Integration of Idemix in IoT 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*—In this paper we design a solution to integrate IBM's Identity Mixer into the IoT ecosystem, integrating its privacy-preserving capabilities thanks to the use of Zero-Knowledge Proof cryptography. To validate our design, we implemented a functional Proof of Concept written in C for IoT devices. The applications of such solution promise to improve both the security and privacy of IoT, which in recent years have proved to be compromised.**

I. Introduction

The IoT is a term with a wide range of interpretations [2], breafly, we can think of it as billions of devices, mainly resource constrained, that are interconnected between them, and the Internet, in order to achieve a goal.

Many of this objectives require the use of a great amount of data, and thanks to organizations like WikiLeaks, people are aware of the implications of their data on the Internet, demanding more security and privacy for it. This includes not only the data shared with others, where one must trust they will keep it safe, but it's also the data collected about us and that we don't have direct control over it.

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

With the proliferation of IoT devices gathering as much information as they can with their sensors, the amount of data stored about anyone can be immense. And IoT has proved to not address neither security nor privacy, with recent events like the Mirai botnet DDoS attack on October 2016, considered the biggest DDoS in history [15], or the multiple vulnerabilities affecting house devices, like baby monitors [18].

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

Intuitively, an attribute-based credential can be thought of as a digital signature by the Issuer on a list of attribute-value pairs, e.g. the list `(fname=Alice, lname=Anderson, bdate=1977-05-10, nation=DE)` [16]. The most straightforward way for the User to convince a Verifier of her list of attributes would be to simply transmit her credential to the Verifier. With anonymous credentials, the User never transmits the credential itself, but rather uses it to convince the Verifier that her attributes satisfy certain properties without leaking anything about the credential other than the shown properties. This has the obvious advantage that the Verifier can no longer reuse the credential to impersonate `Alice`. Another advantage is that anonymous credentials allow the User to reveal a selected subset of her attributes. Stronger even, apart from showing the exact value of an attribute, the User can even convince the Verifier that some complex predicate over the attributes holds, e.g. that her birth date was more than 18 years ago, without revealing the real date.

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

To understand how ZKPs work, in 1990 Guillou, Quisquater and Berson published in *How to Explain Zero-Knowledge Protocols to Your Children* [14] a story about how Ali Baba proved that he knew the magic words to open the cave, but without revealing those words to anyone.

Based on ZKP properties, IBM has developed the Identity Mixer[1], Idemix for short, protocol suite for privacy-preserving authentication and transfer of certified attributes. It allows user authentication without divulging any personal data. Users have a personal certificate with multiple attributes, but they can choose how many to disclose, or only give a proof of them, like being older than 18 years-old, living in a country without revealing the city, etc. Thus, no personal data is collected that needs to be protected, managed, and treated.

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

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

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

and authorization, based on Anonymous credential Systems, like Idemix.

As part of IBM's academic grant for *Privacy Preserving Identity Management applied to IoT*, the goal of this project is to integrate Idemix with the IoT. It will be done using ABC4Trust's P2ABCE, a framework that defines a common architecture, policy language and data artifacts for an attribute based ecosystem, cryptographically based on either IBM's Idemix or Microsoft's U-Prove [2]. This gives us a standardized language to exchange Idemix's messages between IoT devices and any other P2ABCE actor.

Once the IoT devices can execute Idemix, the new step is to take advantage of it in other deployments. In a smart building, different privacy policies could protect sensitive data, like how many people there are, and their identities, where a thermostat would only need to know if the amount of people is between some boundaries, but in case of emergency, the police department could request all the information available. We think that the first approach to achieve these ideas is first to integrate Idemix in known IoT identity systems, like FIWARE project.

This document is structured as follows: In section II we show a state of the art analysis through the history of Idemix and related works, analysing what is of the most interest for the IoT perspective; in section III we describe the formal design of the IoT and Idemix solution, and describe the PoC implementation developed; after an implementation, it is a must to validate it, as showed during the performance tests in section IV; finally, our conclusions and lines for future work are described in section V.

