Integration of Anonymous Credential Systems in IoT constrained environments

Jose Luis Canovas Sanchez, Jorge Bernal Bernabe, Antonio F. Skarmeta
Department of Information and Communications Engineering
Computer Science Faculty, University of Murcia, Spain
{joseluiscanovas, jorgebernal, skarmeta}@um.es

***Abstract*—The pervasive nature of Internet of Things entails additional threats that compromise the security and privacy of IoT devices and, eventually, the users. This issue is aggravated in constrained IoT devices equipped with minimal hardware resources. Current security and privacy implementations need to be re-designed and implemented maintaining its LoA, aiming for this family of devices. To cope with this issue, this paper proposes the first novel attempt to leverage Anonymous Credential Systems (ACS) to preserve the privacy of autonomous IoT constrained devices. Concretely, we have designed a solution to integrate IBMs Identity Mixer into constrained IoT ecosystems, endowing the IoT with ACSs privacy-preserving capabilities. The solution has been designed, implemented and evaluated proving its feasibility.**

***Index Terms*—Anonymous credential system, Internet of Things, Identity Mixer, Zero-knowledge proof**

I. Introduction

Internet of Things is a term with a wide range of interpretations [3], breafly, we can think of it as billions of devices, mainly resource constrained, which are interconnected between them and the Internet, in order to achieve a goal.

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 is also the data collected about us and which we do not have direct control over it.

In traditional M2M (Machine to Machine) environments the issues about security and privacy have already been treated deeply, but in the IoT ecosystem, due to it's recent and fast growth, it lacks of those tools to autonomously solve these issues.

To address these problems 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. Anonymous credentials and selective disclosure techniques allow us to control what information we reveal to others.

The most common type of ACS are ABCs (Attribute-Based Credentials), which can be thought of as a digital signature by an Issuer on a list of attribute-value pairs [17]. The most straightforward way for a 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 any information about the credential other than the chosen properties to display. 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 classical symmetric and asymmetric cryptography it seems rather impossible to create such credentials without an explosion of signatures over every possible combination of attributes. For this reason, current solutions rely on Zero-Knowledge Proofs (ZKP), cryptographic methods that allow to proof knowledge of some data without disclosing it.

Based on ZKPs, IBM developed Identity Mixer[1], Idemix for short, a 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 based on their values. Thus, no personal data is collected that needs to be protected, managed, and treated by third parties.

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

The design and implementation presented in this paper will allow constrained IoT devices to carry out all actions available in the Idemix protocol, being in control of the decisions to take in every step. This achieves oblivious interactions between any traditional machine using Idemix and the IoT devices, without having to adapt the protocol, or a subset it, to interact with these new entities.

[1] Identity Mixer - https://www.zurich.ibm.com/identity_mixer/

This document is structured as follows: Section II provides a state of the art analysis through the history of Idemix and related works, analysing what is of the most interest for the IoT perspective. Section III presents a formal design of the proposed IoT and Idemix solution. Section IV describes the PoC implementation developed. After any implementation, it is a must to validate it, as showed during the performance tests in section V. Finally, our conclusions and lines for future work are described in section VI.

II. State of the art

In this section we present a showcase of competing ACS solutions, focusing their application for the IoT, where the two most notable alternatives to Idemix are Persiano's ABC systems and Microsoft's U-Prove.

In 2004, 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 [18]. 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 k times (where k is the security parameter) in order to obtain a satisfying soundness.

Based on Persiano's proofs, an anonymous authentication for privacy-preserving IoT was presented in [2], but the studied performance analysis was carried out using Java on a traditional desktop PC, theoretically assuming IoT devices to be proportionally 40 times slower than the testing machine.

In 2000, Stefan Brands provided the first integral description of the U-Prove technology in his thesis [7], 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 [6] 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 tokens public-key, which aggregates all information in the token, and a signature from the issuer over this public-key to ensure the authenticity [19].

Independently of the mentioned technologies, Jan Camenisch, Markus Stadler and Anna Lysyanskaya studied in [8], [9], [11] the cryptographic bases for signature schemes and anonymous credentials that later became IBM's Identity Mixer protocol specification [1].

Luuk Danes in 2007 studied theoretically how Idemix's User role could be implemented using smart cards [12], 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 a combination of possibilities where the smart card would give all the information to the terminal, only partial information, or keep everything secret and perform all the private operations within itself.

Later, in 2008 Víctor Sucasas also studied an anonymous credential system with smart card support [15], 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 [5] and [20], but they weren't optimal and didn't include some Idemix's functionalities, like selective disclosure.

In 2013, Vullers and Alpar, implemented an efficient smart card for Idemix [21], aiming to integrate it in the IRMA[2] 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.

Later, the P2ABCE[3] project extended the concept of smart cards, physical or logical, as holders of the credentials. The P2ABCE project is a language framework that unifies different cryptographic ABC engines and policies, currently supporting U-Prove and Idemix. The Idemix library was updated to support P2ABCE and their last version is therefore interoperable with U-Prove. The smart card specification from the P2ABCE project can be considered the official version to work with. The Identity Mixer team instructs the use of P2ABCE in order to use Idemix itself.

Related to the IoT, the P2ABCE project has been used to test in a VANET[4] scenario how an OBU (On Board Unit) with constrained hardware could act as a User in a P2ABCE ecosystem [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.

Another recent approach to integrate Idemix in the IoT was performed in [4], where it can be seen as a first attempt of a real re-implementation, not simulation, but aimed for Android devices, written in Java, and therefore, not suitable for constrained devices either.

III. Design of the proposed solution

This section describes the design of our proposal to integrate IoT constrained devices as part of the privacy preserving ecosystem P2ABCE. The ultimate goal is to enable constrained IoT devices to play the Idemix User and Verifier roles, interacting autonomously in order to authenticate and demonstrate their credential attributes to any Verifier in a

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

[3] https://github.com/p2abcengine/p2abcengine

[4] Vehicular Ad-Hoc Network

privacy-preserving fashion, as well as verifying other devices in a M2M environment, addressing the power and memory constrains many IoT devices face.

