Efficient U-Prove Implementation for Anonymous Credentials on Smart Cards

Wojciech Mostowski* and Pim Vullers**

Institute for Computing and Information Sciences,
Digital Security group, Radboud University Nijmegen, The Netherlands
{woj,pim}@cs.ru.nl, http://www.ru.nl/ds/

Summary. In this paper we discuss an efficient implementation of anonymous credentials on smart cards. In general, privacy-preserving protocols are computationally intensive and require the use of advanced cryptography. Implementing such protocols for smart cards involves a trade-off between the requirements of the protocol and the capabilities of the smart card. In this context we concentrate on the implementation of Microsoft's U-Prove technology on the MULTOS smart card platform. Our implementation aims at making the smart card independent of any other resources, either computational or storage. In contrast, Microsoft suggests an alternative approach based on device-protected tokens which only uses the smart card as a security add-on. Given our very good performance results we argue that our approach should be considered in favour of Microsoft's one. Furthermore we provide a brief comparison between Java Card and MULTOS which illustrates our choice to implement this technology on the latter more flexible and low-level platform rather than the former.

Key words: anonymous credentials, smart cards, U-Prove, MULTOS, Java Card

1 Introduction

An effort to provide citizens with electronic signature (e-signature) capable identity cards is currently in progress in many European Union countries. The first countries to introduce such cards were Belgium and Estonia. More recently (November 2010) Germany introduced a new generation identity card [8] for their citizens, which also provides a limited form of anonymous attributes for improved privacy. Although Dutch identity cards already contain a chip with personal data, like in the e-passport, there is no e-signature functionality available yet. The Dutch government is currently working on adding e-signature capability, and possibly support for attributes, to such a card.

The e-signature application on the identity cards serves two major purposes. First, what is in the name, they can be used to digitally sign documents, for

* Sponsored by the NL-Net Foundation through the OV-chipkaart project.
** Sponsored by Trans Link Systems/Open Ticketing.

example tax return forms. Next, and probably most, they are used to provide strong authentication of the owner of the card, mainly for logging into governmental web services. But this use of signing or authentication certificates also involves a restriction of this use case. In the Netherlands the use of the social security number, which is integrated in the identity card, is by law only allowed within the government domain.

Therefore we study methods of authentication and authorisation which preserve the privacy of the card holder and restrict linkability of card uses. For example, the card holder may wish to prove his age category (an adult over 18 or a senior over 65) without revealing his actual date of birth. One way to achieve this is to use attributes instead of identities. A number of technologies [2, 6, 9] have been developed based on this idea, but the main focus has been on the cryptography and less on (efficient) implementations. The implementations which have been made are mainly for ordinary computers. Our research focuses on implementing and using such technologies on smart cards. This approach offers various new use cases, but also faces difficulties due to the limited capabilities of smart card platforms and hardware.

The work that we present here targets the U-Prove technology developed by Brands [6] and now owned and marketed by Microsoft [5]. Out of the existing privacy-aware protocols [5, 7, 11], this one has not yet been implemented on a smart card in its current specification. The current U-Prove specification [22] does support the *use* of a smart card as an additional protection device. In this scenario the card performs only a fraction of the protocol run. This is motivated by the constrained resources of smart cards and was already described by Brands in 2000 [6]. In Table 1 this approach is compared with our approach which offers the full protocol implementation on a smart card. We provide the full implementation of the U-Prove protocols to solve the main disadvantage of Microsoft's approach: the smart card cannot be used independently, since it is tied to computational (and storage) resources external to the card. This means that it requires a specific, card matching terminal, like the card owner's PC, to run the protocols.

Table 1. Comparison between Microsoft's *device-protected U-Prove token* approach and our *U-Prove token on a smart card* approach.

	Microsoft's approach	our approach
characteristics	add-on security measure	full protocol implementation
card stores	single device-protection attribute	all attributes, other token values
card computes	short zero-knowledge proof for the device-protection attribute	complete presentation proof
advantages	fast, lightweight, protect any number of dynamically issued tokens using pre-issued devices	independent use of the card, no need to trust the terminal
disadvantages	trusted terminal required	requires more card resources (?)

For performance our primary goal was to keep the running times of the protocol on the card sufficient for on-line use.[3] Despite the obvious efficiency concern caused by our choice to implement the full U-Prove protocols on a smart card, we managed to provide a very efficient implementation. Our worst-case execution time of the protocol on the card (with five attributes) is 0.87 seconds. Configuring the implementation for a smaller number of attributes improves this running time considerably. This makes our implementation efficient enough to be possibly considered also for the use in e-ticketing, where transactions with a card should be at or below 0.3 seconds.[4] This discards the disadvantage of our approach mentioned in Table 1, offering an overall better solution than Microsoft's approach. Thus, Microsoft is advised to change its approach to smart card support for U-Prove. Our good result is mostly due to the choice of the smart card implementation platform. Because of its more convenient API, we used a MULTOS smart card [16] in favour of the more popular Java Card platform [14]. The former has been overlooked as a prototyping platform whereas the latter exhibited questionable efficiency in some previous privacy-friendly protocol implementations [4, 25, 28].

The rest of this paper is organised as follows. Section 2 provides the necessary background on privacy-preserving protocols, related work, and open smart card platforms. We describe our MULTOS U-Prove implementation in Section 3, focusing on the implementation challenges without explaining the U-Prove protocols in detail.[5] Section 4 discusses the results of our work and compares Java Card with MULTOS. Further steps in our research on privacy-preserving protocols are presented in Section 5, and finally Section 6 concludes the paper.

