EAP-TLS Smartcards, from Dream to Reality

[1]Pascal Urien, [1]Mohamad Badra, [2]Mesmin Dandjinou
1-ENST Paris, 2-Université Polytechnique de Bobo-Dioulasso, Burkina Faso.
Pascal.Urien@enst.fr, badra@enst.fr, mesmin.dandjinou@voila.fr

***Abstract*-This paper presents the first implementation of the EAP-TLS (Extensible Authentication Protocol-Transport Layer Security) protocol in smartcards. Tests, performed on two java devices, are discussed and analysed. Results show that TLS processing is slow, because smartcards are not (yet) optimized for that purpose; however we clearly demonstrate that TLS processing in smartcards is not today a dream, and can be realized with existing components.**

Key words **Smartcard, Security, EAP, Wi-Fi, TLS.**

I. Introduction

Wireless 802.11 LANs have introduced new security threats, sometime refereed as parking lot attack [1]. User authentication is a prerequisite, before accessing to services available in wireless infrastructures. The extensible authentication protocol [2, 3] is a powerful umbrella that shelters multiple authentication scenarios. It is the cornerstone of the IEEE 802.1x standard [4] which defines key exchange mechanisms between the wireless user (the supplicant) and the authentication server (see figure 1). All emerging security architectures like WPA [6] or IEEE 802.11i [5] are based on 802.1x facilities for users' authentication and cryptographic material calculation. As an illustration the Internet Authentication Server (IAS) is the Microsoft implementation of a RADIUS server, and is natively available in new server platforms.

Although multiple authentication methods have been introduced, like EAP-MD5 [2], EAP-SIM [9], PEAP [10], MSCHAPv2 [11], our paper is focused on EAP-TLS [7], which defines a framework, that transparently transports TLS [8] messages.

We present the first implementation of this protocol in a smartcard. For interoperability reasons, this software was written in java; tests were performed on two cards, offering 32 KB of storage space, and working with different microcontrollers and operating systems.

Section 3, describes the EAP-TLS smartcard architecture, section 4 shows results for two components, section 5 presents final results with an optimized software.

II. Architecture

A. The EAP smartcard

An EAP smartcard is an opened, ISO 7816 [12] microcontroller supporting most of authentication protocols. It is described by an internet draft [14] and a more detailed description may be found in [15]. This innovative technology won "The Best Technological Innovation" award at the "cartes2003" exhibition [13].

EAP methods are computed in a trusted and tamper resistant environment that also securely stores network credentials (shared secrets, RSA private keys, etc.). Smartcard benefits include the following features:

- ***Scalability***. One billion smartcards were produced in 2003; multiple form factors are available like credit cards, SIM modules and USB interfaces.

- ***Sufficient performances***. RSA 2048 bits computations are done in less than 500 ms, memory sizes (available for the java applications) are around 64 KB, but one megabyte is already available thanks to the FLASH technology.

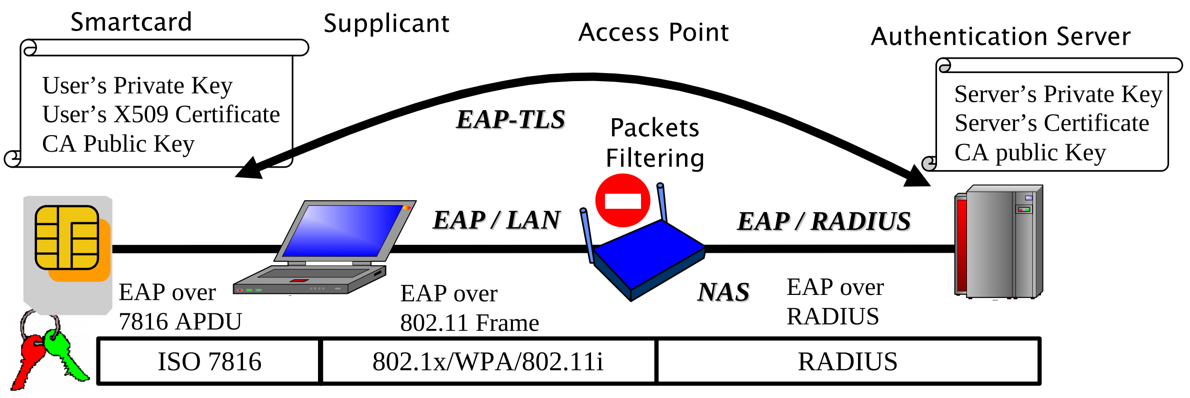


Figure 1. EAP-TLS functional architecture.