II. State of the art

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

Persiano and Visconti presented a non-transferable anonymous credential system that is multi-show and for which it is possible to prove properties (encoded by a linear Boolean formula) of the credentials [13]. Unfortunately, their proof system is not efficient since the step in which a user proves possession of credentials (that needs a number of modular exponentiations that is linear in the number of credentials) must be repeated times (where is the security parameter) in order to obtain a satisfying soundness.

Stefan Brands provided the first integral description of the U-Prove technology in his thesis [5] in 2000, after which he founded the company Credentica in 2002 to implement and sell this technology. Microsoft acquired Credentica in 2008 and published the U-Prove protocol specification [4] 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 [17].

Jan Camenisch, Markus Stadler and Anna Lysyanskaya studied in [8], [6] and [7] 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 [9], identifying what data and operations should be kept inside the device to perform different levels of security. The User role was divided between the smart card, holding secret keys, and the Idemix terminal, that commanded operations inside the smart card, or read the keys in it to perform the instructions itself. The studied sets were:

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

Later, in 2008 Vctor Sucasas also studied an anonymous credential system with smart card support [12], 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 [3] and [19], but they weren't optimal and didn't include some Idemix's functionalities, like selective disclosure.

Later, in 2013, Vullers and Alpar, implemented an efficient smart card for Idemix [20], aiming to integrate it in the IRMA[3] project, and comparing the performance with U-Prove's smart cards. This new implementation was written in C, under the MULTOS platform for smart cards, and describes many decisions made during the development to improve the performance on such constrained devices. The terminal application was written in Java and used an extension of the Idemix cryptographic library to take care of the smart card specifics.

Extending the concept of smart cards, physical or logical, as holders of the credentials, the ABC4Trust's project, P2ABCE[4], was created as a unified ABC system for different cryptographic engines, currently supporting U-Prove and Idemix. The Idemix library was updated to support P2ABCE and the last version is interoperable with U-Prove. Therefore, the smart card specification from the P2ABCE project could be considered the official version to work with.

[2]P2ABCEngine https://github.com/p2abcengine/p2abcengine

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

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

Related to the IoT, the P2ABCE project has been used to test in a VANET[5] scenario how an OBU (On Board Unit), with constrained hardware, could act as a User in an P2ABC system [10]. However, after the theoretical analysis, the paper only simulates a computer with similar performance as an OBU, without adapting the existing Java implementation of P2ABCE to a real VANET system. In our project, we can consider ourselves as part of their *future work*, because our PoC will run on hardware actually used in VANET systems, and has been implemented in C instead of Java, which is more realistic and efficient for its deployment in an OBU.

III. Design and implementation

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

A. Design

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

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

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

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

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

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

Using the *computation offloading* technique to our scenario, our design consists on implementing the smart card logic inside the IoT device, keeping secure our master key and credentials, and for the rest of the P2ABCE system, if the device can not run the complete Engine, it may delegate to a server running it, indicating how to send APDU Commands to the *IoT smart card*.

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

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

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

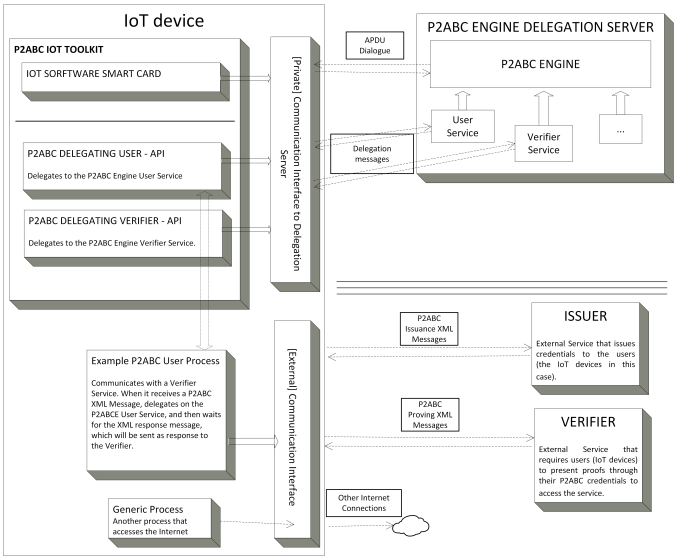

Fig. 1. Proposed high level Architecture for integrating IoT devices in P2ABCE.