2 Background

Before diving into our implementation of U-Prove we first introduce anonymous credentials and some alternatives for the U-Prove technology. Furthermore we provide some background information on smart cards and explain why we opted for the MULTOS platform instead of the more popular Java Card platform.

2.1 Anonymous Credentials

A credential is an attestation of qualification, competence, or authority issued by a third party, the *issuer*, to an individual. This individual, the *prover*, can subsequently use this credential to prove/demonstrate his qualification, competence, or authority to another party, the *verifier*. Examples of credentials are

[3] The proving scenario should be fast (less then a second) whereas the less frequently run issuance scenario can take a few seconds to complete a transaction.

[4] `http://www.smartcardalliance.org/resources/lib/Transit_Financial_Linkages_WP.pdf`

[5] A detailed description of the protocols can be found in the U-Prove cryptographic specification [5] and the mathematical background is addressed in Brands' book [6].

a membership certificate, such as a passport or employee card, or some kind of ticket to obtain some service, such as a cinema ticket or a public transport ticket. These credentials are often bound to a specific person, by means of a name and/or picture (e.g. for a passport or public transport year pass), but this is not necessarily the case (e.g. for a paper train ticket).

Anonymous credentials have the same properties as any other credentials, except that they do not reveal the identity of the prover, i.e. they provide *authorisation without identification*. In the real world this is fairly common, think of coins or public transport tickets, but in the digital world this concept is rare. This is mostly because the authenticity of a credential is usually achieved by using digital signatures which uniquely identify the issuer and prover. It is however possible to achieve anonymous digital credentials by using more advanced cryptography, as described for the first time by Chaum [13] in 1984. In the remainder of this section we will introduce a number of recent technologies which provide anonymous credentials.

U-Prove Stefan Brands provided the first integral description of the U-Prove technology in his thesis [6] in 2000, after which he founded the company Credentica in 2002 to implement and sell this technology. Microsoft acquired Credentica in 2008 and published the U-Prove protocol specification [5] in 2010 under the Open Specification Promise[6] together with open source reference software development kits (SDKs) in C# and Java.

The U-Prove technology is centred 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 (e.g. one can reveal the birth date, but not the residence address). Besides the attributes the token contains two information fields, one defined by the issuer, and one by the prover. These fields are always disclosed and can be used to provide some meta data such as a validity date of the token. Finally there is the token's public-key, which aggregates all information in the token, and a signature from the issuer over this public-key to ensure the authenticity.

A previous attempt to implement this technology on a smart card by Tews and Jacobs [28], based on Brands' description [6], resulted in a highly involved application with running times in the order of 5–10 seconds which make it not really usable in practice. Our implementation, which we describe in Section 3, not only has a much better performance but is also, except from some minimal limitations, compatible with the development kits released recently by Microsoft.

Idemix and DAA Identity mixer (Idemix) is an anonymous credential system, based on the Camenisch-Lysyanskaya anonymous credentials scheme [9, 10, 19] developed at IBM Research in Zürich that enables strong authentication and privacy at the same time. The first prototype [11] was developed in 2002 and

[6] `http://www.microsoft.com/interop/osp/`

has been improved over the years. An open source Java implementation of Idemix was released in 2010 as part of the Open Innovation Initiative.[7]

Direct anonymous attestation (DAA) [7] is a technology based on Idemix. It allows a user to convince a verifier that she uses a platform that has embedded a certified hardware module.[8] The protocol protects the user's privacy: if she talks to the same verifier twice, the verifier is not able to tell whether or not he communicates with the same user as before or with a different one.

In 2009 Bichsel et al. [4] implemented Idemix on a Java Card whereas Sterckx et al. [25] did the same for DAA. They provide the first proper implementations of anonymous credentials on smart cards. The major drawback of these implementations is the running time of several seconds which is still too much for being really practical.

Self-blindable Credentials The idea behind the self-blindable credentials by Verheul [29] is that every time a credential is used it is blinded such that two occurrences of the same credential cannot be recognised. This in contrast to the U-Prove token which is the same in each transaction, and hence serves as a pseudonym. The benefit of this approach is that the use of such credentials is untraceable. Furthermore they can be efficiently implemented on a smart card [2, 17] using elliptic curve cryptography (ECC), providing the best performance results thus far.

The drawback is that, due to the untraceable nature of this technology, incorporating revocation is hard and very costly [18]. Furthermore, ECC support on smart cards is very limited making prototyping very hard. Finally, the technology is fairly new compared to U-Prove and Idemix, and not backed by a big company which offers support for it.

German Identity Card The protocols that we have described so far are (to the best of our knowledge) the only candidates providing anonymity by design and ones that *could* be implemented on a smart card. However, we should also shortly mention an approach of the German identity card that is actually implemented and being rolled out since November 2010, where a limited form of anonymous attribute use is achieved by altering the existing ECC based electronic identity protocols by sharing private ECC keys across large batches of cards [3]. The protocol itself provides restricted access to the card by means of the so-called card verifiable certificate mechanism [8] and allows for selective disclosure of attributes, depending on the rights specified in the certificate (e.g. an alcohol shop is only authorised to check for the "over 18" attribute). Signed attributes are partly anonymous because of the sharing of the signing keys between batches of cards, such that a signature cannot be linked to a single card.

