Mobile Application to Java Card Applet Communication using a Password-authenticated Secure Channel

Michael Hölzl
JRC u'smile and Johannes Kepler University Linz, Institute of Networks and Security
Altenbergerstraße 69, A-4040 Linz, Austria
michael.hoelzl@usmile.at

Endalkachew Asnake
JRC u'smile
University of Applied Sciences Upper Austria
Softwarepark 11, A-4232 Hagenberg, Austria
endalkachew.asnake@usmile.at

René Mayrhofer
JRC u'smile and Johannes Kepler University Linz, Institute of Networks and Security
Altenbergerstraße 69, A-4040 Linz, Austria
rene.mayrhofer@usmile.at

Michael Roland
JRC u'smile
University of Applied Sciences Upper Austria
Softwarepark 11, A-4232 Hagenberg, Austria
michael.roland@usmile.at

ABSTRACT

With the increasing popularity of security and privacy sensitive systems on mobile devices, such as mobile banking, mobile credit cards, mobile ticketing, or mobile digital identities, challenges for the protection of personal and security sensitive data of these use cases emerged. A common approach for the protection of sensitive data is to use additional hardware such as smart cards or secure elements. The communication between such dedicated hardware and back-end management systems uses strong cryptography. However, the data transfer between applications on the mobile device and so-called applets on the dedicated hardware is often either unencrypted (and interceptable by malicious software) or encrypted with static keys stored in applications. To address this issue we present a solution for fine-grained secure application-to-applet communication based on *Secure Remote Password* (SRP-6a), an authenticated key agreement protocol, with a user-provided password at run-time. By exploiting the Java Card cryptographic API and minor adaptations to the protocol, which do not affect the security, we were able to implement this scheme on Java Cards with reasonable computation time.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Cryptographic controls, Information flow controls.

General Terms

Security, Verification

Keywords

Java Card, smart card, SRP-6a, secure channel, secure element, mobile devices

MoMM '14 December 08 - 10 2014, Kaohsiung, Taiwan

ACM 978-1-4503-3008-4/14/12 ...$15.00.
http://dx.doi.org/10.1145/2684103.2684128.

1. INTRODUCTION

Nowadays, mobile devices such as mobile phones, tablets or smart watches have become an indispensable part of our daily life, but are encountering many security threats [24, 25, 29]. They can not only be easily stolen or lost, but also face attacks from malicious third-party applications. We also have to be aware that the flash memory on these mobile devices can not be trusted and unauthorized individuals have the possibility to gain personal, business, or other security and privacy sensitive data from that memory. To provide a secure environment for storing data, applications often use an additional tamper resistant hardware, like smart cards or secure elements. Special variants of them run with Java Card technology, which allows the execution of small Java-based applications (applets). One advantage of these tamper resistant units is the protection of data and executed code against various physical and software attacks. Communication between the card and an off-card application is only possible through a standardized interface for exchanging so-called *Application Protocol Data Units* (APDU). There are many applications that take advantage of this tamper resistance by storing master passwords on the hardware [28, 32]. However, when using smart cards for applications, it is important to consider that the communication outside the applet is not automatically secured. This is especially relevant in an environment where malicious third-party applications could eavesdrop on the data transfer.

The main motivating goal in this paper is to provide an infrastructure for third-party applications on mobile devices to securely communicate through a password-authenticated secure channel with applets running on the Java Card. In an infrastructure where applications running on the main processor also have a corresponding applet running on the tamper resistant environment, they could then use this additional hardware module to securely store security and privacy sensitive data. A mobile web browser could, for example, use it to store passwords. Or, company services could use it for storing private keys for corporate VPN access. The advantage of additional hardware is that malware running on the device would not have the possibility to directly read this password storage. Even if these passwords were encrypted on the flash memory using a master password, an attacker would still be able to brute-force this master password us-

ing an offline dictionary attack. Although there are ways to make brute-force attacks on encrypted user-chosen passwords harder (e.g. using a key-derivation function) this remains problematic as users tend to use weak and easily guessable passwords on mobile devices [6, 26].

There are existing protocols for current smart cards that provide secure communication between the card and an off-card application. The respective standards are defined by GlobalPlatform (GP)[1]. However, these standards only provide a secure interface for managing applications (installation, removal, etc.) and card-specific data (e.g. personalization of applications with user-specific data). A secure channel is established between a back-end server and a security domain on the card using a shared secret. Consequently, using this secure management channel from within a mobile device application to communicate with the applets would require that shared secret to be stored within the application data. An attacker who gained access to that shared key could therefore access and manipulate that security domain with all contained applets and data. To protect the data of each application, a more fine-grained secure channel for application-to-applet transmission is needed [22]. In this paper we address this issue by proposing an implementation of a fine-grained secure channel that is authenticated with a user-provided PIN or password, compatible to current Java Cards and time efficient for the user.

For the implementation of this channel we use the *Secure Remote Password* protocol in its latest revision (SRP-6a) [37, 38]. SRP is a password-authenticated key agreement protocol which is based to the Diffie-Hellman key exchange and can be either constructed from or reduced to Diffie-Hellman [12, 37]. The main extensionis that a Diffie-Hellman key exchange is not authenticated while SRP can be used for password-based mutual authentication over insecure communication channels. Besides SRP, there are multiple other similar password-authenticated key agreement (PAKE) protocols such as SPEKE [23], J-PAKE [20], EKE [5], OKE [27], AuthA [4], and others. There are also PAKE variants which make use of elliptic curve cryptography (ECC). While it has been shown that elliptic curve cryptography is faster than other public key algorithms [16, 18], the usage of such protocols is restricted on Java Cards due to the limitations in terms of ECC support. In the current Java Card 3.0, it is still not possible to perform ECC primitives (such as point addition or multiplication). Although there are smart card manufacturers that add proprietary classes to support such functionality, using them would limit the interoperability of an implementation. Finally, we chose SRP because it can be used without any licensing, has no known flaws in its current implementation and is already included in other cryptographic protocols such as TLS [35]. However, one disadvantage of this protocol is the high complexity of computations due to modulo operations on big numbers. Therefore, a pure software implementation of this protocol on a Java Card with severe restrictions such as an 8-bit CPU and little memory is very time-consuming and not practical. In this paper we present a solution on how to execute SRP on low-end hardware platforms within a reasonable authentication time and an overall protocol runtime below four seconds. In our terms, we define reasonable authentication time as the time a user is willing to wait for a secure channel to be authenticated after password entry. Based on the Nielsen Norman Group, we assume that this value is below 1 second[2]. The key points and main contributions of our approach are:

- Implementation of a password-authenticated secure channel protocol on Java Cards, which is suitable for comparably weak passwords and does not require to store credentials in the mobile device flash memory, by exploiting the RSA public key encryption operation for a significant increase in computation performance.
- Minor adaptations to the SRP protocol scheme to optimize verification time after PIN/password entry.
- Memory optimizations to reduce required transient and persistent memory during protocol execution.
- An open source implementation for developers.

2. RELATED WORK

There have been multiple previous publications in the area of providing an authenticated channel for Java Card applets. From an industry standardization point of view, the *GP Secure Channel Protocols* (SCP) are one of the most relevant ones. The GP specification defines standards for secure channel protocols, namely SCP01, SCP02 and SCP03, to establish secure communication between a Java Card and an off-card application. According to GP Card Specification 2.2 [17], there are two ways in which an applet could handle a secure channel, namely *Direct Handling* and *Indirect Handling*. In Direct Handling, an applet is fully responsible for implementing the protocol and defining its own security domain. The other approach is *Indirect Handling* where the applet uses ready made services provided by security domains to handle the SCP. This enables the applet to be implemented independently from the protocol and leaves secure channel related computation to the security domain it belongs to.

One of the main advantages in using a GP SCP for secure application-to-applet communication is that it is an industry standard and a widely-used protocol with API support on Java Cards. For our use cases we consider the off-card environment as potentially insecure while we trust in the security of the Java Card. Since GP SCP authentication relies on static shared keys between communicating parties, the insecurity of the off-card application breaks the security of the protocol. These limitations force us to look into other password-based authentication schemes for a more fine-grained secure channel protocol.

Various previous papers have been published that make use of the Java Card crypto API to execute modulo operations on the card's cryptographic co-processor for an efficient implementation of different protocols. Sterckx et al. [34] discuss simplified methods to use *Direct Anonymous Attestation* (DAA) [9] on Java Cards. As many other cryptographic protocols, DAA also involves computation of modulo operations on big numbers. In their paper they show that these modulo operations can be computed more efficiently with the help of the cryptographic co-processor compared to a pure software implementation. Tews et al. [36] show different RSA variants and performance measurements of Brands' protocols for zero-knowledge proof [8]. This paper also proposes

[1]http://www.globalplatform.org/

[2]http://www.nngroup.com/articles/response-times-3-important-limits/

an efficient implementation of a protocol by exploiting the Java Card crypto API to perform modulo operations on the cryptographic co-processor.

Our implementation of big numbers and modular arithmetic in the Java Card applet is based on this previous scientific work.

3. THREAT MODEL

Our approach for implementing SRP on smart cards is based on the Android platform due to availability of open source projects which enable access to smart cards and secure elements. In addition to this, the openness of the platform makes it easier to analyse different types of attacks. Hence we use the Android Operating System (OS) to specify our threat model categories (Fig. 1 gives an overview of these categories in a mobile device context):

1. **Channel attacks.** The Android OS has built-in security features which protect applications against different types of threats. One of these features is application sandboxing. Every Android application has a unique user ID that is assigned at install time. At run-time, an application runs in a sandbox where communication to other applications is made possible via Inter Process Communication (IPC) facilities. One IPC facility used in our implementation is a service. Android services are application components that are used to handle long running tasks in the background. They also provide a client-server interface which can be used by different applications to request specific functions of a service. One of the possible channel attacks on Android applications which make use of IPC facilities is *service hijacking*. This happens when an application creates a connection with a malicious service instead of the intended one [10]. With such kind of possibilities, an attacker can gain full control over the message exchange between an application and Java Card applets, which creates a suitable condition for active channel attacks.

2. **Attack on application level.** This sort of attack can come from mobile applications as well as over NFC:

   (a) Malicious applications: We consider two different threats from malicious applications: (i) the first threat can be caused by an application that has been granted root privileges or that uses privilege escalation exploits [21]. Such kind of malware can access application private storage and conduct different attacks on the content (e.g. brute-force attack on encrypted passwords). (ii) in the second threat, the malware does not need root privileges, but makes use of smart card services (cf. [31]). If a malicious application has sufficient privileges to access smart card services, it can impersonate a legitimate application and try to brute-force the password to establish a channel to the applet. With methods discussed in this paper, the first threat is avoided while the second should, in the worst case, end up with denial of service.

   (b) Attacks from an external card reader: Usually, smart cards and secure elements can be accessed from an external card reader through either the contact or contact-less interfaces. An attacker who has physical access to the card and the mobile device or an attacker who is able to communicate with the smart card over *Near Field Communication* (NFC) could launch the same attacks as a malware mobile application. Additionally, we also have to consider attack scenarios over the contactless interface (e.g. relay attacks over NFC [19]).

