Integration of Idemix in IoT environments

Jose Luis Canovas Sanchez, Antonio Skarmeta Gomez, Jorge Benal Bernabe

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

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

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

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

With usual symmetric and asymmetric cryptography it seems impossible to create such credentials, without an explosion of signatures over every possible combination. For this reason, current solutions rely on ZKPs, 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* [**?**] 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[2], Idemix for short, protocol suite for privacy-preserving authentication and transfer of certified attributes. It allows user authentication without divulging any personal data. Users have a personal certificate with multiple attributes, but they can choose how many to disclose, or only give a proof of them, like being older than 18 years-old, living in a country without revealing the city, etc. Thus, no personal data is collected that needs to be protected, managed, and treated.

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

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

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

[2]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 [3]. This gives us a standardized language to exchange Idemix's messages between IoT devices and any other P2ABCE actor.

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

A. Outline of this thesis

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 [**?**]. 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 [**?**] 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 [**?**] 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 [**?**].

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

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

Luuk Danes in 2007 studied theoretically how Idemix's User role could be implemented using smart cards [**?**], 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 [**?**], 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 [**?**] and [**?**], 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 [**?**], aiming to integrate it in the IRMA[4] 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[5], was created as a unified ABC system for different cryptographic engines, currently supporting U-Prove and Idemix. The Idemix library was updated to support P2ABCE and the last version is interoperable with U-Prove. Therefore, the smart card specification from the P2ABCE project could be considered the official version to work with.

Related to the IoT, the P2ABCE project has been used to test in a VANET[6] scenario how an OBU (On Board Unit), with constrained hardware, could act as a User in an P2ABC system [**?**]. 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

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

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

[6]Vehicular Ad-Hoc Network

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.

[width=]gfx/P2ABCE-IoT-bw

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 fig:P2ABCE-IoT shows our proposed architecture, in which the IoT device is represented with two interfaces, physical or virtual. One allows external communications to other machines, including other P2ABCE actors, that could be on the Internet, a corporate LAN, a M2M overlay network, etc. Through this interface, the P2ABCE XML messages are exchanged as in any other deployment. This allows an IoT device to interact with other actors without special adaptations to the protocol. The other interface allows a secure communication with the delegation server. Both the delegation messages and the APDU Dialogue are 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 analysisP2ABCE, 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 fig:DelegationProving we show an example of the IoT acting as a User, Proving a Presentation Policy to a third party Verifier.

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

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

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

[width=]gfx/UML/DelegationProving

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[7] in a 2FA[8] login. We also could connect physically the delegation service through RS-232 serial to the IoT device, securing both physically as we would do with the IoT device on its own, isolating the delegation system from any network attack. This last idea is an approximation to the Arduino Yn[9], 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[10] 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.

[7]Time-based One-time Password
[8]Two-factor authentication
[9]https://www.arduino.cc/en/Main/ArduinoBoardYun
[10]http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/overview.page

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. We will talk more about the hardware specifications in the benchmark chapter. We chose this system because it has a great package repository support, including the Java Runtime Environment, needed to run the P2ABCE Services.

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

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.

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

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

/initIoTsmartcard/issuerParametersUid?host=&port=

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

In a real deployment, we would offer a full library with an API for other processes that want to use the P2ABC Engine, which automatizes the boot of the IoT smart card and handles the security policies, like the one we presented in the deployment diagram 1 of the *P2ABCE IoT Toolkit*.

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

[language=bash] $curl - XPOST - -header'Content - Type : text/xml"http :$ //DelegationServerIP:9200/user/initIoTsmartcard/http

$curl - XPOST - -header'Content - Type : text/xml' - d@firstIssuanceMessage.xml"http :$ //DelegationServerIP:9200/user/issuanceProtocolStep/" ¿ issuanceReturn.xml

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

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

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

In the first instruction, we read two header bytes with the length of the APDU Command or Response to receive, followed by said APDU bytes. The message is sent over TCP for a reliable transmission (concept of session, packet retransmission, reordering, etc.). The second instruction serves for closing the TCP socket in the IoT device when the smart card is no longer needed.