[7] `http://www.zurich.ibm.com/news/10/innovation.html`

[8] DAA has been adopted in 2004 by the Trusted Computing Group in the Trusted Platform Module specification as the method for remote authentication of a hardware module.

2.2 Smart Cards

One of the goals of our research is to assess how fast privacy-friendly protocols are when run on a smart card. Hence implementing our prototypes requires an open smart card platform that also provides the necessary cryptographic hardware support – previous research [28] clearly shows that, in terms of performance, purely software based prototypes are not sufficient for realistic use. In practice that leaves us with two possible smart card platforms, Java Card and MULTOS, described below. We motivate the use of the latter one for the work presented in this paper.

Regardless of the programming technology, all smart cards provide the same external functionality. A smart card is an embedded device that communicates with the environment through Application Protocol Data Units (APDUs) – byte arrays formatted according to the ISO7816 specification. Most notably, the APDUs constrain the communication payload to roughly 256 bytes in each direction for a single APDU exchange. The permanent storage of the card (E^2PROM memory) is considered highly secure, accessible only through the APDU commands offered by the application, which in turn are subject to any authentication and secure messaging requirements that the card application may impose.

Java Card Java Card is a now well-established smart card platform based on a tailored, cut-down version of Java (hence the name) [14]. One of the main features of Java Card is software interoperability. On top of the operating system of the card resides a Java Card virtual machine, compliant with the official specification [27], that executes Java byte code. In parallel, the platform defines the Java Card API [26] that provides the developer an interface to the hardware of the smart card. In terms of the programming and deployment of applications Java Cards are (almost) fully independent of the underlying hardware and operating system of the card. Large numbers of actual smart card products are implemented on Java Cards based on a variety of chips coming from different manufactures. Precise data on the number of deployed Java Cards or MULTOS cards are hard to find, but the Java Card Forum[9] claims there are already over a billion Java Cards in use.

The Java Card API is carefully designed to support the smart card environment and has several built-in security features. For example, it provides predefined Java classes for hardware supported cryptographic key storage (with possible internal encryption). To account for different hardware profiles of a card, parts of the Java Card API implementation are made optional. For example, one card may support both RSA and ECC in hardware and expose this functionality through the API, while another card may only support RSA, in which case all API calls related to ECC are not available and report a corresponding Java exception instead.

This brings us to the main shortcoming of the Java Card platform from our point of view. The Java Card API is predefined and *closed*. Any hardware

[9] `http://www.javacardforum.org/`

functionality that is not exposed through the imposed Java Card API, is not accessible to the developer by any other means. For example, for RSA based cryptography it is only possible to generate public and private keys of predefined RSA lengths (512, 1024 bits, etc.) and perform full RSA de-/encryption or signing with these keys according to standard protocols, such as RSA-PKCS. The more primitive operations that build up RSA operations, such as modulo prime inverse or exponentiation, are not available. Since all of the protocols that we are interested in require access to such cryptographic operations (in large modulo prime and/or EC domains), this is a practical show stopper. We are not the only ones to note this. For example, in [25] similar problems regarding the development of DAA on a Java Card are reported. Even more, an efficient implementation of the e-passport standard [8] on a Java Card also requires cryptographic routines not anticipated by the standard Java Card API. In this case, due to high demand, Java Card producers decided to enrich the Java Card API with proprietary extensions to support e-passport standards [21]. But this only solves the problem for one application type and, moreover, makes the platform non-interoperable.

MULTOS The design principles of the MULTOS platform [16, 20] are similar to those of Java Card. A hardware independent execution platform is run on top of the operating system of the card. Similarly to Java Card byte-code, a MULTOS card executes specific op-codes of the MULTOS Execution Language (MEL) and exposes smart card specific interfaces to the developers through dedicated MEL op-codes. These op-codes already provide a full and detailed API to the card's hardware. Most of the primitive operations that the hardware can possibly support are reflected in the corresponding MEL op-codes. Thus, MEL provides the full base for programming MULTOS cards, and a skilled developer can easily write programs for the card already in the MEL assembly. However, the MULTOS development tools also provide programming interfaces to C and Java. Applications in these languages are translated/compiled by the tools into MEL op-codes and can then be run on a card.

Similar to the Java Card API routines, some of the MEL op-codes are specified to be optional, mostly ones responsible for cryptographic operations. A particular MULTOS card may or may not support the optional op-codes. For our implementation we used development cards based on the SLE66 chip from Infineon. This particular MULTOS implementation [1] supports a wide range of modulo arithmetic operations, a range which is sufficient to fully support all of the U-Prove calculations. This is the main reason to choose MULTOS in the context of this work – its more low-level and flexible API as opposed to less flexible and more high level Java Card API.

Our choice is to use the MULTOS C interface to do our prototype implementation of U-Prove. For simple smart card applications the C interface seems to provide an easier programming environment than Java, and although C programming platforms are not type safe by definition (as opposed to Java), per application memory safety is guaranteed by the MULTOS platform, regardless of the high-level language used during development.

3 Implementing U-Prove for Smart Cards

U-Prove consists of two protocols. We briefly introduce these protocols here. A detailed description of the protocols and the necessary computations can be found in the U-Prove cryptographic specification [5].