   (c) User interface (UI) masquerading or control by malicious entity: Mobile malware that gains control over the UI (i.e. finds methods to listen to users' input), or is faking the UI, can steal users' passwords, which could result in severe consequences. This is especially problematic when the user is not aware of the compromised password. These threats are out of scope of this paper but could potentially be addressed by using a trusted execution environment (TEE) for secure password input (e.g. TrustZone[3]). Hence, the password for establishing a secure channel cannot be stolen by a malicious application. However, the methods discussed in this paper would still help to secure the data path between application and applet after password entry.

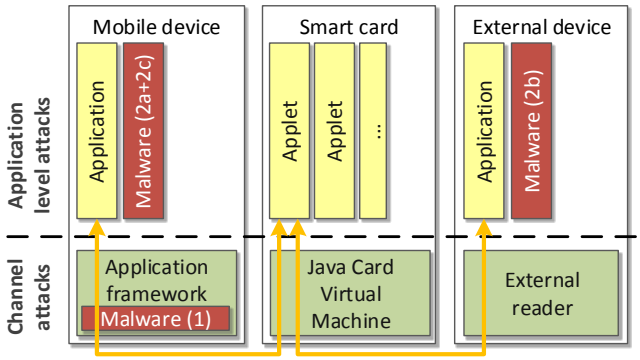


Figure 1: Overview of our threat model. The arrows depict the secure channel between application and applet.

4. SECURE REMOTE PASSWORD (SRP)

The SRP protocol was introduced by Wu [37] in 1998 and is a password-authenticated key agreement protocol that can be used over insecure channels for providing password-based secure key agreement and authentication. Similar to Diffie-Hellman key exchange [12], an eavesdropper is not able to guess the computed session key even with the knowledge of the complete data transfer. The big advantage of SRP in comparison to Diffie-Hellman is that SRP provides password-based mutual authentication. Additionally, the properties of SRP make the protocol resistant against most prominent attacks (e.g. off-line dictionary attacks, replay attacks, etc.)

4.1 SRP Key Exchange Procedure

Scheme 1 describes the steps of the original SRP protocol. Communication between both parties is generally separated into two phases: (i) a key agreement phase where both parties calculate the shared secret (steps 1-5 in Scheme 1) and (ii) a mutual verification phase where they authenticate each other

[3]TrustZone white paper at: http://www.arm.com/products/processors/technologies/trustzone/index.php

	Client		Server
1.		$\xrightarrow{I}$	(lookup s, v)
2.	$x = H(s, I, P)$	$\xleftarrow{s}$	
3.	$A = g^a$	$\xrightarrow{A}$	$B = kv + g^b$
4.	$u = H(A, B)$	$\xleftarrow{B}$	$u = H(A, B)$
5.	$S = (B - kg^x)^{a+ux}$		$S = (Av^u)^b$
6.	$M_1 = H(A, B, S)$	$\xrightarrow{M_1}$	(verify M_1)
7.	(verify M_2)	$\xleftarrow{M_2}$	$M_2 = H(A, M_1, S)$
8.	$K = H(S)$		$K = H(S)$

Scheme 1: Original SRP-6a protocol scheme as described by Wu in [38].

Table 1: SRP protocol notation [37].

g, n	The generator g and a large prime modulo number n for all computations
s	A random user's salt for the password
I, P	Identifier and password of the user
k	Constant multiplier, computed from the hash of the modulo and concatenated with g
x	Private key derived from identifier, password and salt
v	The password verifier calculated from g^x
u	Random scrambling parameter, publicly revealed
a, b	Ephemeral private keys, generated randomly and not publicly revealed
A, B	Corresponding public keys
$H(\cdot)$	One-way hash function
K	Computed session key

(steps 6-8 in Scheme 1). For the key agreement phase, public keys A and B are calculated by modular exponentiation with a private key exponent a and b. On the server-side, the public part g^b is additionally XORed with the password verifier v multiplied by constant k [38] (steps 1-4).

After exchanging public parameters A and B, the client computes the shared secret S based on the user identification (identifier I and password P), while the server does the same using the pre-computed verifier v (step 4 in Scheme 1). Based on this shared secret both parties have to mutually authenticate each other in the verification phase. This is done by exchanging the verifiers M_1 and M_2 with the corresponding opposite party (steps 6-7). Then both parties ensure that legitimate values of the client password x and the server verifier v are used in the key agreement phase. After this verification, they compute the same session key $K = H(S)$ (step 8) and use this as basis for securing future data transfers against eavesdroppers or active attackers. A list of all SRP protocol notations can be found in Table 1.

4.2 Proposed Protocol Adaptations for Java Card Applets

To improve computation time and memory usage of the protocol on a Java Card, we made minor changes to the SRP protocol (changes are highlighted in bold font in our protocol Scheme 2). First, we combine the transmission of the server's public key B with the user's salt s. The current revision of

	Client		Server
1.	$A = g^a$	$\xrightarrow{A}$	$B = kv + g^b$
2.	$u = H(A, B)$	$\xleftarrow{\boldsymbol{B,s}}$	$u = H(A, B)$
3.	$x = H(s, P)$		
4.	$S = (B - kg^x)^{a+ux}$		$S = (Av^u)^b$
5.	$\boldsymbol{K = H(S)}$		$\boldsymbol{K = H(S)}$
6.	$M_1 = H(\boldsymbol{u}, S)$	$\xrightarrow{M_1}$	(verify M_1)
7.	(verify M_2)	$\xleftarrow{M_2}$	$M_2 = H(\boldsymbol{u}, M_1, S)$

Scheme 2: Our protocol implementation based on the current revision of the SRP protocol [38] with minor changes to improve memory consumption and performance on the Java Card (server-side). Differences to the original SRP protocol are highlighted in bold font.