1) System architecture:

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

- **IoT device**
  In Figure 1 shows our proposed architecture, in which the IoT device is represented with two interfaces, physical

[5]Vehicular Ad-Hoc Network

or virtual. One allows external communications to other machines, including other P2ABCE actors, that could be on the Internet, a corporate LAN, a M2M overlay network, etc. Through this interface, the P2ABCE XML messages are exchanged as in any other deployment. This allows an IoT device to interact with other actors without special adaptations to the protocol. The other interface allows a secure communication with the delegation server. Both the delegation messages and the APDU Dialogue are transmitted over this interface, making it a point of attack to the system, and we will talk about its security in the delegation process.
The scheme also shows the *P2ABCE IoT Toolkit*. This piece of software includes the IoT Smart Card, and the API for other processes that want to use the P2ABCE system.
The IoT Smart Card is the implementation of a software smart card, listens for APDU Commands from the secure interface and stores securely the credentials and private keys within the device's memory.
The P2ABCE API is an interface for other processes that wish to use the private-preserving environment of P2ABCE. It provides access to every operation available, hiding the delegation process. In the future, if for example the Verification Service is implemented for the IoT device, i.e., there's no need to delegate to other machine to act as a Verifier, then any program using the API won't need to change anything, the toolkit conceals the transition from delegating to native execution.

- **P2ABCE actors**
  If we recall from **??**, the possible roles in the system were the Issuer, the User, the Verifier, the Revocation Authority and the Inspector. All of them use the P2ABCE XML schema in the specification to communicate to each other. Any third party actor will be unaware of the fact that the device is a constrained IoT device that delegates on the P2ABCE server.
- **P2ABCE Delegation Server**
  The machine in charge of receiving authorized IoT devices' commands to parse the XML files exchanged and orchestrate the cryptographic operations the IoT smart card must perform.

Delegation process

Here we describe the computation offloading carried out by the IoT device. In Figure 2 we show an example of the IoT acting as a User, Proving a Presentation Policy to a third party Verifier.

1) Communication with P2ABCE actor.
   The IoT device, acting as a User, starts an interaction with another actor, e.g., against an Issuer to obtain a signed credential, or against a Verifier to demonstrate certain property of its attributes in a privacy-preserving way.
2) Delegation to the P2ABCE Server.
   Depending on what role the IoT device is acting as, it will delegate in the corresponding service, e.g. User Service. The delegation message must include the XML file, and any parameter required to accomplish the task, like the information on how to communicate with the IoT smart card (listening port, security challenge, etc.).
3) APDU Dialogue (if necessary).
   The server may need to send APDU Commands to the IoT smart card to read the credential information or perform cryptographic operations involving private keys, necessarily stored inside the IoT device.
   These APDU Commands include a list of 70 different instructions, identified by the APDU INS byte. Some of them manage the smart card, like changing the PIN code, and many are P2ABCE specific, like storing the system cryptographic parameters.
4) Server response.
   The server may return a status code or a XML file if the first one required an answer from the IoT device, in which case, it will send as response to the third party actor, resuming the communication.

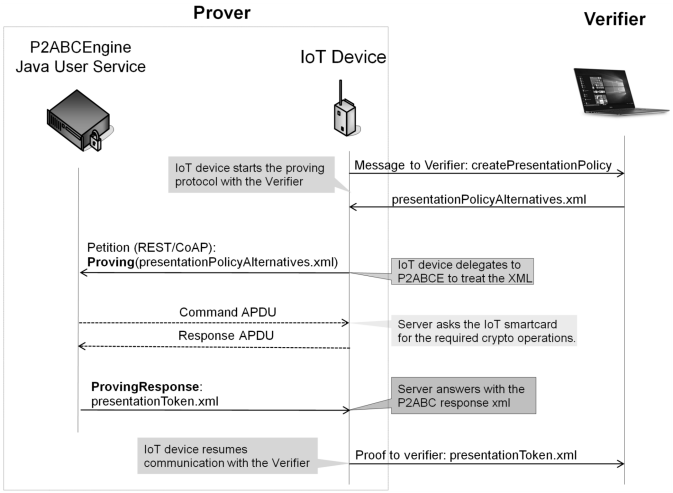

Fig. 2. Designed interactions exchanged during the IoT delegation for P2ABCE Proving operation.