The EAP smartcard offers four classes of services,

- *Network interface*. EAP messages (requests and notifications) are transported by ISO 7816 APDUs and processed by a secure microcontroller. If session keys (Pairwise Master Key, etc.) are needed by radio security protocols (WPA, 802.11i), the smartcard computes and delivers these keys to the supplicant operating system.
- *Operating System interface*. Multiple triplets (EAP-ID, EAP-Type, cryptographic keys) are stored in smartcard and identified by a parameter called Identity. Each Identity may be associated to a profile that for example stores a list of preferred SSIDs or an X509 certificate.
- *Management/Personalization interface*. This service provides smartcards personalization (identity update, etc.) and management facilities.
- *User/Issuer Interface*. Two Personal Identification Number (PIN) codes are available. The first one authenticates the smartcard bearer whereas the second one protects data (identity triplets, etc.) and is managed by card issuers.

B. Integrating Smartcards with the Extensible Authentication Protocol

According to [2], EAP implementation conceptually consists of the three following components (see figure 2):
- *Lower layer*. The lower layer is responsible for transmitting and receiving EAP frames between the peer and authenticator
- *EAP multiplexing layer*. The EAP layer receives and transmits EAP packets via the lower layer, implements duplicate detection and retransmission and delivers and receives EAP messages to and from EAP methods.
- *EAP method*. EAP methods implement the authentication algorithms and receive and transmit EAP messages via the EAP layer. Since fragmentation support is not provided by EAP itself, this is the responsibility of EAP methods.

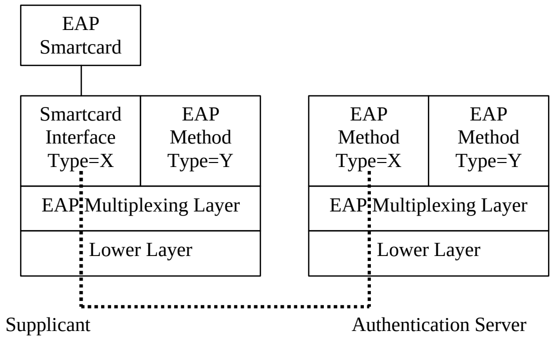


Figure 2. Smartcard role in an EAP implementation.

An EAP smartcard implements an EAP method and works in cooperation with a smartcard interface entity, which sends and receives EAP messages to/from this component. The simplest form of this interface is a software bridge that transparently forwards EAP messages to smartcard. According to EAP methods complexity and smartcard computing capacities, protocol sub-sets, which do not deal with security features may be computed by the smartcard interface entity.

C. EAP-TLS Segmentation Issues

According to [7] a TLS record may be up to 16384 bytes in length, a TLS message may span multiple TLS records, and a TLS certificate message may in principle be as long as 16MB. Furthermore the group of EAP-TLS messages sent in a single round may thus be larger than the maximum LAN frame size. Therefore EAP-TLS [6] introduces a segmentation process that splits TLS messages in smaller blocs, acknowledged by the recipient.

The RADIUS server generates acknowledgement requests and the supplicant acknowledgment responses.

A double segmentation mechanism (see figure 3) is necessary in order to forward TLS packets to smartcard. These messages are divided in smaller segments, whose size is typically 1400 bytes, and than encapsulated in EAP-TLS packets. Each block is again split in a collection of APDUs (ISO7816 commands), whose size is around 240 bytes, and that are forwarded to the smartcard.

TLS messages produced by smartcard are transported by EAP-TLS (response) fragments whose length is around 240 bytes. These blocs are directly encapsulated in APDU responses. The smartcard interface, running on the supplicant side, can reassembly these information and choose another segment size.

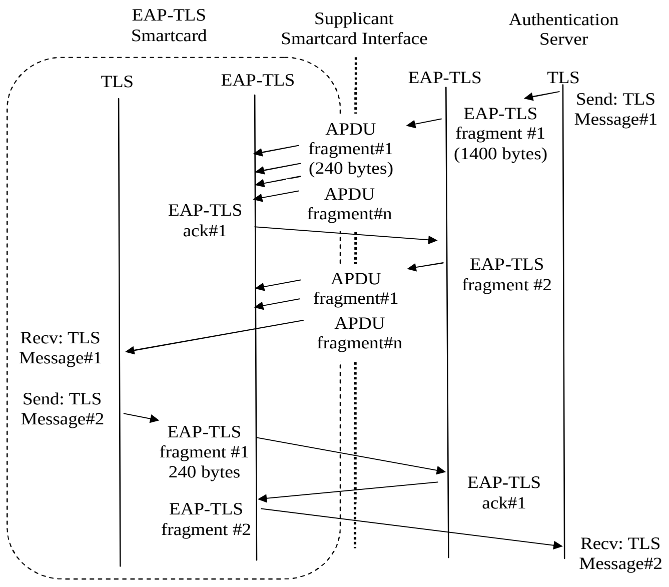


Figure 3. The double segmentation process

D. Smartcard benefits for EAP-TLS protocol

Mutual authentication is a prerequisite for security of wireless LANs. From the network point of view, user's authentication establishes its authorizations and privileges. From the user's point of view, the access network must be authenticated in order to avoid associations to rogue access points.