SRP-6a uses an additional round of transmission to agree on the identifier I and the salt s at the beginning [38]. In our elaborated use case of fine-granular application-to-applet communication, we only need one secure channel instance for each applet. Therefore, we do not need an identifier as we only have one verifier v – which is computed during applet installation – and can directly start the communication by sending the public key A to the Java Card (see steps 1 and 2 in Scheme 2). This reduces the amount of required round trips from 3 to 2 (a similar approach was also suggested as an optimized version in the original protocol publication, cf. [37]).

The second adaptation influences the sequence of calculating the session key K. Usually, this key is calculated after the mutual verification phase (see step 8 in the original SRP protocol Scheme 1). To reduce the time required for the verification (step 6 and 7 in Scheme 2), we moved the calculation of K to the key agreement phase (steps 1 to 5 in Scheme 2). On the server-side (the smart card), this first key agreement phase does not require any authentication of the user and can therefore be done before or while the user is typing password or PIN (or while running other authentication mechanisms like fingerprint, face unlock [15], special touch patterns [11], etc.) On the client-side, steps 3 to 5 are performed after the password has been entered. However, the computation time of these operations is negligible on a high performance mobile device processor. The actual time a user has to wait for the server to establish a secure channel is therefore reduced to the key verification phase in steps 6 and 7. So in the use case of a password manager on the tamper resistant hardware, the first data exchange starts when the application is opened. While the user then enters the password, the more computationally intensive operations of the key agreement phase can be executed simultaneously on the server-side. After the user enters the password, the server only requires the verifier M_1 to verify the secure channel and give access to the password manager.

The third adjustment is related to the computation of verifiers M_1 and M_2. In the original protocol these verifiers were computed as $M_1 = H(A, B, S)$ and $M_2 = H(A, M_1, S)$. However, the authors of the protocol suggest this approach as only one possible way to mutually verify client and server. In our approach we suggest another efficient way of computing the verifiers. Instead of using the public keys A and B for

computing M_1 and M_2, we use the scrambling parameter u. This change optimizes the time of execution for the verification steps on the Java Card side. For example, using SHA-256 as a hash function and a modulus N of 2048 bit, the verification of M_1 will require computing SHA-256 over a 768 bytes input. This is reduced to 288 bytes by replacing (A, B) with (u) and reduces the number of intermediate operations required to compute M_1 and M_2.

4.3 Security Analysis

From a security point of view we argue that these changes do not affect the security of the verification and the key agreement. The first two adaptations change the sequence without actually changing the communication and computations from the original SRP-6a scheme.

For the third proposed adaptation (the change in the verification) we argue that the security of the protocol is not affected as other proven protocols use a similar approach of double hashing. The most famous one is *Hash-based Message Authentication Code* (HMAC), which also makes use of a hash inside a hash function [2].

5. SECURE SESSION

In any symmetric key cryptographic system the first step is an agreement on a secure session key between communicating parties. This session key is used to establish a confidential and authentic channel between the parties. The standard approach in providing confidentiality and authenticity is to use two different algorithms for each purpose with an authenticated encryption scheme known as Encrypt-then-MAC. Recently, different algorithms have been standardized to provide both functionalities at the same time. A well-known example for this would be *Galois/Counter Mode* (GCM) from ISO/IEC 19772:2009. However, such algorithms or operation modes are not yet included in the Java Card standard.

For a secure session between the card and the off-card application, the smart card standard provides a protocol named *Secure Messaging* (SM) which is defined in ISO/IEC 7816-4. ISO/IEC 7816-4 is a standard for inter-industry message exchange between smart cards and external interfaces that also defines the APDU message structure. APDU messages are exchanged in command-response pairs which have a header and an optional body part. The SM standard defines a separate BER-TLV[4] coded format for encoding command and response APDUs for secure transmission. It also states types of algorithms to be used for confidentiality and authentication. The choice of specific algorithms and parameters are implementation dependent.

As pointed out in the introduction section, the GP secure channel is managed by a security domain of an applet. In order to establish application-to-applet level secure messaging, the protocol should be implemented on applet level. We implement the ISO/IEC 7816-4 secure messaging based on ETSI TS 102 176-2. This specification defines a set of algorithms and protocols for constructing secure channels between off-card applications and signature creating devices. The secure messaging construction provided in this document is ISO/IEC 7816-4 compliant. The changes we make for our implementation are related to using newer versions of algorithms than used in that specification.

[4]Standard for *Tag-Length-Value* encoded data structures

5.1 Confidentiality

According to ISO/IEC 7816-4 confidential command and response APDUs are exchanged using specific BER-TLV data objects protected by a suitable encryption algorithm. ETSI TS 102 176-2 specifies two algorithms for confidentiality: 3DES and AES-128 bit in *Cipher Block Chaining* (CBC) mode. For our implementation we use AES-256 in CBC mode with a random initialization vector (IV), as this is supported by the Java Card 2.2 crypto API and provides stronger confidentiality. The random IV is generated on the card and is shared with the off-card application during the key agreement phase together with the user's salt in step 2 of the protocol Scheme 2. Padding is performed according to ISO/IEC 7816-4 with mandatory byte value *0x80* followed by zeros to fill the block.

5.2 Message Authentication (MAC)

As in the previous case a BER-TLV data object is defined for exchanging message authentication codes for both command and response APDUs. The MAC data object for a command APDU is constructed from header bytes, data objects containing encrypted information and the expected response length. The MAC data object for the response APDU is constructed from data objects containing an encrypted response and the status word. In addition, both request and response APDU MAC computations include a counter variable which is incremented for every transmission. This prevents replay attacks on both off-card application and applet side. ETSI TS 102 176-2 specifies CBC-MAC with 3DES and AES as algorithms for computing MACs. In both cases it encrypts the last block, i.e. the output of the normal CBC-MAC operation, with a different key to protect against attacks on variable length messages [3]. In our implementation we use AES-CMAC [33] because it fixes security weaknesses of CBC-MAC [13].