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

We could use a corporative PKI to issue certificates to the server and devices and configure policies for access control; design a challenge-response system combined with the smart card PIN, like a password and TOTP[6] in a 2FA[7] login. We also could connect physically the delegation service through RS-232 serial to the IoT device, securing both physically as we would do with the IoT device on its own, isolating the delegation system from any network attack. This last idea is

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

an approximation to the Arduino Yn[8], a development board that integrates two microcontrollers, one a typical Arduino with very low resources, and another one running a fork of OpenWrt. The Arduino microcontroller can control the terminal of the more powerful one, using the serial pins as commented before.

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

a) Notes for more constrained devices: Our architecture is designed for devices that could in a future run a reimplemented version of P2ABCE, that means, the devices could perform more tasks than only running the smart card software and their main purpose process, e.g. recollecting sensor data. But if our target devices are so constrained that can barely run the smart card, they may not be able to handle the XML files because of memory restrictions, like a MSP430[9] running Contiki-OS, the microcontroller has between hundred of bytes to tens of kilobytes of memory, making impossible to store multiple XML files in the size range of tens of kilobytes.

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

B. 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.

C. IoT system

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

In our delegation server we will run Raspbian OS, a distribution based on Debian for the Raspberry Pi. 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.

[8]https://www.arduino.cc/en/Main/ArduinoBoardYun

[9]http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/overview.page

D. PoC Delegation

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

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

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

/initIoTsmartcard/issuerParametersUid?host=&port=

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

2) APDU Dialogue Transmission: To transmit the APDU messages in our PoC we use a simple protocol, that we will refer as BIOSC (Basic Input Output Smart Card), consisting in one first byte for the instruction. In the first instruction, we read two header bytes with the length of the APDU Command or Response to receive, followed by said APDU bytes. The message is sent over TCP for a reliable transmission (concept of session, packet retransmission, reordering, etc.). The second instruction serves for closing the TCP socket in the IoT device when the smart card is no longer needed.

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

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

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 BIOSC, making the use of a physical or IoT smart card totally transparent to the `HardwareSmartcard` class, enhancing maintainability,

and following the *expert pattern* from the known GRASP guidelines.

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

E. IoT Smart Card Implementation

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

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

In Figure 3 we show the project's structure. The first section is what could be called as the core of the smart card, the second one the interface for those tools the core needs and that may depend on the target's platform, and finally, the third party libraries, that may be empty if the interfaces implementation doesn't require any.

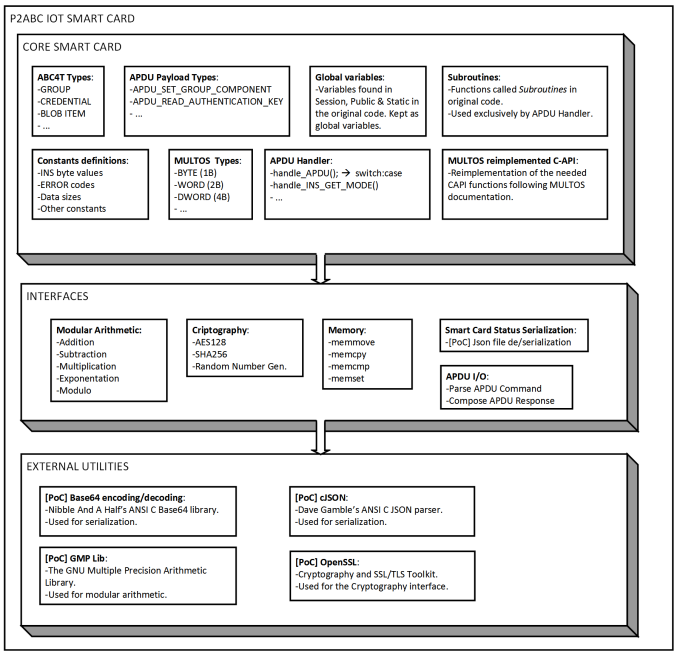

Fig. 3. IoT Smart Card Code Structure.