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

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

As we mentioned in the analysis of P2ABCE's code, the `HardwareSmartcard` class implements the `Smartcard` interface using the package of abstract classes `javax.smartcardio` to communicate with the physical

smart cards. The Oracle JRE implements these package for the majority of smart card manufacturers. For our PoC, we implement the `javax.smartcardio` package so it transmits the APDUs with BIOSC, making the use of a physical or IoT smart card totally transparent to the `HardwareSmartcard` class, enhancing maintainability, and following the *expert pattern* from the known GRASP guidelines.

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

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 fig:IoTCScomponents-bw 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.

[width=]gfx/IoTCScomponents-bw

Fig. 3. IoT Smart Card Code Structure.

a) Core smart card:

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

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

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

However, the ABC4Trust's code heavily depended on the MULTOS platform for the input and output of data, modular arithmetic, memory management, and AES128 and SHA256 cryptographic operations. Therefore, we reimplement the used MULTOS functions following the documentation, adapting it in some cases.

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

The `main.h` file, from the original ABC4Trust project, also implemented some functions that were equivalent to the ones available in `multos.h`. We replaced them for the standard functions in the C-API, checking that the MEL code did the same as the documentation specified. This last detail is very important, because in vullers2013efficient they noted that MULTOS' `ModularExponentiation` function does not accept exponents larger than the modulus size, so they implemented `SpecialModularExponentiation`, dividing the exponentiation in two that MULTOS could perform. After this analysis, when those particularities appeared, our reimplementation of the MULTOS API accepts the *expanded functionality*. In total, we reduced to eighteen the number of MULTOS functions to reimplement, and with future refactoring this number may decrease.

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

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

Another of the mentioned particularities is that the MULTOS compiler does not apply padding between variables in the data structures. For example, in an X86 machine, where a `short` is stored in 2 bytes, in the following code[11]

[language=C] struct MyData short Data1; short Data2; short Data3; ;

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

[language=C] struct MixedData char Data1; short Data2; int Data3; char Data4; ;

[11]From https://en.wikipedia.org/wiki/Data_structure_alignment

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

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

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

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

[language=C,frame=tblr] typedef struct BYTE prover$_i d; unsigned int ksize; unsigned int csize; BYTE kx[MAX_SMALLINT_SIZE]; BYTE c[HASH_SIZE]; BYTE proofsession[$ $****/PROVER provers[NUM_PROVERS]$;

/*****/ case INS$_S ET_P ROVER$: $//[...] memcpy((provers[temp_prover_i d -$ $1].prover_i d), (apdu_d ata.set_p rover_i n.prover_i d), 5); //under the hood, this memcpy ... size$ This is the only case where a comment showcases that it is copying more than one byte.

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

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

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

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

[language=C] if(condition) return a; return b;

The use of `multosExit` in ABC4Trust code is similar, but used in almost every function. This leads to a sequence execution with many possible interruptions, leaving a full stack behind with *un-returned* function calls, i.e., parameters stored in memory after a function call that never returns. In MULTOS, this memory won't affect future executions, because with the next APDU Command, the stack will be loaded with the clean application, and the Session memory stays untouched between APDU Commands, during an APDU Dialogue.

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

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

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

b) Interfaces:

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

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

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

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

c) External utilities:

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

In our PoC, we use two ANSI C libraries, for base64 and JSON, and two shared libraries available in as packages in LEDE, GMPLib and OpenSSL. These libraries use dynamic memory and offer more functionality than we need, although

they are a good tool in this PoC, future versions should use more lightweight solutions.

[width=0.9gfx/example Atmel's cryptography chips.

For example, Atmel's ATAES132A[12] offers a serial chip for secure key storage, AES128 execution and random number generation. Another serial chip like ESP8266 offers WiFi connectivity, typically used with Arduino, and can also perform AES encryption. For random number generation, a technique used with Contiki devices is to read from sensors aleatory data and use it as seed.

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

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

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

[width=]gfx/UML/sequenceBIOSC

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.

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

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

F. Design and implementation conclusions

During this design and implementation process we had to study multiple technologies, including the ones mentioned in the state of the art, regarding Idemix, and those related to the IoT, we studied many systems (e.g. Contiki, Arduino) and hardware (e.g. MSP430, ESP8266), and after this, we had to take many decisions during development.