During the first protocol, the issuing protocol, the U-Prove token is constructed by combining the public key of the issuer with the attributes. To authenticate this token it is signed by the issuer. However, just signing the token would allow the issuer to later recognise the resulting signed token. Therefore a blind signature scheme [12] is applied such that the issuer does not learn the exact value of the resulting signature. As a result of this protocol the prover now has a signed token containing his attributes.

The second protocol, the presenting or proving protocol, is used to present a number of attributes from the token. During this protocol the prover presents his token to a verifier together with a selection of its attributes. To verify the authenticity of the token the verifier checks the signature of the issuer. Finally the prover needs to prove that the presented attributes are actually the attributes contained in the token (and thus the signed attributes). For this purpose the prover constructs a zero-knowledge proof [15] in which he proves that he knows all the attributes contained in the token, including those not disclosed to the verifier. Due to the zero-knowledge properties of the proof the verifier does not learn anything about the attributes not disclosed to him. He is, however, able to verify, using the proof and the disclosed attributes, that the attributes actually correspond to those stored in the token.

3.1 U-Prove and Smart Cards

The use of U-Prove in combination with a smart card was already envisioned by Brands [6] and published by Microsoft in the latest release of the U-Prove cryptographic specification [22]. Their idea is to use a smart card (or even any trusted computing device) as a manner of protecting U-Prove tokens, which they then call device-protected tokens. This is achieved by having the device contribute one attribute to the token. The actual value of this attribute is, like a private key, only known by the device and will always be hidden. Therefore the device is required during the proving protocol, since a prover has to prove knowledge of *all* attributes contained in a token.

Besides adding an additional layer of protection the U-Prove technology overview [24] describes a number of other benefits gained when using device-protected tokens. For example, a device can be used to enforce dynamic policies or prevent the use of a token at a blacklisted website. It also helps to enforce non-transferability of tokens by having the prover authenticate to the device before allowing it to be used in a protocol interaction. Another option, especially interesting for smart cards, is to use the device as a carrier, or secure roaming store, for entire U-Prove tokens and not one attribute. This way the U-Prove token is always available when needed.

This last feature of a device-protected U-Prove token has one major drawback, namely one will need to trust the device that is used to perform the proving protocol. This is because the actual attribute values are used during the computation steps of this protocol. Hence the device must release all information, except its own special attribute, during a protocol run. When using a personal computer this might be acceptable, but in scenarios where the device should be used directly with a verifier, for example at a public transport gate, or at a vending machine for cigarettes, this turns out to be problematic. Since these are the areas of use which are most interesting for us, we decided to develop our own implementation of the full prover protocol specification on a smart card instead of using Microsoft's more limited approach.

3.2 U-Prove on MULTOS

A very general view of our implementation of the U-Prove technology is that it provides storage for preloaded (e.g. cryptographic domain parameters) and calculated (e.g. generated keys) values of the protocols, as well as attribute storage, and, more importantly, a sequence of hash and modulo prime arithmetic operations to execute the corresponding stages of the protocols. These arithmetic operations are the core of the performance considerations of our implementation. A few hashing operations are executed and multiple exponents over numbers in a large prime field have to be calculated during a proving protocol run. For example, the commitment a to blinding values w_i is calculated according to the following formula.

$$a = \mathcal{H}(h^{w_0} \prod_{i \in U} g_i^{w_i} \bmod p) \tag{1}$$

Here U is the set of attributes *not* to be disclosed, hence disclosing less attributes requires more exponentiation and multiplications modulo prime number p. The range of these calculations is also restricted by the limits of our MULTOS implementation platform. Namely, on our development cards we are limited to a modulus size of 1024 bits for modulo arithmetic,[10] and SHA-1 is the only built-in hashing algorithm available. Although this may sound restrictive, it also makes the choice of the U-Prove protocol configuration (protocol parameters) for our implementation easy. We have simply chosen to implement the protocols using the domain parameters fixed to the same ones as in the default configuration of the official U-Prove SDK reference implementation and official U-Prove test vectors [23], that is 1024 bits for modulus size and SHA-1 for hashing to match with the capabilities of the card.

To make the U-Prove protocol calculations efficient the smart card memory issues have be to taken into account. The first and most important aspect of developing *any* smart card application is the allocation of memory. The two rules of thumb are:

[10] The card actually supports up to 2048 bits, but then during exponentiation only small enough exponents can be used, a requirement which the U-Prove operations do not satisfy.

1. the total memory allocation should be optimised, and
2. to prevent memory exhaustion during operation there should be no dynamic memory allocation.

Furthermore, for any smart card platform the developer is usually offered a few kilobytes of RAM memory, which is normally used for fast "scratch-pad" computations and whose contents disappear on every power down (in MULTOS this is called session memory). The other kind of memory is the E^2PROM, which provides the permanent storage for the card (in MULTOS called static memory). Substantially more E^2PROM than RAM is usually available on a card, in the range of tens of kilobytes. However, it is slower than RAM, especially during writing. Moreover, on the hardware level E^2PROM is updated in block mode, hence repeated updating of single bytes of this memory (e.g. with a **for** loop) further hinders efficiency.

Considering the size of the U-Prove data that is used in the protocols and the requirements of the MULTOS cryptographic routines (all data for a cryptographic operation needs to be in one continuous array) the first thing to take care of is a careful split of the card data between E^2PROM and RAM. Only 960 *bytes* of RAM are available on our development cards, compared to 36 *kilo*bytes of E^2PROM. The most frequent use case of the card is the execution of the proving protocol, hence this is where good use of RAM is highly desirable. For that we limited the maximum number of stored attributes to 5 and then we ensured that all data participating in the proving protocol is allocated in RAM, as shown in Listing 1. After this the total RAM requirement for this protocol is 756 bytes, which just safely fits within the RAM available on the card.