5.3 Key Derivation

From the SRP key agreement, we obtain a 256 bit session key K. Since the secure messaging standard provides confidentiality and authentication, we need a different key for each purpose. The derivation of encryption and authentication keys from the shared secret is done in accordance with ANSI X9.63 [1] using a cryptographic hash function and a counter. If K is the session key from the key agreement, H the secure hash function and c the counter variable; encryption and authentication keys are derived with:

$$K_{\text{Enc}} = H\left(K \mid\mid c\right) \quad \text{where} \quad c = 1$$
$$K_{\text{Auth}} = H\left(K \mid\mid c\right) \quad \text{where} \quad c = 2$$

We use SHA-256 as the hash function. The newly derived encryption and authentication key parameters serve only for one secure messaging session.

6. IMPLEMENTATION

6.1 Implementation of SRP-6a for Key Agreement

In recent years the SRP protocol has been adopted by different standards (e.g. TLS-SRP [35]) and it has also been implemented in many cryptographic libraries[5]. In contrast,

[5]http://en.wikipedia.org/wiki/Secure_Remote_Password_

there is no implementation of SRP for Java Cards (even in the latest Java Card specifications). However, the Java Card environment supports the execution of SRP within hardware even though the explicit operations in the API are missing. In our implementation we exploit the Java Card 2.2 crypto APIto perform server-side SRP computation in a reasonable amount of time. Our implementation supports SRP prime modulus sizes of 1024, 1536 and 2048 bits. For the evaluation of the protocol we also included 512 and 768 bit versions. According to recommendations [14], in order to get 80 bit security the modulus in RSA or Diffie–Hellman should be at least 1248 bit long. Therefore we recommend using 1536 and 2048 bit implementations.

The important building blocks of the Java Card protocol implementation are discussed in more detail in the following:

- **Server-side static parameters** (before step 1 of protocol Scheme 2): On the server-side, the password verifier v, the multiplier parameter k and the product of the two (kv) remain constant as long as the user password is not changed. This is a performance advantage as the static values can be computed once during applet installation and stay the same until the user changes the password.

- **Server secret key generation** (before step 1 in protocol Scheme 2): For our implementation we use a 256 bit random output from the smart cards secure random number generator. According to RFC 5054 [35], the ephemeral secret parameters a and b should be at least 256 bits long. The advantage of sticking to this minimum recommended length is a better performance in modular exponentiation operations involving the secret parameters.

- **Public parameter computation** (step 1 in Scheme 2): The public key parameter is computed as $B = kv + g^b$. Since kv is already computed during applet initialization, the remaining operations are one modular addition and one modular exponentiation.

- **Shared secret computation** (step 4 in Scheme 2): The shared secret is computed as $S = (Av^u)^b$. The variable A is the public key of the client, u the random scrambling parameter and b the ephemeral secret value. This computation contains one modular multiplication and two modular exponentiations. The main difficulty in computing this operation on Java Card is that there is currently no support for modulo operations over big numbers. Since Java Card 2.2 a single class has been added to support big numbers which, however, does not support modular arithmetic operations. Moreover, it is included under optional packages and is not implemented by most Java Cards available on the market today. These limitations forced us to look for other options to perform operations on big numbers.

6.1.1 Big Numbers and Modular Arithmetic

Our implementation for modulo operations over big numbers is motivated by previous scientific work in [7, 34, 36] – especially, by BigNat[6] (based on research by Tews et al. [36]), a generic open source library for big numbers which supports modulo operations. Because of memory limitations we only implement specific big number operations tailored to our purpose.

- Big numbers are handled with byte arrays: While it is also possible to use arrays of short data types, using byte arrays is more convenient and can be used by native APIs with no need for conversion.

- Big number modular addition and subtraction can be computed efficiently on a Java Card Virtual Machine using basic subtraction and addition in base-256 encoded numbers. For example, with our implementation a single modular addition and subtraction of 1024 bit numbers on JCOP 2.4.1 smart cards take an average of 27 or 54 ms, depending on the result being less or greater than modulus n.

- Big number modular exponentiation: The suitable way for big number modular multiplication as explained in [34, 36] is to leverage the Java Card RSA crypto API which is accelerated by a cryptographic co-processor. In [36] it is stated that a pure Java Card implementation for multiplication of 2048 bit long numbers takes about 64 seconds. From such a performance we can conclude that an implementation of SRP without hardware acceleration is impractical. The usage of hardware for big number modulo operations is made possible by using RSA public key encryption [30] without padding. The Java Card crypto API for RSA supports setting the plain text parameter m and the public exponent e of the RSA encryption $c \equiv m^e \pmod{n}$. By simply using our exponent as exponent of the RSA public key (e) and a base padded with leading zeros as the plain text (m), the cipher text we get from encryption with the public key is the result of a modular exponentiation. With the methods mentioned above the computation for a 1024 bit public parameter B completes in less than 150 ms on JCOP 2.4.1 smart cards.

- Finally, modular multiplication is simplified by using square multiplication [34,36] which reduces the equation to modular additions, subtractions and squaring.

$$x \cdot y = \frac{((x+y)^2 - x^2 - y^2)}{2}$$

  This operation requires 3 modular exponentiations (which can be done with RSA public key encryption as mentioned in the previous bullet point), 1 – 4 additions, 2 – 3 subtractions and one right shift for the division by 2. The number of additions and subtractions varies if single results are out of the modulus range. This also influences the overall time (e.g. on JCOP 2.4.1 a single 1024 bit multiplication takes between 130 and 230 ms). This method is used for the computation of the shared secret S (step 4 in Scheme 2) and kv (during installation time).