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.

We also analysed how the P2ABCE architecture uses the logic of smart cards to gather the cryptographic operations independently from how the data is exchanged between P2ABCE actors.

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.

Using the *computation offloading* technique to our scenario, our design consists on implementing the smart card logic inside the IoT device, keeping safe 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 a new technique for IoT environments. For example, to reduce the overhead of IPv6 we use 6LoWPAN to compress packets and use smaller address sizes. In order to communicate a 6LoWPAN with other networks, the IoT devices delegate on a proxy that can manage the 6LoWPAN and IPv6 stacks. In the scope of consumer devices, smart watches can install applications which delegate on the user's phone to accomplish performance demanding 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 is 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 that P2ABCE server delegates on the IoT device, acting now as the smart card.

A. Solution requirements

When designing a privacy preserving solution for IoT, we must aim for a series of objectives that guarantee its integrity and security. Our requirements are:

- *Security*: the solution should not compromise the security of the device's credentials.
- *Usability*: it should provide an API for developers to use the P2ABCE privacy capabilities in their applications.
- *Full P2ABCE support*: the IoT device should be capable of performing every action a traditional User or Verifier could perform.
- *Transparent interactions*: any third party actor of the system should not be aware of the IoT condition of the device, therefore, there should be no changes to the P2ABCE protocol.
- *Constrained devices*: the solution should aim to be portable and applicable to as many IoT devices as possible, including those with less computing capabilities.

B. Fundamentals on Zero-Knowledge Proofs

To understand Zero-Knowledge Proofs, *Fundamentals of Computer Security* [16, Chapter 12] offers a good introduction to the topic. Here is shown how one can indeed proof knowledge of a value, without revealing it, using the classic ZKP of knowledge based on the discrete logarithm problem, also known as Schnorr's identification scheme. Based on this discrete logarithm ZKP, Idemix then derives multiple ZKPs of relations and properties of the values hidden in a credential.

Using the notation introduced by Camenisch and Stadler [10], the discrete logarithm ZKP can be written as

$$PK\{(\alpha) : y = g^{\alpha}\}$$

given a known group $G = \langle g \rangle$ of prime order q and public value $y \in G$. The notation means: "I know a secret value α such that g^{α} is y", i.e. the discrete logarithm of y, $log_g y = \alpha$.

During the first step of the protocol, the prover chooses randomly a value r_{α}, computes the *commitment* $t := g^{t_{\alpha}}$ and sends t to the verifier. Then, the verifier chooses another random exponent, the *challenge* c, and sends it to the prover. Next, the prover computes the *response* $s := r - c\alpha \, mod \, q$ and sends it to the verifier. Finally, the verifier checks whether or not $t \overset{?}{=} g^{s_{\alpha}} y^{c}$ holds. The verifier never receives the secret value α, and the verifier will not be able to compute it given t and s.

The protocol holds because $g^{s_{\alpha}} y^{c} = g^{r_{\alpha} - c\alpha} g^{\alpha c} = g^{r_{\alpha}} = t$.

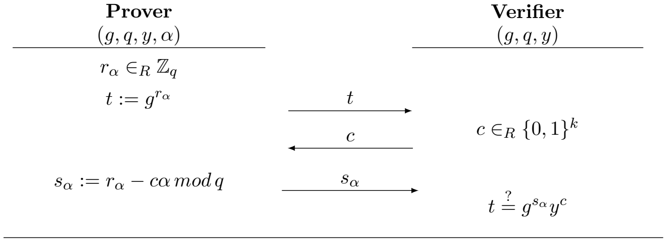

Prover
(g, q, y, α)
Verifier
(g, q, y)
$r_{\alpha} \in_R \mathbb{Z}_q$
$t := g^{r_{\alpha}}$
t
c
$c \in_R \{0,1\}^k$
$s_{\alpha} := r_{\alpha} - c\alpha \, mod \, q$
s_{α}
$t \overset{?}{=} g^{s_{\alpha}} y^{c}$

Fig. 1. Schnorr's protocol $PK\{(\alpha) : y = g^{\alpha}\}$. The prover knows (g, q, y, α) such that $g^{\alpha} = y$. The verifier knows (g, q, y).

The Fiat-Shamir heuristic lets us replace the verifier challenge with a hash function $\mathcal{H}$, computing the challenge as $c := \mathcal{H}(g \mid y \mid t \mid n)$. The value n is a random nonce generated by the verifier at the beginning of the non-interactive ZKP. The nonce makes the verifier trust that the current ZKP is fresh and not a forgery from a previous ZKP.

In Idemix, every ZKP follows the scheme shown above. First, we have the *commitment* t, next the *challenge* c (computed with $\mathcal{H}$), and finally, the *response* s. A prover can achieve parallel ZKPs if she first computes the multiple t-values of each individual proof, next, she computes the same challenge c for all the proofs by combining all the information in one hash call, and then she computes all the s-values in parallel.

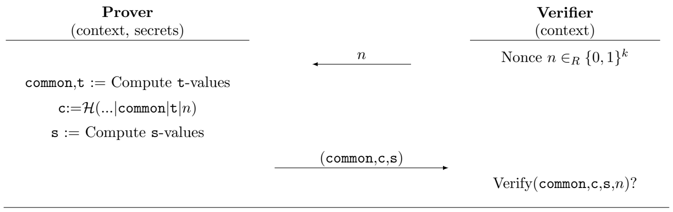


Fig. 2. Identity Mixer's Non-Interactive ZKP with Nonce.