Listing 1. Declaration of the variables residing in RAM.

```
#pragma melsession // These vars will sit in RAM

union {
    ... // Overlapping temporary storage for other parts of the protocol
    unsigned char array[328];
} temp_ram; // Temporary storage, 328 bytes needed in the worst case

unsigned char UD[MAX_ATTR]; // Attribute disclosure selection, 5 bytes

NUMBER_QSIZE w_i[MAX_ATTR + 1]; // w_0, ..., w_n (total 6*21 bytes)
NUMBER_QSIZE r_i[MAX_ATTR + 1]; // r_0, ..., r_n (total 6*21 bytes)

NUMBER_QSIZE a, c; // a and c values (2*21 bytes)
NUMBER_PSIZE t; // Another temporary storage (129 bytes)

#pragma melstatic // The following will sit in E^2 PROM
...
```

Listing 2. The function to compute commitment a from (1).

```
void computeCommitmentA(void) {
    ModularExponentiation(QSIZE_BYTES, PSIZE_BYTES,
          w_i[0].number, p.number, h.number, t.number);
    for(int i = 0; i < MAX_ATTR; i++) {
        if(UD[i]) continue; // i is in D, not interested
        ModularExponentiation(QSIZE_BYTES, PSIZE_BYTES,
              w_i[i+1].number, p.number, g_i[i+1].number, temp_ram.vars.a.number);
        ModularMultiplication(PSIZE_BYTES,
              t.number, temp_ram.vars.a.number, p.number);
    }

    // t now contains h^w0 ∏_{i∈U} g_i^{w_i} mod p
    int len = putNumberIntoArray(PSIZE_BYTES, t.number, temp_ram.array);

    // a = H(t) (mod q)
    SHA1(len, a.number, temp_ram.array);
    ModularReduction(QSIZE_BYTES, QSIZE_BYTES, a.number, q.number);
}
```

The initialisation and issuance protocol require more scratch-pad memory than the available RAM, hence we were forced to use E^{2}PROM there. Moreover, the issuance protocol makes use of E^{2}PROM for permanent storage of the issued U-Prove token and other permanent protocol parameters (prime numbers p, q, etc.). Because of the block mode characteristics of E^{2}PROM updates mentioned before, it is particularly important to use predefined MEL functionality for block operations (e.g. ADDN, COPYN, etc.). This way the E^{2}PROM memory is updated in block mode by the platform and execution speed can be maintained. In contrast, updating E^{2}PROM one byte at a time with a **for** loop causes dramatic performance loss – for updates of kilobytes of memory execution time is counted in seconds. The size of E^{2}PROM is not an issue – 36kB is more than sufficient to store the static data of a U-Prove token with 5 attributes each sized at the maximum of 255 bytes.

This completes the efficiency considerations for our implementation. Otherwise the implementation of the U-Prove protocols is rather straightforward in the MULTOS environment and mostly entails direct calls to the MULTOS API. An example is given in Listing 2, which computes formula (1).

3.3 Integration into the Microsoft U-Prove SDK

The previous section described the implementation of the U-Prove protocols which mainly concerns storage and the mathematical computations. This is, however, not sufficient to use it in combination with Microsoft's U-Prove SDK. We need to bridge between the high-level Java interfaces defined in this SDK and the low-level APDU interface of the smart card.

We designed the low-level APDU interface to be as simple as possible. Essentially it has to provide three types of functionality: (1) sending data to the card, (2) ask the card to perform the necessary computations, and (3) retrieve the results from the card. The second type of the interface functionality is easiest, we just defined an APDU instruction for each of the steps in the protocols. For transferring data to and from the card we restricted the values to the maximum amount of data that can be transferred in one APDU (255 bytes). This allows us to just define one APDU instruction per variable, parametrised only with the index if needed (for example g_i), for setting or getting a value.

Finally we need to bind this low level APDU API to the interfaces and data types provided by the U-Prove SDK. Luckily the SDK just uses byte arrays for the external access to the data types such that no additional conversion is needed. The only thing that needs to be done for a data type, for example IssuerParameters, is that the setter and getter have to be divided into the individual APDU instructions, for example the setPublicKey and setEncodingBytes instructions.

All this functionality has been combined into a single Java class which provides setters and getters for the data stored on the card as well as methods for the protocol steps. Using the Java built-in smart card library it serves as an interface between our MULTOS implementation and the Microsoft SDK.

4 Results and Performance Analysis

The two most important factors for us to test in our U-Prove implementation were correctness of the protocol calculations (obviously) and the speed. Testing the correctness was fairly easy. Since we interfaced our card to Microsoft's U-Prove SDK we could simply test it by invoking the protocol runs from the SDK and check the results. During the first stages of the development partial protocol calculations were verified with the test vectors provided with the U-Prove SDK [23]. In the whole process a few corner case problems with our calculations surfaced that required minor corrections.