6.1.2 Implementation Notes

As of the latest Java Card specification (Java Card 3.0) there is no support for transient RSA public keys. Transient RSA keys are included in JCOP-specific extension API. However, in order to support as many devices as possible we recommend to use standard Java Card facilities. The drawback of using static RSA keys, which reside in persistent

protocol#Real_world_implementations

[6] http://www.sos.cs.ru.nl/ovchip/

Table 2: SRP with 2048 bit modulo performance: computation times for card-side key agreement phase (agt.), key verification phase (verif.) and complete (comp.) protocol run-time (including data transfer time).

		agt.	verif.	comp.
microSD SE	min [ms]	2744	341	3170
	max [ms]	3318	377	3787
	median [ms]	3036	356	3496
external smart card	min [ms]	1204	89	1498
	max [ms]	1477	98	1960
	median [ms]	1293	90	1600

Table 3: Card-side secure messaging performance (including data transfer time).

		Data size in bytes			
		16	32	64	128
microSD SE	min [ms]	322	375	463	655
	max [ms]	360	415	534	738
	median [ms]	337	384	494	687
external smart card	min [ms]	88	90	95	105
	max [ms]	97	103	107	143
	median [ms]	90	92	97	107

memory, is that read and write operations on EEPROM are very slow compared to operations on transient memory.

One solution for this is to initialize two RSA public key objects, one with fixed exponent 2 for squaring and another one for 32 byte long ephemeral exponents u and b. Using separate public key objects for squaring reduces the amount of write operations to the EEPROM for each authentication cycle. In total we currently need two EEPROM rewrites per cycle.

6.2 Secure Messaging Implementation

In our secure messaging implementation, the operations needed to wrap and unwrap APDUs have a small memory overhead because of paddings and intermediate operations. However, no extra memory is needed as the transient memory allocated for the key agreement phase is large enough to be reused for operations in secure messaging.

The encryption operation is straightforward as AES-CBC-256 is supported in Java Card 2.2. For the MAC operation, the algorithm AES-CMAC-128 is not included in the Java Card standard 2.2. However, cipher-based MACs could be implemented efficiently in the Java Card environment if their underlying cipher algorithm can be executed with hardware support. For AES-CMAC-128, the underlying block cipher AES-CBC-128 is supported by the Java Card standard.

Additionally, the Java Card standard 2.2 supports AES-CBC-MAC signatures. The internal operations of AES-CMAC and AES-CBC-MAC are similar, except for the sub-key derivation and XORing of the last input block employed by CMAC. After this XOR operation of the last block with the corresponding sub-key, the remaining operation of AES-CMAC is the same as for AES-CBC-MAC [33].

7. PERFORMANCE EVALUATION

7.1 Secure Channel Protocol Establishment

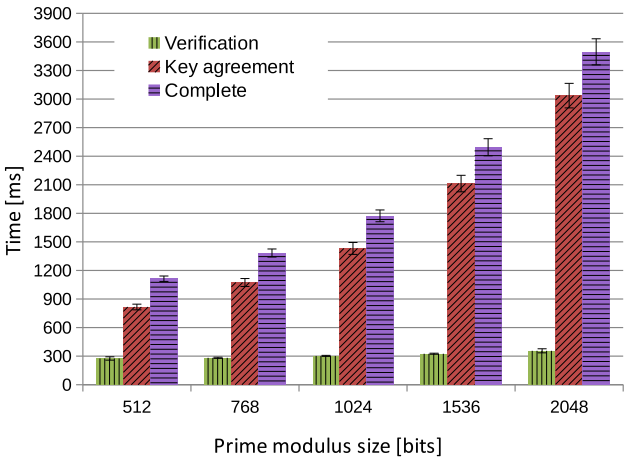
In our SRP-6a based password-authenticated key agreement protocol implementation, we use two rounds of message exchange as stated in [38]. The first round is the key agreement phase where the two parties generate the shared secret S and the session key K. This includes operation steps 1-5 shown in Scheme 2. The second message exchange round is the verification phase which is shown in steps 6 and 7. In this section, we show the performance of the server-side computations of these two phases using two Java Card variants and different modulus sizes. For the first test scenario we used a DeviceFidelity microSD SE (credenSE 2.8J), a Samsung Galaxy S3 with the *SuperSmile ROM*[7], and Open Mobile API for accessing the secure element on the card. We made the measurements using $System.nanoTime()$ before sending and after receiving the APDU. For the second case we used a JCOP 2.4.1 external smart card with contact-less interface to give insight on performance differences between different tamper resistant hardware variants. Measurements were taken using NetBeans application profiler[8] on the client-side. Although our main focus is the performance evaluation of the protocol in Java Card applets, we also include the data transfer time between the client application and the Java Cards. This should give a better insight on the actual use cases of the protocol implementation. In both test cases we took 100 measurements. The results of the evaluation are split into four parts:

1. **Key agreement performance:** This evaluation show the request-response time required from sending the public parameter A in the client application until receiving the public parameter B and the user's salt s (steps 1 to 5 in Scheme 2). The length of the user salt is 16 bytes, while the public key parameter has a length equal to the prime modulus used in the test.