As defined in the TLS specification [7], server is authenticated by its X509 certificate and client by the RSA encryption (with its private key) of a dual digest (MD5 + SHA1) of all previous handshake messages.

There are two ways for using smartcards in the EAP-TLS protocol. In the first mode, smartcard stores a certificate and performs an RSA encryption with its private key. It is a kind of blind signature because smartcard does not check the server certificate. This mode is supported by [14] thanks to appropriate procedures named method functions.

In the second mode, smartcard processes all TLS messages. In particular, the server certificate is checked and the RSA signature is delivered only upon success of this operation. Our experimental smartcard supports these two modes, but for obvious security reasons we focus our work on the second one.

E. EAP-TLS smartcard architecture

Our EAP-TLS smartcard (see figure 4) is organized around two software entities instanced by two java classes.

- The first layer (EAP.class) implements basic EAP facilities as described in [14, 15]; especially the identity management and the PIN code verification.

- The second layer (TLSUtil.class) implements EAP-TLS and TLS services. It realizes all segmentation and re-assembly operations. It also analyses and produces TLS messages. It two main components are the handshake protocol entity, that processes the mutual authentication protocol, and the record layer which encrypts and signs TLS messages. A dedicated module (Certificates Management) checks incoming server certificates.

The smartcard contains the Certificate Authority (CA) public key (2048 bits), the client private key (1024 bits) and its X509 certificate.

The total code byte size is around 22 KB including about 10KB of data stored in the non volatile memory (E2PROM). Therefore, it was possible to download this package on different JavaCard (JC) platforms, offering at least 32 KB of storage space.

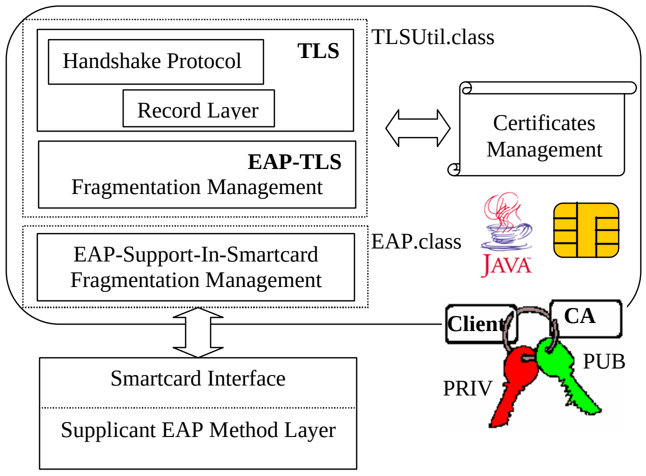
JavaCard 2.1 [16] platform natively provides essential cryptographic services that are required by the TLS protocols; in particular:

- Random number generation.
- MD5 and SHA1 digest functions.
- RSA public key encryption and decryption.
- RSA private key encryption and decryption.
- DES or 3DES ciphering.

However some additional facilities that are not currently available in JC platforms are provided by the EAP-TLS application. For example:

- Keyed-hashing procedures ([17] HMAC-MD5 and HMAC-SHA1).
- The pseudo random function (PRF) defined by the TLS protocol.
- The RC4 algorithm, which is often used by the TLS record layer.
- An X509 certificate parser required for signature analysis and public key extraction.

Java procedures, which are interpreted by the java virtual machine, are obviously slowest than those that are directly handled by the smartcard operating system. Next generation of JavaCard could support these facilities if there is a strong market requirement.