Core smart card

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

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

However, the ABC4Trust's code heavily depended on the MULTOS platform for the input and output of data, modular arithmetic, memory management, and AES128 and SHA256 cryptographic operations. Therefore, we reimplement the used MULTOS functions following the documentation, adapting it in some cases for *expanded functionality* when needed, like in [20] when they noted that MULTOS' `ModularExponentiation` function does not accept exponents larger than the modulus size, so they implemented `SpecialModularExponentiation`.

In MULTOS, to reduce the memory usage, the MULTOS compiler does not apply padding between variables in the data structures, i.e., data structure alignment[10]. This affects the inherited ABC4Trust's code because of the use of `memcpy` to copy zones of memory from one address to another. The problem is the code copies multiple variables in one `memcpy` call, because, for example, the APDU Command payload includes multiple data, and instead of copying one variable at a time, the `struct` is defined with the same order and copies everything at once.

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

Interfaces

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

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

The interfaces defined can be organized in 5 groups (see Figure 3), depending on their purpose: Modular Arithmetic, Cryptography, Memory Management, Serialization, APDU Parsing.

External utilities

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

[10]https://en.wikipedia.org/wiki/Data_structure_alignment

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

2) Execution workflow: The sequence diagram from Figure 4 shows the execution of the PoC IoT smart card.

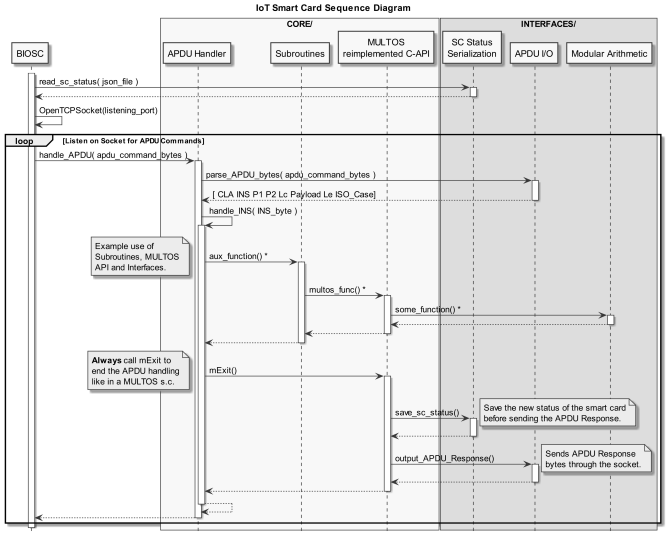

Fig. 4. IoT Smart Card Sequence Diagram.

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

IV. Validation and Performance Evaluation

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

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[11]. It is based on a soccer club, which wishes to issue VIP-tickets for a match. The VIP-member number in the ticket is inspectable for a lottery, ie. after the game, a random presentation token is inspected and the winning member is notified.

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

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

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

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

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

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[12], exactly like in the laptop. Finally, we will deploy the IoT smart card in a Omega2[13] 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 applications, because the smart cards hardware and MULTOS API include support for such common operations in 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, 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.

[11] https://github.com/p2abcengine/p2abcengine/wiki/
[12] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[13] https://docs.onion.io/omega2-docs/omega2.html

B. Results

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

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

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

a) The setup:

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

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

Fig. 5. Setup times (milliseconds). Times and relative speedup

As we can see in Figure 5, 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* object that the User service will use in the following REST calls.

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

Fig. 6. Create smart card times (seconds). Times and relative speedup

	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

From Figure 6 we see that the RPi3 is about 16 times slower than the laptop in the creation of the *SoftwareSmartcard*, but almost 9 times faster than the setup of the smart card in the Omega2 using APDUs. This gives us that the laptop is 145 times faster than the combination of RPi3 and Omega2 in our IoT deployment. But looking at the times, this process lasts up to 20 seconds, making it something feasible.

This is the first interaction between the RPi3 and the IoT smart card running in the Omega2. To setup the smart card 30 APDU Commands, and their respective Responses, are exchanged, 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	$/issuanceProtocolStepUi$
(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. There are 45 APDU Commands in total, 3197 bytes exchanged, that would introduce a latency of 45ms in the network, negligible.