As we stated in the previous section, for speed we concentrated our implementation efforts on the every day use case of the application, i.e. the attribute proving protocol. However, we also strived to optimise the rest of the protocols to maintain speed also during the initialisation and issuance parts. For the performance analysis, we executed a number of full protocol runs (initialisation, issuance, proving) on the card in various configurations. First of all we varied the number of stored attributes on the card, then within this attribute range we varied the number of (un)disclosed attributes. As shown in Figure 1 this resulted in a running time of 3.6 and 5.5 seconds for the issuance of a U-Prove token with respectively 2 and 5 attributes. The dark grey area on the graph indicates the core running time of the protocol calculations on the card, whereas light grey indicates the remaining overhead. This overhead consists of transferring data to the card and communicating the results of the protocol run between the card and PC.

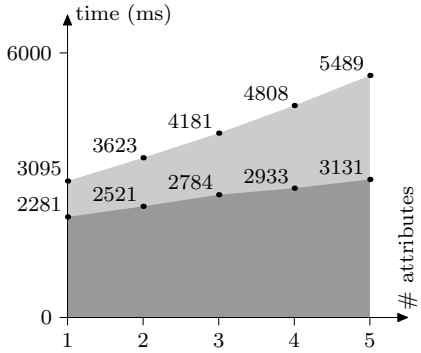

Fig. 1. U-Prove token issuance times (: computation, : overhead).

Correspondingly, the cumulative results for the attribute proving protocol are shown in Figure 2. What can be seen in these graphs is that under "full load" our implementation executes the complete proving protocol in just under 0.9 seconds (graph 2(b)). In this worst-case scenario 5 attributes are stored on the card, none of which are disclosed during the protocol run. In other words, the U-Prove token is only validated for its authenticity without revealing any attribute data. Such a scenario is not very likely to occur in reality. In a more likely scenario at least one or two attributes are going to be disclosed and we can also assume that a U-Prove token will contain less attributes (or, that a large number of attributes can be split into several separate U-Prove tokens). As the graphs show, reducing the number of stored attributes improves the running time at a rate of 100 milliseconds per attribute, and also that the performance increases along with increasing the number of disclosed attributes, roughly 50 milliseconds per each extra disclosed attribute. Overall, this brings the total execution time for a two attribute token disclosing one attribute to under 0.5 seconds (graph 2(a)).

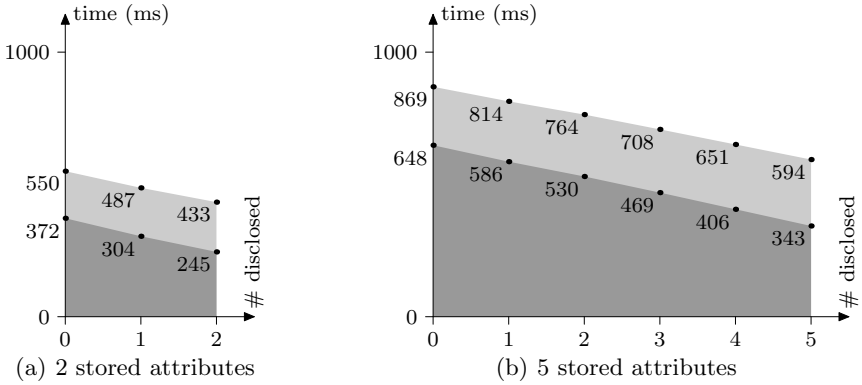

Fig. 2. Attribute proving times for different configurations (/: same as in Fig. 1).

One of the reasons to justify the Microsoft's device protected approach as described in Section 3.1 are possible resource issues with smart cards (limited storage space and limited speed). Our performance results undermine this argument. The worst case execution time of the proving part is 0.87 seconds. This not only makes the card implementation fast enough to be usable in general, it also makes it usable for "field" applications, e.g. dispensing machines. Even more, for smaller numbers of smaller attributes the running times become almost acceptable for use in public transport/e-ticketing, where the commonly required card transaction times should stay below 0.3 seconds. We also see a potential to improve the running times using faster smart card hardware, we elaborate on this in the upcoming section. Overall, these good results strongly justify the idea to use U-Prove standalone on a smart card rather than to use Microsoft's device-protected token approach, which now has no obvious functional or performance advantages over our approach.

Furthermore, excluding our own previous work on implementing ECC-based self-blindable signatures on a smart card [2, 17] our performance results are by far better than all the previously reported results for anonymous credentials implemented on smart cards. One of the first attempts within our group to implement a U-Prove like protocol on a Java Card [28] resulted in running times closing to 10 seconds for a setup closely corresponding to ours. The DAA protocol was also implemented on a Java Card by Sterckx et al. [25] with the running times of close to 4 seconds for the DAA signing protocol. In [4] yet another implementation of anonymous credentials on a Java Card is reported with running times of around 7–10 seconds. Our MULTOS U-Prove implementation is simply way faster.

The only limitations of our implementation are imposed by the limited resources of the MULTOS smart card. We had to limit the prime modulus size to 1024 bits, use only SHA-1 hashing, and because of the available RAM (<1kB) on the card we could only allow for the maximum of 5 attributes, each one up to 255 bytes in size. Otherwise our implementation is fully flexible and provides full U-Prove functionality, *including* the smart card features described in Section 3.1. However, it is not uncommon for modern smart cards to support up to 2048 bits for modulus size and 2 kilobytes of RAM, only no such MULTOS cards were available to us. In the following we make some speculative performance estimations based on tests performed with Java Cards that we have.

4.1 MULTOS vs. Java Card

As we already stated in Section 2.2 providing an efficient implementation of U-Prove on a Java Card is currently not possible, mainly because of the inflexible Java Card API. However, we can use Java Card to do further (speculative) performance analysis.