Fig. 2 shows the Idemix Proving scheme, where prover and verifier share a context information, the verifier generates the nonce, and the prover, non-interactively, computes the proof. The common values are equivalent to Schnorr's t, which was shared between prover and verifier. The verifier can recover some $\hat{t}$-values from common, c and the s-values. This is equivalent to recovering Schnorr's t from $g^{s_\alpha}y^{c}$. The verifier then checks whether or not c equals $\mathcal{H}$(...|common|$\hat{t}$|n) to verify if the $\hat{t}$-values are actually the t-values, making the proof valid.

Therefore, to describe any Idemix proof being performed by the User, one can focus on the key steps, compute the t-values, the challenge c and the s-values, and ignore the mathematical operations performed for each specific ZKP.

C. System architecture

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

- **IoT device**
  Figure 3 shows our proposed architecture, in which the IoT device is represented with two communication interfaces. One allows external communications to other machines, including other P2ABCE actors. Through this interface, the P2ABCE XML messages are exchanged as in any traditional P2ABCE scenario. This lets an IoT device to interact with other actors without adaptations of the protocol. The other interface allows a secure communication channel with the delegation server. Both the delegation messages and the APDU Dialogue are transmitted over this interface, making it a point of attack that must be thoroughly secured.
  The scheme also shows the *P2ABCE IoT Toolkit*. This piece of software includes the *IoT Smart Card*, and the P2ABCE API.
  The *IoT Smart Card* is the implementation of a software smart card, which 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 to the server. In the future we could implement, for example, the Verification Service to run entirely in the IoT device, then the toolkit would conceal the transition from delegation to native execution.
- **P2ABCE actors**
  The possible roles in a P2ABC system are the Issuer, the User, the Verifier, the Revocation Authority and the Inspector, where these last two actors are optional. All of them use the P2ABCE language to communicate to each other. Any third party actor that communicates with the IoT device will be unaware of the fact that the device is a constrained device, because it will accept and generate the same XML as a traditional User.
  Figure 4 showcases the different actors and their interactions, where the User is in the center, which receives a credential from an Issuer, can generate privacy-preserving Tokens for Verifiers or revoke an stolen credential. The Inspector is the only entity capable of reading some ciphered attributes from a Token, if the User accepted to include that ciphered information in it. It serves the Law Enforcement authorities, warrant granted, to track any misuse of a credential.
- **P2ABCE Delegation Server**
  The machine in charge of receiving commands from authorized IoT devices to parse the XML files exchanged. It will also orchestrate through APDU Commands the cryptographic operations the IoT smart card must perform.

D. Delegation process

In this section we describe the computation offloading carried out by the IoT device. The steps the device will perform in any kind of interaction are:

- Communication with P2ABCE actor
  The IoT device, acting as a User, starts an interaction with another actor. If it is an Issuer, it will start the issuance process. If it is a Verifier, then it will provide a Presentation Policy for the device to proof in a privacy preserving way. In Figure 5 we show an example of this last case.
  It may also happen that the device is contacted as a Verifier by another actor, e.g. in a M2M scenario where the IoT device requires authentication to access to its resources.
- Delegation to the P2ABCE Server
  Depending on what role the IoT device is acting as, it will use the corresponding API from the P2ABCE IoT Toolkit. The selected API will delegate to the Service deployed in the delegation server, e.g. User Service. The delegation message will include the XML data, and any parameter required to accomplish the task, as the information on how to communicate back with the IoT smart card.

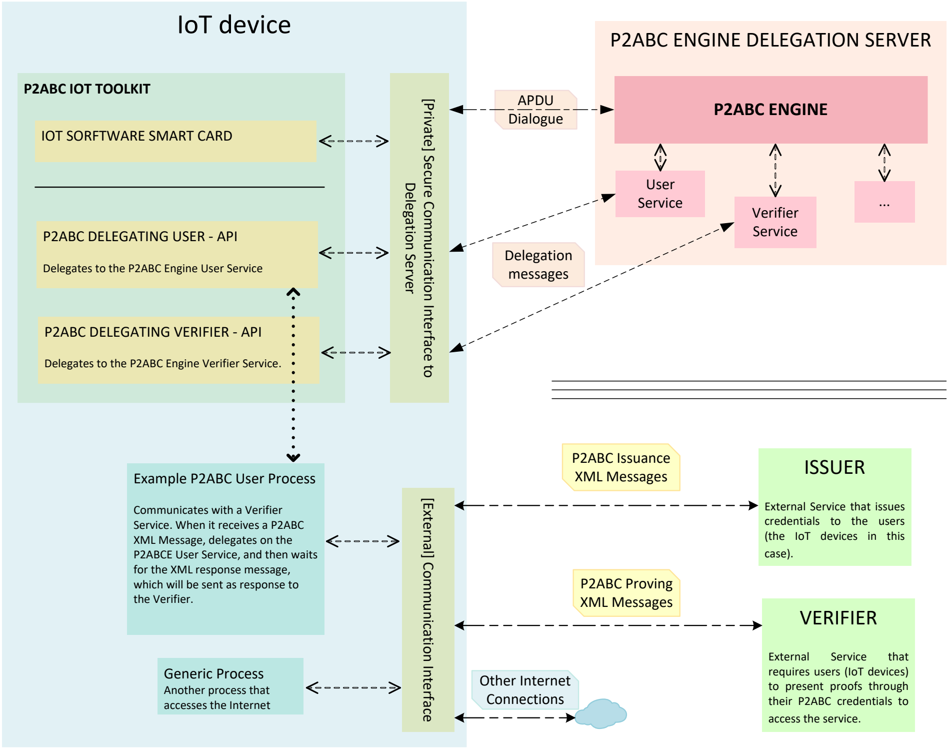

IoT device
P2ABC IOT TOOLKIT
IOT SORFTWARE SMART CARD
P2ABC DELEGATING USER - API
Delegates to the P2ABC Engine User Service
P2ABC DELEGATING VERIFIER - API
Delegates to the P2ABC Engine Verifier Service.
[Private] Secure Communication Interface to Delegation Server
APDU Dialogue
Delegation messages
P2ABC ENGINE DELEGATION SERVER
P2ABC ENGINE
User Service
Verifier Service
...
Example P2ABC User Process
Communicates with a Verifier Service. When it receives a P2ABC XML Message, delegates on the P2ABCE User Service, and then waits for the XML response message, which will be sent as response to the Verifier.
Generic Process
Another process that accesses the Internet
[External] Communication Interface
P2ABC Issuance XML Messages
ISSUER
External Service that issues credentials to the users (the IoT devices in this case).
P2ABC Proving XML Messages
VERIFIER
External Service that requires users (IoT devices) to present proofs through their P2ABC credentials to access the service.
Other Internet Connections

Fig. 3. Proposed high level Architecture for integrating IoT devices in P2ABCE.