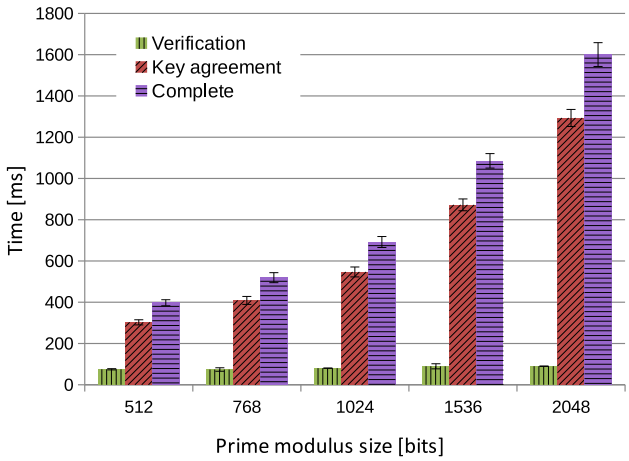
   From the performance results in Table 2, we observe that there is a considerable difference between the minimum and maximum time required to complete the operations. This time difference is introduced by extra modular additions and subtractions needed to get the right modulus result when there is an overflow in other modular additions and subtractions implemented in software. From a cryptanalysis point of view this could give an attacker some information about the ephemeral secret key and verifier parameters used in the SRP protocol. Such issues can be solved by introducing dummy addition and subtraction operations which compensate the time difference. The time difference in the two test scenarios (external smart card and microSD SE) is caused by the differences in hardware implementation and communication channel. Due to the usage of standard file system IO for exchanging data, the microSD SE is much slower in transmitting data (cf. time measurements in [22]).

2. **Performance of verification:** As it is shown in steps 6 and 7 of Scheme 2, the verification of the shared secret consists of one message exchange of the verifiers M_1

[7] http://usmile.at/downloads
[8] https://profiler.netbeans.org/



(a) Performance on JCOP 2.4.1 microSD SE



(b) Secure channel performance on JCOP 2.4.1 external smart card

Figure 2: Performance evaluation of different modulus sizes with standard deviation on both Java Card variants.

and M_2. For our implementation we use a SHA-256 hash function, so that both M_1 and M_2 are 32 bytes. However, since the shared secret S is also included in the computation of M_1 and M_2, the performance depends on the length of S which is equal to the size of the prime modulus n.

In many use cases, the performance of the verification stage is close to the actual time that a user has to wait before the secure communication starts. This is because the time intensive key agreement phase can be started in the background while the user enters the PIN/password. As shown in the results of the evaluation in Table 2, the verification operation takes a maximum time of 377 ms (microSD SE) and 98 ms (external smart card).

3. **Complete secure channel establishment:** This case includes the complete protocol running time required by the client- and server-side. For the client-side computations we use the Bouncy Castle crypto API[9].

4. **Evaluation of prime modulus sizes:** In Figure 2 we visualize the influence of different prime modulus sizes to the required computation time. Our protocol supports prime modulus sizes of 1024, 1536 and 2048 bit. For the purpose of performance evaluation we also included 512 and 768 bit though we recommend to use 1536 and 2048 bit versions. Higher prime modulus sizes of above 2048 bit are currently not supported due to the limitation of RSA operations in the current Java Card standard.

   The results in Figure 2 show that the key agreement time of the protocol is significantly influenced by the prime modulus size. At the same time, the verification phase of both Java Card variants is not varying much. As already described, the big advantage of this is the minimized time the user has to wait after entering PIN/password.

[9]http://www.bouncycastle.org/

7.2 Secure Messaging Performance

In this section we analyse the performance of our secure messaging implementation with both Java Card variants. To perform these tests, the client-side wraps random data in a secure request APDU object and sends this object. On the Java Card side, a dummy applet receives the incoming secure request APDU and unwraps it (performs MAC verification and decryption) to get the random data from the client. Then this random data is encoded inside a secure response APDU object and is sent back to the client. Table 3 shows the median, minimum and maximum results for different packet sizes. The values for the external smart card vary between 90 and 107 ms, with a data rate reaching 1.19 kB/s. The microSD SE is slower with a median data rate of up to 186 B/s.

7.3 Memory Optimization

In addition to slight adaptations, discussed in section 6.2, we optimize our implementation by considering performance, memory and security trade-offs:

- Using the APDU buffer for performing and storing intermediate operations and public values.
- Using static memory offsets for different sections instead of allocating several smaller areas to save memory.
- Sharing (reuse) of transient memory between key agreement and secure messaging implementation.

The current implementation with 2048 bit prime uses 1040 bytes for byte arrays in persistent (EEPROM) and 834 bytes for byte arrays in transient (RAM) memory.

8. CONCLUSION

In this paper we propose an efficient way of implementing a secure communication between Java Card applets and off-card applications in a mutually authenticated secure channel based on the Secure Remote Password (SRP) protocol and a standard authenticated encryption scheme. Although

the Java Card environment is equipped with the necessary hardware for computation of modulo operations in SRP, limitations in Java Card APIs on accessing the cryptographic co-processors make it challenging to implement SRP with acceptable performance. However, by exploiting the RSA encryption API provided by the platform, we show that it is possible to compute exponentiations and multiplications with support of the cryptographic co-processor. This, and minor adaptations to the protocol, made it possible to implement the SRP-6a server-side in a Java Card applet with reasonable computation time. For our implementation with a 2048 bit long prime modulus, the complete protocol runs in less than 2 seconds for the smart card and less than 4 seconds for the secure element tests. However, considering our use cases, a user only has to wait for the verification phase (i.e. less than 100 ms for the smart card and 400 ms for the secure element) since the time intensive key agreement phase runs simultaneously with the password/PIN entry. Finally, we also provide an applet level implementation for the ISO/IEC 7816-4 secure messaging standard. The source code of the whole implementation is available under an open source license[10].

9. ACKNOWLEDGMENTS

This work has been carried out within the scope of u'smile, the Josef Ressel Center for User-Friendly Secure Mobile Environments. We gratefully acknowledge funding and support by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, and NXP Semiconductors Austria GmbH.

10. REFERENCES

[1] American National Standards Institute, American Bankers Association, and Global Engineering Documents (Firm). *American National Standard for Financial Service X9.63-2001 : Public Key Cryptography for the Financial Services Industry*. American Bankers Association, 2001.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology—CRYPTO'96*, page 1–15, 1996.