Our Java Cards are implemented on the SmartMX hardware platform from NXP, which provides excellent hardware cryptographic support (2048 bit RSA and 320 bit ECC), and is considered state of the art when it comes to speed.

By running comparative speed tests between our Infineon SLE66 MULTOS card and a brand new NXP SmartMX (JCOP31) Java Card we estimate two things:

1. How fast a sibling implementation, equal in terms of the supported protocol parameters, would be on the SmartMX chip?
2. How fast would an implementation supporting greater modulus size and more attributes would be on the SmartMX chip?

For this we simply compared the speed of raw SHA-1 and RSA operations between the two platforms, operating both on RAM and E^2PROM. The results are shown in Table 2, the running times are expressed in milliseconds for 100 iterations of each test, for example a single SHA-1 execution storing the results in RAM for the first case (MULTOS card on a contact interface) takes 51.2 milliseconds on average. More generally and roughly speaking, the JCOP31 card is 4 times faster for SHA-1, and 1.3 to 1.5 times faster for RSA-1024, depending on the target memory. Although exact estimations are not possible, we speculate that the attribute proving part with the same protocol parameters as our MULTOS implementation could be improved by a factor of 2 making the worst execution time for 5 attributes stay below 0.5 seconds. We also estimate that for the 2 attribute configuration the running times would drop below the 0.3 seconds required for public transport and e-ticketing applications. As for the implementation supporting larger modulus size and more attributes, the JCOP31 card drops its performance going from RSA-1024 to RSA-2048 by the factor of 2 to 2.5. Based on this we believe that the proving part of the protocol would be within 2 seconds realm for 2048 bit modulus size and 10 attributes. This would still be faster than any of the existing Java Card anonymous credentials implementations that only support modulus sizes smaller than 2048 with reasonable efficiency.

Yet again this stresses the Java Card shortcoming of the limited hardware interface provided by the API – had the API been more flexible, our speculative figures above would probably be factual. Although this issue has been brought up before and we know that the smart card industry is very well aware of this problem, we see hardly any improvements in this respect. The MULTOS plat-

Table 2. Performance comparison between *MULTOS on an Infineon SLE66* chip and *JCOP31 on a NXP SmartMX* chip (time in milliseconds for 100 successive operations).

	MULTOS		JCOP31	
	contact	wireless	contact	wireless
SHA-1 RAM	5120	5274	1110	1136
SHA-1 E^2PROM	6125	6308	1442	1466
RSA-1024 RAM	1016	1060	772	777
RSA-1024 E^2PROM	2936	3041	1941	1952
RSA-2048 RAM	14289	14898	1926	1950
RSA-2048 E^2PROM	17237	17956	3838	3865

form proved itself very strong here with its flexible API design. What MULTOS is lacking from our point of view is wider hardware support for cryptography other than RSA and DES. In our own privacy-friendly protocol designs we rely heavily on ECC, and although the MULTOS API specification supports ECC, no MULTOS cards with hardware ECC support are currently available to us for small scale development. Finally, we find the size of the RAM (960 bytes) available on the MULTOS development cards a little bit of a limiting factor to fully commit to MULTOS as our prototyping framework.

5 Ongoing Research

In our research we continue to look for efficient solutions for privacy-friendly smart card applications. For this we develop our own protocols as well as explore the existing ones. Both require prototypes for feasibility and efficiency analysis. One of the *by-products* of the work presented in this paper is the discovery of the MULTOS cards as an efficient implementation platform for this kind of protocols.

Hence, the obvious next step is to investigate the implementation of the Idemix protocol suite on a MULTOS card. Idemix has been already implemented on a Java Card [4] and despite the best effort of the implementers to maintain reasonable efficiency the running times still leave room for improvement in our opinion. Idemix has more features and is more complex than U-Prove and more involved computations are required, so clearly we do not expect an equally fast implementation as the one we just presented, but we certainly believe we can considerably improve over the current Java Card Idemix implementation.

In [2, 17] we presented an efficient Java Card implementation of our own protocol based on ECC and self-blindable signatures. This protocol provides a very strong anonymity property, however our implementation, despite the achieved efficiency, still suffers from the inability to fully utilise the hardware capabilities of the card hidden beyond the Java Card API. Here, a MULTOS card with full ECC support would provide further improvement possibilities. When (if at all) such cards are available to us we will certainly investigate these possibilities.

In parallel to this protocol and speed quest we also develop case studies and a demo suite for on-line use of anonymous credentials. To this end, we are implementing a general framework in the form of a browser plug-in for smart card enabled web services. This framework will be targeted for the set of anonymity friendly protocols under our consideration and will allow us to do more practical comparative studies between the different anonymous credential approaches exemplified by suitable demos.

6 Conclusion

We have presented an efficient MULTOS implementation of the U-Prove technology that allows to run the complete prover side of the protocols on a smart card.

This provides an anonymity friendly credentials mechanism for users of such a smart card, with full independence from authentication resources external to the smart card. From the user perspective, the most performance sensitive part of the protocol is attribute proving. Here, the achieved worst-case running times of 0.87 seconds for the whole set of attributes clearly establishes the practical usability of our implementation. Our performance results also strongly support our idea to use a stand-alone U-Prove smart card rather than the Microsoft device-protection approach, which seems to overlook the current capabilities of smart cards. One other thing that seems to be overlooked by scientists and smart card developers is the existence of the MULTOS smart card platform. During our work it proved itself highly flexible and reasonably fast, hence our next steps are to implement and assess the performance of other anonymity friendly protocols, primarily Idemix, in a (MULTOS) smart card setting.