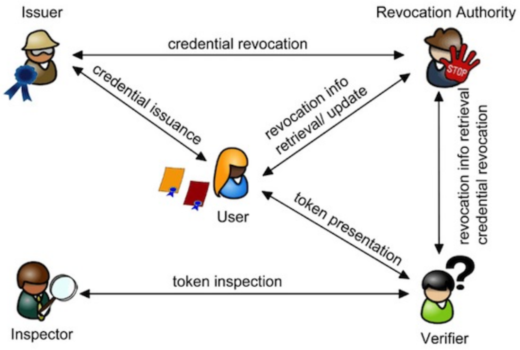


Fig. 4. Entities in a P2ABC System. Source: P2ABCE project.

The server will then parse the XML messages and begin to orchestrate the response. In the case of the Verifier Service, it will answer immediately with an *accept* or *reject* of the Token. In case it is the User Service, it will need information from the IoT credentials and cryptographic operations.

- APDU Dialogue
Through the secure channel between IoT device and server, the User Service will send APDU Commands to the *IoT smart card* to read the credential information or perform cryptographic operations involving private keys, necessarily stored inside the IoT device.
During a proof, the Service will read the credential public information to fill the Presentation Token, and then will request the IoT smart card to calculate the `t`-values of the proof. The Service will read the `common` and `t`-values, use the nonce present in the Presentation Policy of the Verifier and compute the challenge `c`. Next, the Service will send `c` to the IoT smart card through another APDU Command to request the calculation of the `s`-values.
After the APDU Dialogue, the Service has all the needed values to fill in the Token, and the IoT device performed all the operations involving private values on its own.
- Server response
After the APDU Dialogue, if needed, the server may return a status code indicating success or failure, or a XML response if the third party actor requires an answer from the IoT device, as a Presentation Token, or the intermediate issuance messages.

Transmissions over the *Server-IoT* channel must be secured in order to avoid attacks like impersonating the P2ABCE delegation server, an attacker sending APDU Commands to the IoT smart card, or delegating as a device to the server but giving the parameters of another device, making the delegation server send the APDU Commands to a victim IoT smart card.

We could use a corporative PKI to issue certificates to the server and devices and configure policies for access control; design a challenge-response system combined with the smart card PIN, like a password and TOTP[5] in a 2FA[6] 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.

[5] Time-based One-time Password
[6] Two-factor authentication

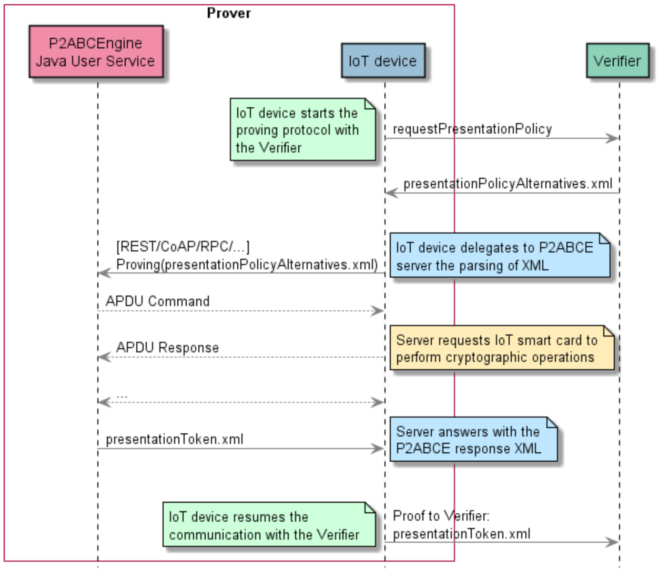

Prover
P2ABCEngine Java User Service
IoT device
Verifier
IoT device starts the proving protocol with the Verifier
requestPresentationPolicy
presentationPolicyAlternatives.xml
[REST/CoAP/RPC/...] Proving(presentationPolicyAlternatives.xml)
IoT device delegates to P2ABCE server the parsing of XML
APDU Command
Server requests IoT smart card to perform cryptographic operations
APDU Response
...
presentationToken.xml
Server answers with the P2ABCE response XML
IoT device resumes the communication with the Verifier
Proof to Verifier: presentationToken.xml

Fig. 5. Messages exchanged during the proving delegation.

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.

Regarding the definition of the P2ABCE API, it is a work in progress, and in the meantime, we will work with the REST API currently available in the P2ABCE project to perform the delegation. Regarding the *IoT Smart Card*, the P2ABCE project defines the APDU instructions with the corresponding format of the APDU Commands and Responses. In the next section we will explain the PoC for IoT based on these P2ABCE specifications.

IV. Proof of Concept Implementation

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

We will develop our PoC in Linux based systems aimed for IoT environments, that will serve as a starting point for future implementations in more constrained devices or different systems.

In our delegation server we will run Raspbian OS in a Raspberry Pi3, and for our IoT device, we will use LEDE, Linux Embedded Development Environment, a distribution forked from OpenWrt, aimed for routers and embedded chips with low resources requirements.

A. PoC Delegation

The delegation process has two steps, first the IoT device calls the P2ABCE server to offload the parsing of the XML data, and then the P2ABCE server sending APDU Commands to the IoT smart card in the device.