[3] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *J. Comput. Syst. Sci.*, 61(3):362–399, Dec. 2000.

[4] M. Bellare and P. Rogaway. The AuthA protocol for password-based authenticated key exchange. In *IEEE P1363*, pages 136–3, 2000.

[5] S. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, 1992.

[6] N. Ben-Asher, N. Kirschnick, H. Sieger, J. Meyer, A. Ben-Oved, and S. Möller. *On the Need for Different Security Methods on Mobile Phones*, page 465–473. MobileHCI '11. ACM, 2011.

[7] P. Bichsel, J. Camenisch, T. Groß, and V. Shoup. *Anonymous credentials on a standard Java Card*, page 600–610. CCS '09. ACM, 2009.

[10] https://gitorious.org/secure-element/secure-channel-srp6a-android-lib and https://gitorious.org/secure-element/secure-channel-srp6a-applet

[8] S. A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. MIT Press, 2000.

[9] E. Brickell, J. Camenisch, and L. Chen. *Direct anonymous attestation*, page 132–145. CCS '04. ACM, 2004.

[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 239–252. ACM, 2011.

[11] A. De Luca, A. Hang, F. Brudy, C. Lindner, and H. Hussmann. Touch me once and i know it's you!: implicit authentication based on touch screen patterns. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, page 987–996, New York, NY, USA, 2012. ACM.

[12] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[13] M. J. Dworkin. SP 800-38B. Recommendation for block cipher modes of operation: The CMAC mode for authentication. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2005.

[14] European Network of Excellence in Cryptology II. ECRYPT II yearly report on algorithms and keysizes. June 2011.

[15] R. D. Findling and R. Mayrhofer. Towards face unlock: On the difficulty of reliably detecting faces on mobile phones. In *Proc. MoMM 2012: 10th International Conference on Advances in Mobile Computing and Multimedia*, pages 275–280, New York, USA, 2012. ACM.

[16] V. Gayoso Martinez, C. Sanchez Avila, J. Espinosa Garcia, and L. Hernandez Encinas. *Elliptic curve cryptography: Java implementation issues*, pages 238–241. Oct 2005.

[17] GlobalPlatform. Secure channel protocol – GlobalPlatform card specification v2.2 - Amendment D, 2009.

[18] J.-H. Han, Y.-J. Kim, S.-I. Jun, K.-I. Chung, and C.-H. Seo. *Implementation of ECC/ECDSA cryptography algorithms based on Java card*, pages 272–276. 2002.

[19] G. Hancke. A practical relay attack on ISO 14443 proximity cards. Technical report, 2005.

[20] F. Hao and P. Y. A. Ryan. Password authenticated key exchange by juggling. In *Proceedings of the 16th International conference on Security protocols*, Security'08, page 159–171, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] S. Höbarth and R. Mayrhofer. A framework for on-device privilege escalation exploit execution on android. In *Proceedings of IWSSI/SPMU*, 2011.

[22] M. Hölzl, R. Mayrhofer, and M. Roland. Requirements analysis for an open ecosystem for embedded tamper resistant hardware on mobile devices. In *Proc. MoMM 2013: International Conference on Advances in Mobile Computing and Multimedia*, Vienna, Austria, 2013. ACM.

[23] D. P. Jablon. Strong password-only authenticated key exchange. *SIGCOMM Comput. Commun. Rev.*,

26(5):5–26, Oct. 1996.

[24] S. Khan, M. Nauman, A. Othman, and S. Musa. *How secure is your smartphone: An analysis of smartphone security mechanisms*, page 76–81. 2012.

[25] M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys Tutorials*, 15(1):446–471, 2013.

[26] M. Landman. *Managing Smart Phone Security Risks*, page 145–155. InfoSecCD '10. ACM, 2010.

[27] S. Lucks. Open key exchange: How to defeat dictionary attacks without encrypting public keys. In B. Christianson, B. Crispo, M. Lomas, and M. Roe, editors, *Security Protocols*, LNCS, pages 79–90. Springer Berlin Heidelberg, Jan. 1998.

[28] T. Mantoro and A. Milisic. Smart card authentication for internet applications using NFC enabled phone. In *2010 International Conference on Information and Communication Technology for the Muslim World (ICT4M)*, pages D13–D18, 2010.

[29] R. Mayrhofer. An architecture for secure mobile devices. *Security and Communication Networks*, 2014.

[30] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[31] M. Roland, J. Langer, and J. Scharinger. Practical attack scenarios on secure element-enabled mobile devices. In *2012 4th International Workshop on Near Field Communication (NFC)*, pages 19–24, 2012.

[32] A. Ruiz-Martinez, O. Canovas, and A. Gomez-Skarmeta. Smartcard-based e-coin for electronic payments on the (mobile) internet. In *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, 2007. SITIS '07*, pages 361–368, 2007.

[33] J. Song, R. Poovendran, J. Lee, and T. Iwata. The AES-CMAC Algorithm. RFC 4493 (Informational), 06 2006.

[34] M. Sterckx, B. Gierlichs, B. Preneel, and I. Verbauwhede. Efficient implementation of anonymous credentials on java card smart cards. In *Information Forensics and Security, 2009. WIFS 2009.*, page 106–110, 2009.

[35] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. RFC 5054, 11 2007.

[36] H. Tews and B. Jacobs. Performance issues of selective disclosure and blinded issuing protocols on java card. In *Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks*, page 95–111. Springer, 2009.

[37] T. Wu. The secure remote password protocol. In *Proc. of the 1998 Internet Society Network and Distributed System Security Symposium*, page 97–111, Nov. 1998.

[38] T. Wu. SRP-6: improvements and refinements to the secure remote password protocol. *http://srp.stanford.edu/*, Oct. 2002.