The design is open in those aspects that depend on each specific project's requisites, and closed on the aspects involving the P2ABCE's core logic.

However, the PoC had to make more concessions when implementing the cryptographic and modular arithmetic logic. But those are parts of the code that are more prone to be changed when developing for a new system. The core logic of the PoC is preserved as it is in every system and device.

Even with the many points of future work left, we think we achieved the most extensible solution for Idemix+IoT at the time, our main objective of the project.

IV. Validation and Performance Evaluation

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

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[13]. 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.

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

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

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

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

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

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

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

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

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

TABLE I
RASPBERRY PI 3 SPECIFICATIONS.

c) Onion Omega2: A device that falls inside the category of IoT, powerful enough to run a Linux embedded environment, such as LEDE, where we can develop and debug the first PoC without troubling ourselves with problems not related to the project itself.

MCU	Mediatek MT688[14]
CPU	MIPS32 24KEc 580MHz
RAM	64MB
Storage	16MB
Firmware	LEDE (OpenWRT fork distro)
Connectivity	Wifi b/g/n
Power	3.3V 300mA

TABLE II
ONION OMEGA2 SPECIFICATIONS.

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

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

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

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

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

B. The test code

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

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

a) P2ABCE services:

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

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

b) IoT smart card:

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

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

c) Shell scripts:

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

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

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

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

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

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

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

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

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

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

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

C. 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.

[Times and relative speedup] [width=]gfx/graphics/setuptable [Comparison graph] [width=.8]gfx/graphics/setup

Fig. 5. Setup times (milliseconds)

As we can see in fig:setup:graph, 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.

[Times and relative speedup] [width=0.5]gfx/graphics/createSCtable [Comparison graph] [width=0.45]gfx/graphics/createSC

Fig. 6. Create smart card times (seconds)

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

This is the first interaction between the RPi3 and the IoT smart card running in the Omega2. To setup the smart card **30 APDU Commands**, and their respective Responses, are exchanged, as shown in ch:resultsdiagrams, fig:APDUsInitIoTSC, 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 issuance is done in three steps for the User service, shown in fig:IssuanceInteraction with a red note showing the start of each step for the User delegation, in green for the interactions between Issuer and User, and the darker red is the Identity service, choosing the first available identity to use. The arrows in the figure show the REST calls performed during the test, where the IoT device acts as User, the RPi3 hosts the P2ABCE delegation services and the laptop is the Issuer.

The three delegation steps and the REST method called are:

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

[width=]gfx/UML/IssuanceInteraction

Fig. 7. Issuance interaction.

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

[Times (ms) and relative speedup] [width=]gfx/graphics/issuancetable [Comparison graph] [width=.8]gfx/graphics/issuance

Fig. 8. Issuance times (milliseconds)

In fig:issuance:graph 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. In fig:ProvingInteraction, using the same colors as in the Issuance interaction, we can see the delegation messages done by the Omega2.

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

In ch:resultsdiagrams, fig:APDUsProving, we provide the APDU Commands for each step, 28 in total, with 1939 bytes, giving us about 27ms of delay in the network transmission.

[width=]gfx/UML/ProvingInteraction

Fig. 9. Proving interaction.

Again, as shown in fig:proving:graph, 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*.

[Times (ms) and relative speedup] [width=.8]gfx/graphics/provingtable [Comparison graph] [width=.7]gfx/graphics/proving

Fig. 10. Proving times (milliseconds)

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

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.

D. Validation conclusions

Below totaltime 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 III
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 vullers2013efficient, 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 vanet. However, the use of third party libraries and no hardware acceleration support, makes the PoC too slow for certain cases, e.g., a Real-Time system requires almost immediate operations, like a car warning that another one is approaching too fast. There is a long path of research before we can see this design in production, as many decisions depend on the specific deployment in consideration.