1) PoC Delegation to the P2ABCE Server: Currently P2ABCE offers multiple REST web services to run different roles in P2ABCE system: User Service, Issuer Service, Verification Service, etc. After an analysis of P2ABCE's code, the `HardwareSmartcard` class implements the `Smartcard` interface using the package of abstract classes `javax.smartcardio` to communicate with the physical smart cards. The Oracle JRE implements these package for the majority of smart card manufacturers. For our PoC, we implement the `javax.smartcardio` package so it transmits the APDUs with our custom protocol, making the use of a physical or IoT smart card totally transparent to the `HardwareSmartcard` class, enhancing maintainability, and following the *expert pattern* from the known GRASP guidelines.

In this PoC, the P2ABCE's User Service was modified to add a new method receiving the IoT device's IP address and a port where the IoT Smart Card will be listening for the APDU Commands. This method creates a new *HardwareSmartcard* object, but instead of the *javax.smartcardio* Oracle's *CardTerminal* implementation, we use our own *IoTsmartcardio* implementation for the *HardwareSmartcard* constructor.

The remaining REST methods are left untouched, and will serve to parse the XML files exchanged.

2) APDU Dialogue Transmission: To transmit the APDU messages in our PoC we use a simple protocol, that we will refer as BIOSC (Basic Input Output Smart Card). It consists on one byte for the instruction, that can mean either an APDU Commnad or a finishing instruction to close the connection.

In the first case, the header continuous with two more bytes for the length of the payload bytes to read, which are the APDU Command bytes. To send an APDU Response in BIOSC, the IoT device sends back to the server two bytes for the length and then the raw APDU Response bytes. The messages are sent over TCP for a reliable transmission.

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

B. IoT Smart Card Implementation

In this section we present the code architecture and sequence diagram of the IoT Smart Card PoC, based on ABC4Trust's Card Lite.

CODE STRUCTURE We divide the project in three different sections with the objective of enhancing maintainability, improving future changes, ports, fixes, etc. In Figure 6 we show how the project is structured in three sections:

Core smart card: The smart card logic lies here, with the concepts of APDU Commands, the instructions that are

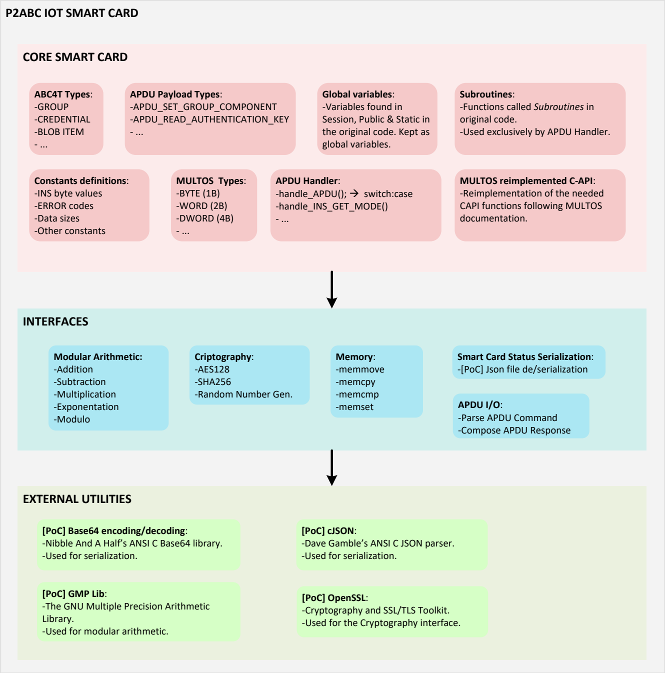

P2ABC IOT SMART CARD
CORE SMART CARD
ABC4T Types:
-GROUP
-CREDENTIAL
-BLOB ITEM
- ...
APDU Payload Types:
-APDU_SET_GROUP_COMPONENT
-APDU_READ_AUTHENTICATION_KEY
- ...
Global variables:
-Variables found in Session, Public & Static in the original code. Kept as global variables.
Subroutines:
-Functions called Subroutines in original code.
-Used exclusively by APDU Handler.
Constants definitions:
-INS byte values
-ERROR codes
-Data sizes
-Other constants
MULTOS Types:
-BYTE (1B)
-WORD (2B)
-DWORD (4B)
- ...
APDU Handler:
-handle_APDU(); → switch:case
-handle_INS_GET_MODE()
- ...
MULTOS reimplemented C-API:
-Reimplementation of the needed CAPI functions following MULTOS documentation.
INTERFACES
Modular Arithmetic:
-Addition
-Subtraction
-Multiplication
-Exponentation
-Modulo
Criptography:
-AES128
-SHA256
-Random Number Gen.
Memory:
-memmove
-memcpy
-memcmp
-memset
Smart Card Status Serialization:
-[PoC] Json file de/serialization
APDU I/O:
-Parse APDU Command
-Compose APDU Response
EXTERNAL UTILITIES
[PoC] Base64 encoding/decoding:
-Nibble And A Half's ANSI C Base64 library.
-Used for serialization.
[PoC] cJSON:
-Dave Gamble's ANSI C JSON parser.
-Used for serialization.
[PoC] GMP Lib:
-The GNU Multiple Precision Arithmetic Library.
-Used for modular arithmetic.
[PoC] OpenSSL:
-Cryptography and SSL/TLS Toolkit.
-Used for the Cryptography interface.

Fig. 6. IoT Smart Card Code Structure.

defined for P2ABCE smart cards, and how to process them and generate proper APDU Responses.

Here we adapted the ABC4Trust Card Lite's code. However, the ABC4Trust's code heavily depends on the MULTOS platform, therefore, we reimplement the used MULTOS functions following the documentation, adapting it in some cases for *expanded functionality*, to avoid some MULTOS API limitations, like in [21], when they noted that MULTOS' `ModularExponentiation` function does not accept exponents larger than the modulus size, so they implemented `SpecialModularExponentiation`.

