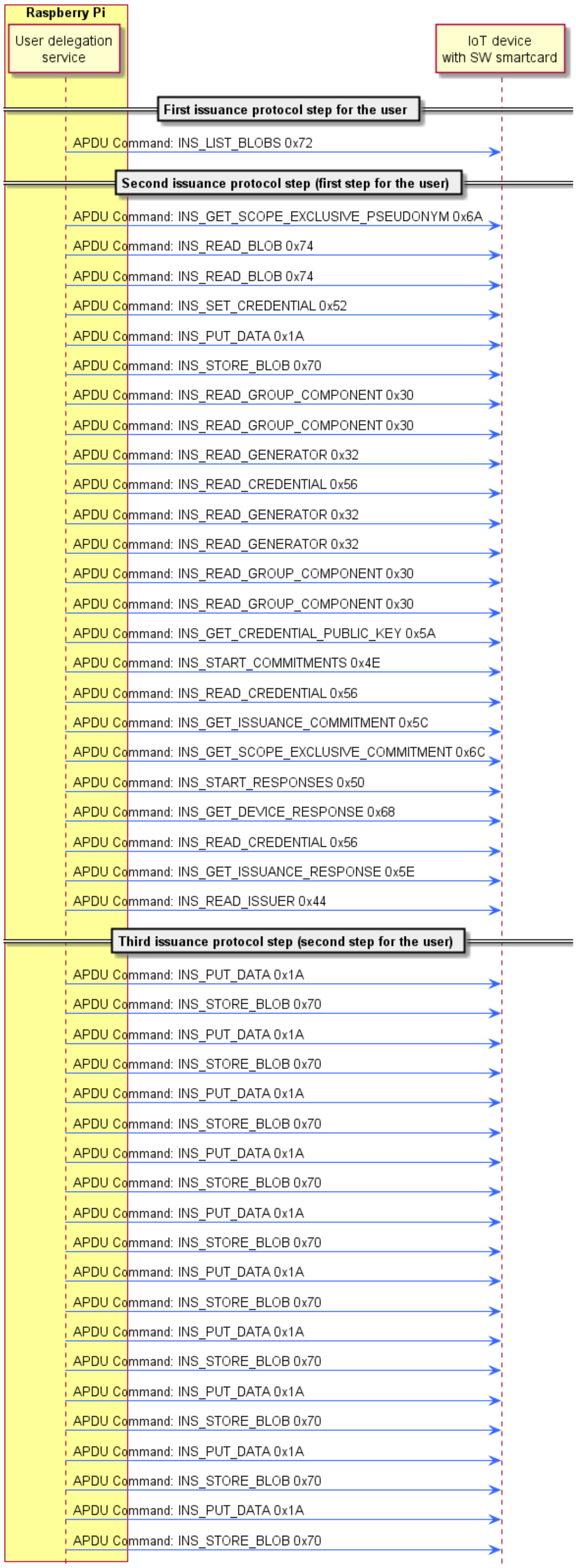
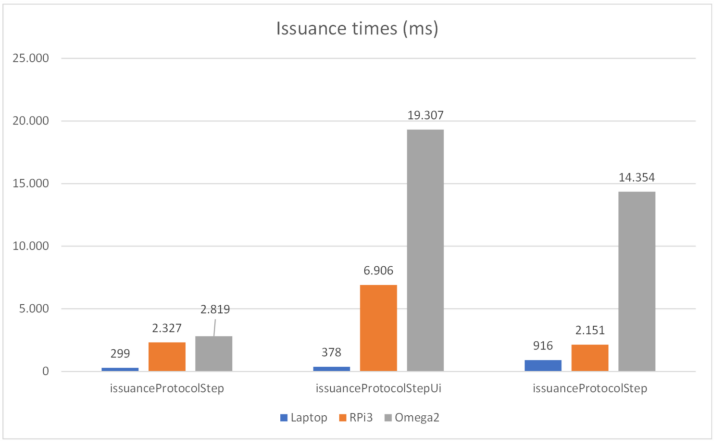


Figure 19: Issuance APDU Commands.

	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 20: Issuance times (milliseconds)

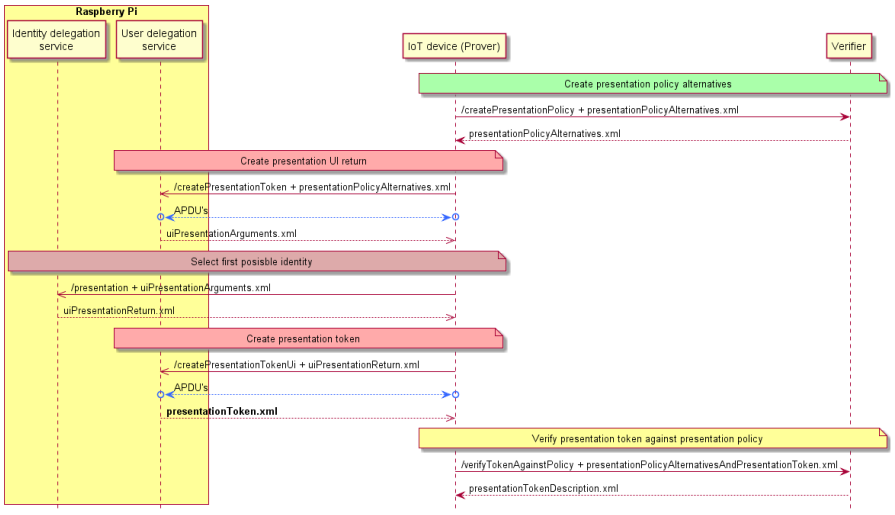


Figure 21: Proving interaction.