Acknowledgements We are grateful to Jaap-Henk Hoepman, Bart Jacobs, Christian Paquin, Erik Poll and the anonymous reviewers for their valuable comments which helped to improve this work.

References

1. MULTOS implementation report. Tech. Rep. MAO-DOC-TEC-010 v1.36a, MAOSCO Limited (February 2010)
2. Batina, L., Hoepman, J.H., Jacobs, B., Mostowski, W., Vullers, P.: Developing efficient blinded attribute certificates on smart cards via pairings. In: Gollmann, D., Lanet, J.L. (eds.) Smart Card Research and Advanced Applications – CARDIS 2010. LNCS, vol. 6035, pp. 209–222. Springer-Verlag (April 2010)
3. Bender, J., Kügler, D., Margraf, M., Naumann, I.: Privacy-friendly revocation management without unique chip identifiers for the German national ID card. Computer Fraud & Security (September 2010)
4. Bichsel, P., Camenisch, J., Groß, T., Shoup, V.: Anonymous credentials on a standard Java Card. In: Computer and Communications Security – CCS 2009. pp. 600–610. ACM (November 2009)
5. Brands, S., Paquin, C.: U-Prove cryptographic specification v1.0. Tech. rep., Microsoft Corporation (March 2010)
6. Brands, S.A.: Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy. MIT Press (August 2000)
7. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Pfitzmann, B., Liu, P. (eds.) Computer and Communications Security – CCS 2004. pp. 132–145. ACM (October 2004)
8. Bundesamt für Sicherheit in der Informationstechnik: Advanced security mechanisms for machine readable travel documents, Version 2.05. Tech. Rep. TR-03110, German Federal Office for Information Security (BSI), Bonn, Germany (2010)
9. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In: Advances in Cryptology – EUROCRYPT 2001. pp. 93–118. Springer-Verlag (May 2001)
10. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) Advances in Cryptology – CRYPTO 2002. LNCS, vol. 2442, pp. 101–120. Springer-Verlag (August 2002)

11. Camenisch, J., Van Herreweghen, E.: Design and implementation of the idemix anonymous credential system. In: Computer and Communications Security – CCS 2002. pp. 21–30. ACM (November 2002)
12. Chaum, D.: Blind signatures for untraceable payments. In: Chaum, D., Rivest, R.L. (eds.) Advances in Cryptology – CRYPTO 1982. pp. 199–203. Plemum Publishing (1983)
13. Chaum, D.: Security without identification: transaction systems to make big brother obsolete. Communications of the ACM 28, 1030–1044 (October 1985)
14. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Java Series, Addison-Wesley (June 2000)
15. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A. (ed.) Advances in Cryptology – CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer-Verlag (1987)
16. France-Massey, T.: MULTOS – the high security smart card OS. Tech. rep., MAOSCO Limited (September 2005)
17. Hoepman, J.H., Jacobs, B., Vullers, P.: Privacy and security issues in e-ticketing – Optimisation of smart card-based attribute-proving. In: Cortier, V., Ryan, M., Shmatikov, V. (eds.) Foundations of Security and Privacy – FCS-PrivMod 2010 (July 2010), (informal)
18. Hoepman, J.H., Lueks, W., Vullers, P.: Revoking self-blindable credentials (2011)
19. Lysyanskaya, A.A.: Signature schemes and applications to cryptographic protocol design. Ph.D. thesis, Massachusetts Institute of Technology (September 2002)
20. MAOSCO Limited: MULTOS Developer's Reference Manual (October 2009)
21. NXP Semiconductors: Smart solutions for smart services (z-card 2009). NXP Literature, Document 75016728 (2009)
22. Paquin, C.: U-Prove cryptographic specification v1.1. Tech. rep., Microsoft Corporation (February 2011)
23. Paquin, C.: U-Prove cryptographic test vectors v1.1. Tech. rep., Microsoft Corporation (February 2011)
24. Paquin, C.: U-Prove technology overview v1.1. Tech. rep., Microsoft Corporation (February 2011)
25. Sterckx, M., Gierlichs, B., Preneel, B., Verbauwhede, I.: Efficient implementation of anonymous credentials on Java Card smart cards. In: Information Forensics and Security – WIFS 2009. pp. 106–110. IEEE (September 2009)
26. Sun Microsystems, Inc.: Java Card 2.2.2 Application Programming Interface Specification (March 2006)
27. Sun Microsystems, Inc.: Java Card 2.2.2 Virtual Machine Specification (March 2006)
28. Tews, H., Jacobs, B.: Performance issues of selective disclosure and blinded issuing protocols on Java Card. In: Markowitch, O., Bilas, A., Hoepman, J.H., Mitchell, C., Quisquater, J.J. (eds.) Information Security Theory and Practice – WISTP 2009. LNCS, vol. 5746, pp. 95–111. Springer-Verlag (September 2009)
29. Verheul, E.R.: Self-blindable credential certificates from the Weil pairing. In: Boyd, C. (ed.) Advances in Cryptology – ASIACRYPT 2001. LNCS, vol. 2248, pp. 533–550. Springer-Verlag (December 2001)