MULTOS compiler does not apply data structure alignment. This affects the inherited ABC4Trust's code because of the use of `memcpy` to copy multiple variables in one call.

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

Interfaces: To reimplement some of the MULTOS functions, 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. With this facade we could, for example, change the implementation of cryptographic functions with a hardware implementation (e.g. Atmel's chips for SHA and AES), or software implementations optimised for the target platform.

The interfaces defined can be organized in 5 groups (see Figure 6), depending on their purpose: Modular Arithmetic, Cryptography, Memory Management, Serialization and APDU Parsing.

External utilities: In our PoC we used two ANSI C libraries, for base64 and JSON, and two shared libraries available as packages in LEDE: GMPLib and OpenSSL. These libraries use dynamic memory and offer more functionality than we need, and although they are useful tools for the early development versions of the PoC, future versions should use more lightweight solutions.

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

Execution workflow

The sequence diagram from Figure 7 shows the execution of the PoC IoT smart card.

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

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

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

Finally, every instruction handler must end, before the `return;` expression, with a call to `mExit`. This reimplemented MULTOS function will save the current status of the smart card and send the APDU Response back to the delegation server.

After returning from these functions, `mExit`, the instruction handler and the APDU handler, the program listens again from the socket.

V. Validation and Performance Evaluation

In this section, we will describe the deployment of three testing scenarios: a laptop, a Raspberry Pi 3, and an 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 runs as expected and is feasible.

A. Testbed description

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

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[7]. 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,

[7] https://github.com/p2abcengine/p2abcengine/wiki/

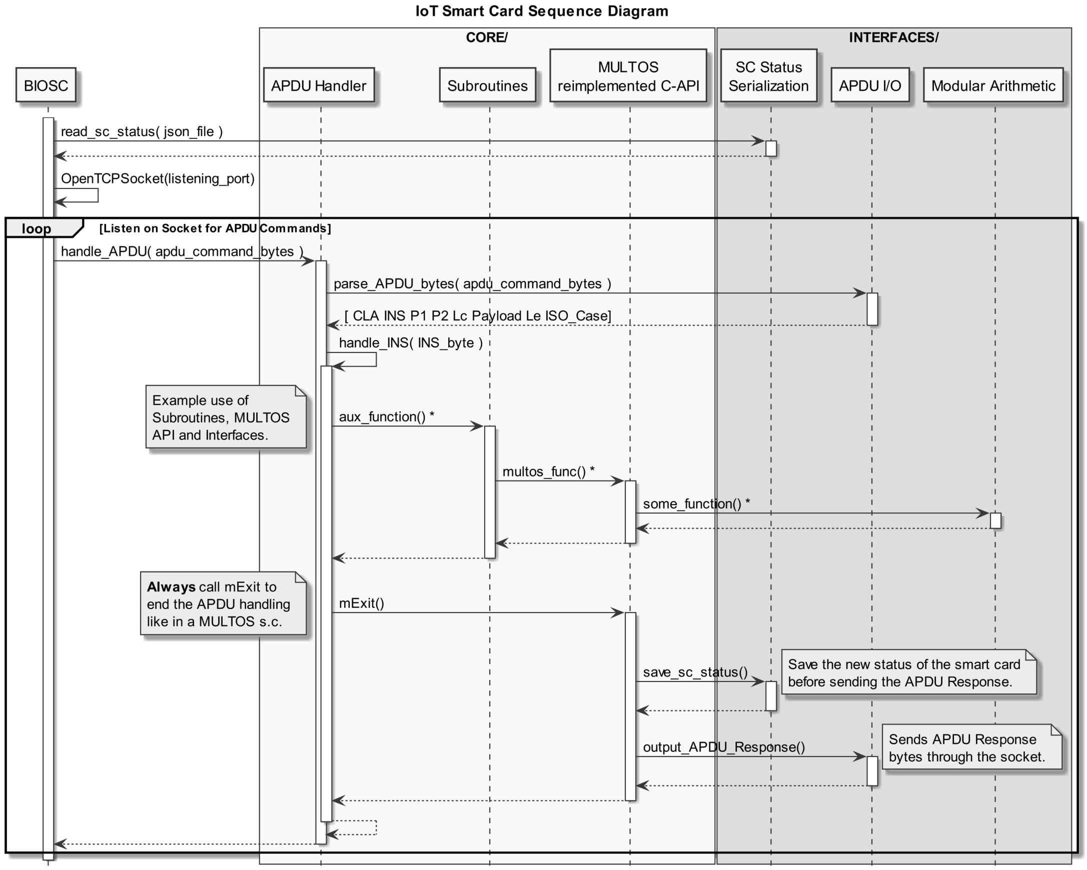

IoT Smart Card Sequence Diagram
CORE/
INTERFACES/
BIOSC
APDU Handler
Subroutines
MULTOS reimplemented C-API
SC Status Serialization
APDU I/O
Modular Arithmetic
read_sc_status(json_file)
OpenTCPSocket(listening_port)
loop
[Listen on Socket for APDU Commands]
handle_APDU(apdu_command_bytes)
parse_APDU_bytes(apdu_command_bytes)
[CLA INS P1 P2 Lc Payload Le ISO_Case]
handle_INS(INS_byte)
Example use of Subroutines, MULTOS API and Interfaces.
aux_function() *
multos_func() *
some_function() *
Always call mExit to end the APDU handling like in a MULTOS s.c.
mExit()
save_sc_status()
Save the new status of the smart card before sending the APDU Response.
output_APDU_Response()
Sends APDU Response bytes through the socket.

Fig. 7. IoT Smart Card Sequence Diagram.

a random presentation token is inspected and the winning member is notified.

First go through the **setup** phase, where several artifacts are generated and distributed to the various P2ABCE entities. 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, where a *presentation policy* from the verifier 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.