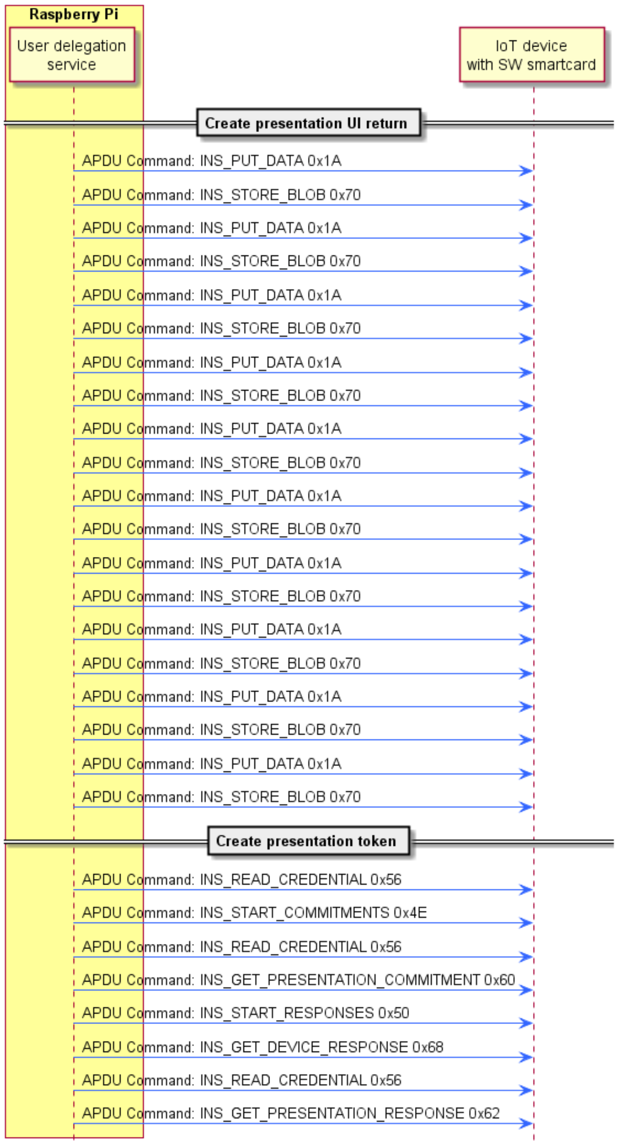
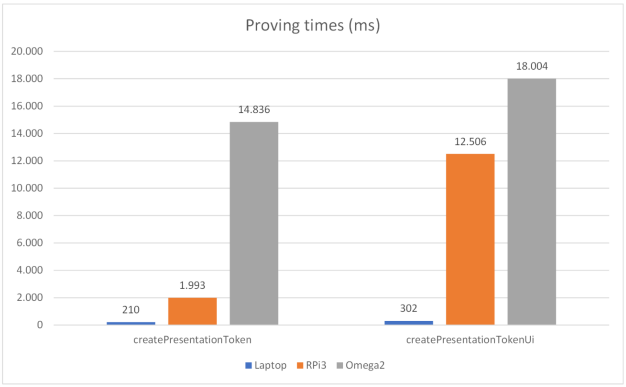


Figure 22: Proving APDU Commands.

	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 23: Proving times (milliseconds)

APPENDIX

APPENDIX TEST

Lorem ipsum at nusquam appellantur his, ut eos erant homero concludaturque. Albucius appellantur deterruisset id eam, vivendum partiendo dissentiet ei ius. Vis melius facilisis ea, sea id convenire referrentur, takimata adolescens ex duo. Ei harum argumentum per. Eam vidit exerci appetere ad, ut vel zzril intellegam interpretaris.

More dummy text.

A.1 APPENDIX SECTION TEST

Test: Table 5 (This reference should have a lowercase, small caps A if the option `floatperchapter` is activated, just as in the table itself → however, this does not work at the moment.)

LABITUR BONORUM PRI NO	QUE VISTA	HUMAN
fastidii ea ius	germano	demonstratea
suscipit instructior	titulo	personas
quaestio philosophia	facto	demonstrated

Table 5: Autem usu id.

A.2 ANOTHER APPENDIX SECTION TEST

Equidem detraxit cu nam, vix eu delenit periculis. Eos ut vero constituto, no vidit propriae complectitur sea. Diceret nonummy in has, no qui eligendi recteque consetetur. Mel eu dictas suscipiantur, et sed placerat oporteat. At ipsum electram mei, ad aeque atomorum mea. There is also a useless Pascal listing below: Listing 1.

Listing 1: A floating example (`listings` manual)

```
for i:=maxint downto 0 do
begin
{ do nothing }
end;
```

BIBLIOGRAPHY

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

[2] *ATmega328 Datasheet*. http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf.

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

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

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

[6] *Espressif ESP8266 Datasheet*. https://espressif.com/en/products/hardware/esp8266ex/resources.

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

[8] *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.

[9] *Identity Mixer*. https://www.research.ibm.com/labs/zurich/idemix/.

[10] Donald E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

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

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

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

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

[15] *OPKG Package Manager*. https://wiki.openwrt.org/doc/techref/opkg.

[16] *OpenWrt's build system*. https://wiki.openwrt.org/about/toolchain.

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

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

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

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

DECLARATION

Put your declaration here.

, Junio 2017

José Luis Cánovas Sánchez

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of May 16, 2017 (`classicthesis`).