	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

Fig. 7. Issuance times (milliseconds). Times (ms) and relative speedup

In Figure 7 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.

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 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.

The APDU Commands and Responses pairs sent are 28 in total, with 1939 bytes, giving us about 27ms of delay in the network transmission.

Again, as shown in Figure 8, 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*.

	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

Fig. 8. Proving times (milliseconds). Times (ms) and relative speedup

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.

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, 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.

C. Validation conclusions

Below Table I sums up the time in seconds used in each step of the test for the Omega2 and Raspberry Pi 3 setup.

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 performed calculation by the IoT device. The fact that it lasts over half a minute implies that we should not use this PoC for *real-time* applications yet, usually used in critical secure systems. Nevertheless, for many other IoT applications, the fact this operation can be performed multiple times per hour, presents an useful tool for privacy, e.g. the thermostat system mentioned in the smart building, which can pool data from the sensors every few minutes.

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 SETUP.

V. CONCLUSIONS AND FUTURE WORK

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

A. Conclusions

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

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

During the development, we had to investigate a lot of concepts related to IoT, smart cards, and even the insides of P2ABCE's code, to fix many existing bugs in the original project and minimize the amount of changes it had to undergo, in order to work with the IoT devices. In the implementation chapter we give guidelines to port MULTOS applications, considering the particularities we encountered, that's why we

can't consider it an *instruction manual to port MULTOS apps*, but an interesting reading on how to confront a similar project. For example, if we wanted the Idemix implementation from [20], previous to P2ABCE, in a IoT device, we could apply almost the same steps, obtaining a core functionality of Idemix for constrained devices in C.

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

Finally, we are very thankful to IBM's grant for the *Privacy Preserving Identity Management applied to IoT* project, which allowed us to get this far.

B. Future work

Due to the nature of the project, there exist many options to continue researching in this area.

In the implementation side, we would continue with a second PoC which aimed for more constrained systems, maybe starting with Arduino, or another line of development would be to implement more P2ABCE functionality inside the IoT device. It would be interesting to compare the execution of the current pure software PoC to that with cryptographic hardware support, e.g. using Atmel's chips for SHA, AES and secure memory.

On the design aspect, now that an IoT device can perform P2ABCE operations, we should integrate it in a bigger project, like inside the IdM[14] system of FIWARE[15], to issue Idemix credentials to both users and IoT devices. And once the identification is achieved in a privacy-preserving fashion, more applications should be researched, like how the IoT device could benefit from the ZKP cryptography to prove information not issued in its credential, by expanding the P2ABCE presentation policies.

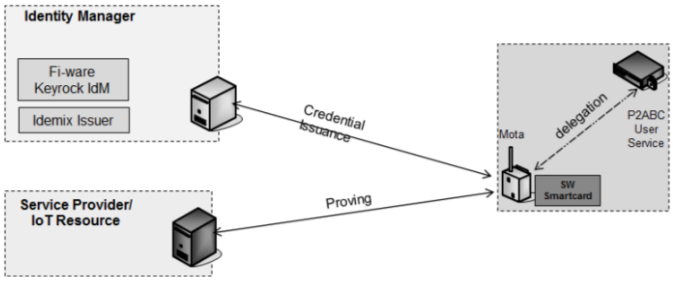

Fig. 9. IoT+Idemix Fi-Ware integration.

[14]Identity Management - KeyRock https://catalogue.fiware.org/enablers/identity-management-keyrock

[15]https://www.fiware.org/

REFERENCES

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

[2] 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.

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

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

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

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

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

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

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

[10] 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.

[11] 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/.

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

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

[14] Louis C. Guillou Jean-Jacques Quisquater and Thomas A. Berson. How to explain zero-knowledge protocols to your children. *Advances in Cryptology - CRYPTO '89*, 1990. Proceedings 435: 628-631. http://pages.cs.wisc.edu/~mkowalcz/628.pdf.

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

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

[17] 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.

[18] Mark Stanislav and Tod Beardsley. Hacking iot: A case study on baby monitor exposures and vulnerabilities. Technical report, Rapid7, 2015.

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

[20] 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.