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[8], exactly like in the laptop. Finally, we will deploy the IoT smart card in an Omega2[9] 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. A downside of using the Omega2 is the lack of hardware acceleration for cryptographic operations, unlike the MULTOS smart cards.

a) 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 taken from previous executions, and the results show that the mean transmission time is less than half a millisecond per APDU. From our analysis of the tests we conclude that the network time is negligible compared to the rest of the computations.

B. Results

After 20 executions for each scenario (laptop, RPi3, Omega2+RPi3), we measure the time of each REST call executed against the User Service in the delegation server and take the means to compare each step of the testbed. Because during a call to the User Service, the server may contact the

[8] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

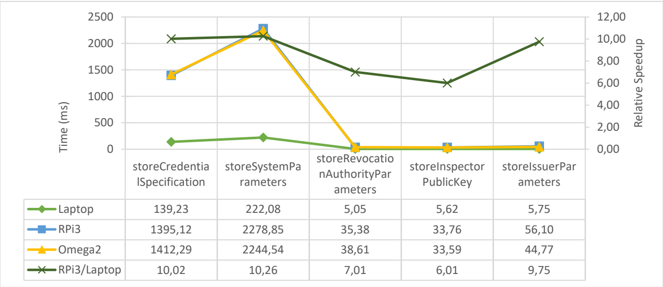
[9] https://docs.onion.io/omega2-docs/omega2.html

IoT Smart Card, the difference in times between the second and third scenario will show the performance of our PoC.

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.

a) The setup:

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



	storeCredentialSpecification	storeSystemParameters	storeRevocationAuthorityParameters	storeInspectorPublicKey	storeIssuerParameters
Laptop	139,23	222,08	5,05	5,62	5,75
RPi3	1395,12	2278,85	35,38	33,76	56,10
Omega2	1412,29	2244,54	38,61	33,59	44,77
RPi3/Laptop	10,02	10,26	7,01	6,01	9,75

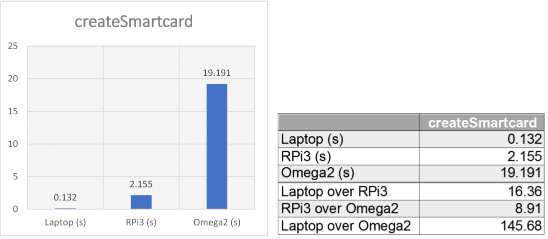
Fig. 8. Setup times (milliseconds) and relative speedup.

As we can see in Figure 8, 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.

b) Creation of the smart card:

Here we create a *SoftwareSmartcard* or a *HardwareSmartcard*, pointing to the *IoT Smart Card*, object that the User service will use in the following REST calls.

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



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

Fig. 9. Create smart card times (seconds) and relative speedup.

From Figure 9 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 for a one time process.

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, 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.

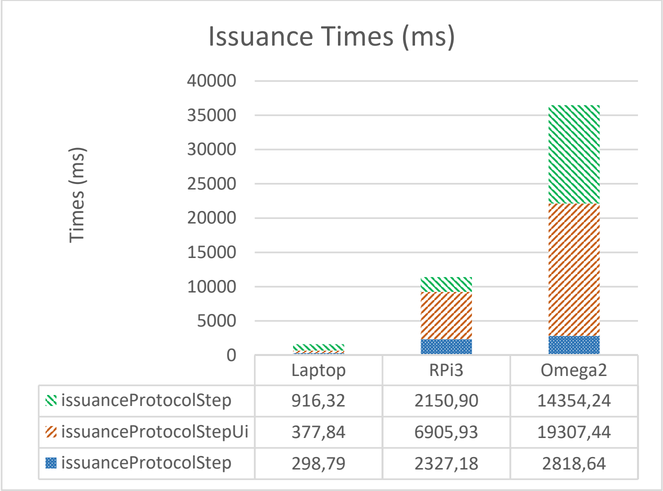
c) Issuance of the credential:

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

The three delegation steps and the REST method called are:

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

As we can see, the three REST calls to the delegation User service involve communication with the smart card. There are 45 APDU Commands in total, 3197 bytes exchanged, that would introduce a latency of 45ms in the network, negligible.



	Laptop	RPi3	Omega2
▧ issuanceProtocolStep	916,32	2150,90	14354,24
▨ issuanceProtocolStepUi	377,84	6905,93	19307,44
▦ issuanceProtocolStep	298,79	2327,18	2818,64

Fig. 10. Issuance times (milliseconds)

In Figure 10 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 for an IoT environment.

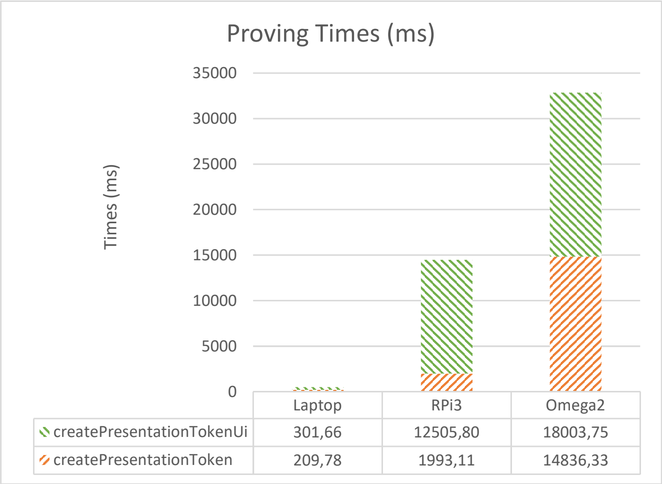
Lets compare the Raspberry Pi 3 and the Omega2 scenarios. There is a correlation between the APDU Commands used in each step with the increment in time when using the *IoT Smart Card*. During the first REST call, only one APDU pair (Command and Response) was used, with 33 bytes total, and times are almost 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, we can see the difference in processing power between Omega2 and Raspberry Pi 3.

d) 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.