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

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

B. Future work

Due to the nature of the project, there exist many options to continue researching in this area. We list below a collection

of the tasks we couldn't accomplish but would have been our next steps:

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

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

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

  GMPLib and OpenSSL libraries must be replaced by others with similar capabilities, or special implementations for the particular device in use.
  All these changes affect only the *external utilities* and *interfaces* sections of the code, leaving the *core* untouched, because the smart card logic doesn't change.
- Implement more P2ABCE functionality in the device
  If instead of aiming for more constrained devices, we work with similar devices to the Omega2, the computational power of the device could assume all the functionality of P2ABCE, at least for some roles. In a M2M (machine to machine) environment, the devices could act as Provers and Verifiers in the same interaction, and the verifying logic could be implemented entirely in the device (generate a Presentation Policy and checking a prove in a Presentation Token is valid), reducing the need of a delegation server.
- Improve the *core smart card* code
  From the simple refactoring of removing the MULTOS reimplemented API for direct calls to the *interfaces*, to identify all the `memcpy` copies of multiple variables with one call, removing the compiler directive for not using memory padding in the defined structures, letting the compiler manage the memory automatically.
- Use cryptographic hardware
  Atmel offers many solutions for secure memory and cryptographic operations chips, with serial interfaces for the IoT devices. We could benefit from this chips by storing the secret keys of the credentials inside the secure memory, and speed up the SHA256 and AES128 calculations, currently software implemented and slow compared to any hardware implementation.
- Support Extended APDUs
  Currently, the APDU instruction handlers use the GET RESPONSE command to send large APDU Response Payloads, but we can reduce the number of APDUs exchanged supporting the Extended APDU format. Because the P2ABCE already sent some Extended APDUs, the APDU I/O interface in the PoC has support for Extended APDU Commands, but not for Extended Responses.
- Integrate the IoT+Idemix solution in FIWARE
  Our sixth objective, that we couldn't get to design in depth. The original idea was to integrate in a FIWARE[15] environment the Fiware Keyrock and Idemix Issuer, so both, users and IoT devices' identities are managed in the IdM[16] system. The Issuer generates credentials based on the users attributes held in the Keyrock, and IoT devices can authenticate against Fi-ware services (using Idemix), or authenticate against other IoT devices (M2M).

[width=0.8]gfx/fiware

Fig. 11. IoT+Idemix Fi-Ware integration.

- Research P2ABCE policies for IoT applications
  The next step, after integrating Idemix with an Identity Management system, should be to design more complex applications to IoT scenarios. We talked about a smart building in the introduction, but the straightforward proofs we can present depend on the signed values in the credentials. The Zero-Knowledge Proofs cryptography is already implemented, we only have to extend the protocols to benefit from them.
  For example, a combination of authentication and arbitrary proof could be achieved by a combined proof of knowledge of a signed credential (current authentication system) and said arbitrary proof, although the values used in it aren't actually signed by the Issuer, the first proof gives credibility.
  Another solution for the same problem, similar to a PKI scheme, the IoT device receives a signed credential including an attribute which is his own public key. The IoT device can issue new credentials with the parameters it needs for the arbitrary proofs. The Verifier then certifies the new credential is signed by the IoT device's public key.

APPENDIX A
TEST: APDU COMMANDS EXCHANGED

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

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

The second image shows the issuance of the credential, divided in the three REST calls needed during the delegation. If we recall Idemix's issuance protocol, the User performed,

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

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

during his first step, an exponentiation with his secret key and a ZKP, which can be identified by the `COMMITMENT` and `ISSUANCE RESPONSE` commands. During the second step, the User only has to verify the Issuer's ZKP and store the credential if the signature is valid. Because the P2ABC Engine can perform Verification of ZKP without the need of secret keys stored in the smart card, the APDU Commands in this phase are only for storing the credential information inside the IoT smart card's BLOB[17] database, with the `STORE BLOB` instruction.

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

[width=0.9]gfx/UML/APDUsInitIoTSC

Fig. 12. Init IoT Smart Card APDU Commands exchanged.

[width=0.78]gfx/UML/IssuanceAPDUs

Fig. 13. Issuance APDU Commands.

[width=]gfx/UML/APDUsProving

Fig. 14. Proving APDU Commands.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[17] **B**inary **L**arge **Ob**jects