To ensure that all the process was successful, it's enough to check with the Verifier and the Inspector they accepted the prove or returned an error code. Of course, every execution measured in the test was successful, validating the PoC implementation.

Again, as shown in Figure 11, 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.



	Laptop	RPi3	Omega2
▨ createPresentationTokenUi	301,66	12505,80	18003,75
▨ createPresentationToken	209,78	1993,11	14836,33

Fig. 11. Proving times (milliseconds)

Unlike the previous steps, the Presentation or Proving is done more than once, being the key feature of P2ABC ecosystem. 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.

e) 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, we measure the *IoT Smart Card* binary running in the Omega 2, after the described testbed execution.

After another round of tests, the field in the `time` output 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 debugging purposes, it stores a copy of every saved variable in the JSON tree structure, then creates a string with the JSON, that the user can then write to the save 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, an optimized binary serialization method, and many other improvements, are part of 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 even more constrained devices.

C. Validation conclusions

Below Table I sums up the time in seconds used in each step of the test for the Omega2 scenario.

The first step, *System Setup* is done only once when the system is being deployed, and the *IoT Smart Card Setup* only once per device.

Because a device can have more than one credential, the Issuance step is significant when we issue multiple credentials over the device's lifetime. We recall that our tests used a credential with five attributes and key sizes of 1024 bits.

Finally, the Proving step is expected to be the most commonly performed operation. The fact that it lasts over half a minute implies that we should not use this PoC for *real-time* applications yet. Nevertheless, for many other IoT applications, the fact this operation can be performed multiple times per hour, presents an useful tool for privacy.

System Setup	IoT Smart Card Setup	Issue credential	Prove Presentation Policy
3.77 s	19.19 s	36.48 s	32.84 s

TABLE I
TOTAL TIME SPENT FOR EACH STEP IN THE OMEGA2+RPI3 SCENARIO.

VI. CONCLUSIONS AND FUTURE WORK

This paper has presented a generic privacy-preserving solution for the vast world of the Internet of Things. The IoT devices can operate as individual actors in the P2ABCE ecosystem, and when in need of performing computation offloading, the delegation server can also be a device considered into the IoT family. Our PoC implementation demonstrates that this project is actually feasible in constrained devices, not by performing a simulation of an IoT device, like in [13] or [2], but deploying it in a real IoT device.

On the design aspect, we mentioned a P2ABCE API to abstract the delegation process to other processes running in the IoT devices. To develop this API we should study the available solutions to decide how to delegate to the server, e.g. using REST, CoAP, RPC, and consider the security issues mentioned in previous sections. Then, we can define an standard API and implement it assuring a great level of trust in the technique. On the implementation side, it would be interesting to compare the execution of the current PoC to a version with cryptographic hardware acceleration, for example, using Atmel's chips for SHA, AES and secure memory.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgements

The project leading to this application has received funding from IBM 2015 Faculty Award for Cyber Security and the European Unions Horizon 2020 research and innovation programme under grant agreement No 700085 (ARIES project).

References

[1] Specification of the identity mixer cryptographic library (v2.3.43). Technical report, IBM Research, January 2013.

[2] Almudena Alcaide, Esther Palomar, José Montero-Castillo, and Arturo Ribagorda. Anonymous authentication for privacy-preserving iot target-driven applications. *Computers & Security*, 37:111–123, 2013.

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010. http://www.sciencedirect.com/science/article/pii/S1389128610001568.

[4] Jorge Bernal Bernabé, José Luis Hernández Ramos, and Antonio Fernandez Gómez-Skarmeta. Holistic privacy-preserving identity management system for the internet of things. *Mobile Information Systems*, 2017:6384186:1–6384186:20, 2017.

[5] P. Bichsel, J. Camenisch, T. Grob, and V. Shoup. Anonymous credentials on a standard java card. ccs, 2009.

[6] S. Brands and C. Paquin. U-prove cryptographic specification v1.0. tech. rep. Technical report, Microsoft, March 2010.

[7] S.A. Brands. Rethinking public key infrastructures and digital certificates: Building in privacy. mit press, August 2000.

[8] Jan Camenisch and Anna Lysyanskaya. An eficient system for non-transferable anonymous credentials with optional anonymity revocation. *Advances in Cryptology EUROCRYPT 2001*, 2001.

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

[10] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. *Advances in CryptologyCRYPTO'97*, pages 410–424, 1997.

[11] Jan Camenisch and Markus Stadler. Eficient group signature schemes for large groups. *Advances in Cryptology CRYPTO 97*, 1997.

[12] Luuk Danes. Smart card integration in the pseudonym system idemix. Master's thesis, Faculty of Mathematics. University of Groningen, 2007.

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

[14] Tom Henriksson. How strong anonymity will finally fix the privacy problem. *VentureBeat*, October 2016. https://venturebeat.com/2016/10/08/how-strong-anonymity-will-finally-fix-the-privacy-problem/.

[15] Vctor Sucasas Iglesias. Implementation of an anonymous credential protocol. Master's thesis, Escuela Tcnica Superior de Ingenieros de Telecomunicacin. Universidad de Vigo, 2008-2009.

[16] Jennifer Seberry Josef Pieprzyk, Thomas Hardjono. *Fundamentals of Computer Security*. Springer, 2003.

[17] Gregory Neven. A quick introduction to anonymous credentials, August 2008.

[18] Giuseppe Persiano and Ivan Visconti. *An Efficient and Usable Multi-show Non-transferable Anonymous Credential System*, pages 196–211. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[19] 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*. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 96. Springer-Verlag Berlin Heidelberg, 1 edition, 2012.

[20] M. Sterckx, B. Gierlichs, B. Preneel, and I. Verbauwhede. Efficient implementation of anonymous credentials on java card smart cards. in: Wifs, 2009.

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