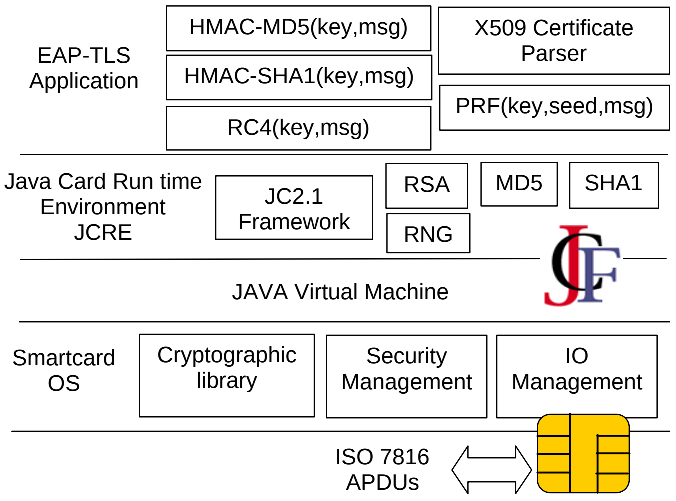


Figure 4. Software architecture (left side), relationship with the JavaCard platform (right side)

III. Results

A. Benchmark realization

First we built a Certification Authority and delivered X509 certificates for both a RADIUS authentication server and a Windows Wi-Fi equipped terminal. Then, we recorded an EAP-TLS authentication scenario between these two entities. This trace was reverse-engineered and all cryptographic calculations were checked. EAP-TLS messages, issued by the authentication server, were encapsulated in a reference list of 7816 APDUs.

B. EAP-TLS authentication scenario

Schematically at TLS level, the authentication scenario works according to a four-way handshake.

Step1. A client (the EAP-TLS smartcard) sends a first message named ClientHello that includes a 32 bytes random number. The message size is quite small (70 bytes in our benchmark). The first four bytes of the nonce represent the UNIX time encoded value. As the smartcard is not able to maintain a clock, the supplicant interface adds this parameter to the EAP-TLS Start message.

Step2. The authentication server produces a message (4710 bytes in our benchmark) including a random number (32 bytes) and a list of certificates.

Step3. According to the TLS protocol, the smartcard client checks server certificates and performs additional cryptographic calculations. Finally, it delivers a response (1825 bytes in our benchmark) signed with its private key. We divide this process in 13 sub parts:

a- *Message Transfer*. Thanks to a double segmentation mechanism, the TLS message (4710 bytes, transported by four EAP-TLS segments) is downloaded by the smartcard.

b- *Server Certificate Analysis*. The signature included in the certificate is checked with the CA public key. Server public key is extracted.

c- *Pre-Master-Secret Generation*. Smartcard generates a 48 bytes nonce (pre master secret) and encrypts this value with the server public key.

d- *Verify Digest*. The Smartcard (SC) computes a dual digest (MD5+SHA1) of previously received data (as required by the verify protocol).

e- *Verify RSA Encryption*. The dual digest is encrypted with the client private key.

f- *Master Secret Calculation*. The SC computes a master secret by applying the PRF function to the pre master secret and the random values.

g- *Key block generation*. Cryptographic material is generated by applying the PRF function to the master secret and the random values.

h- *Client Finished Calculation*. The SC performs a dual digest (MD5+SHA1) of previously received messages and applies the PRF function to the result before computes data transported by the finished protocol.

i- *Server Finished Calculation*. The SC performs a dual digest (MD5+SHA1) of previously received messages. This value should be included in the server finished message that will end the TLS handshake phase.

j- *MAC Record Calculation*. The Record layer computes the HMAC value appended to the finished message with an integrity key extracted from the key block.

k- *RC4 Initialization*. The SC initializes the RC4 algorithm with the encryption key extracted from the key block.

l- *Record Enciphering*. The SC encrypts using RC4 the finished message and its associated HMAC.

m- *Response Transfer*. TLS response (1825 bytes, transported by eight EAP-TLS messages) is sent to the authentication server.

Step4. The server produces the last message (43 bytes in our benchmark) which is verified by the EAP-TLS smartcard that finally computes a master key (PMK, Pairwise Master Key, etc.). The supplicant uses this key for security protocols (801.1x, WPA, etc.) initialization. We divide this process in five sub parts:

a- *Record Decryption*. The SC initializes the RC4 algorithm with the encryption key extracted from the key block. It decrypts the incoming record block, containing the server finished message.

b- *MAC Record Checking*. The record layer computes the HMAC value appended to the finished message using the appropriate integrity key extracted from the key block.

c- *Server Finished Calculation*. The SC uses the PRF function for computing and checking the data transported by the finished protocol.

d- *PMK Calculation*. The SC uses the PRF function in order to compute the PMK key, as defined in [7].

e- *Acknowledgment Generation*. The SC produces an EAP-TLS acknowledgment message.

C. Microcontrollers characteristics.

The EAP-TLS application was downloaded on two 32 KB JavaCard, whose physical characteristics are very close (see Table I). The two components include a cryptographic co-processor able to compute RSA algorithm (1024 or 2048 bits) in less than 500ms.

Figure 5 presents the device A functional diagram and a die picture of the component B.

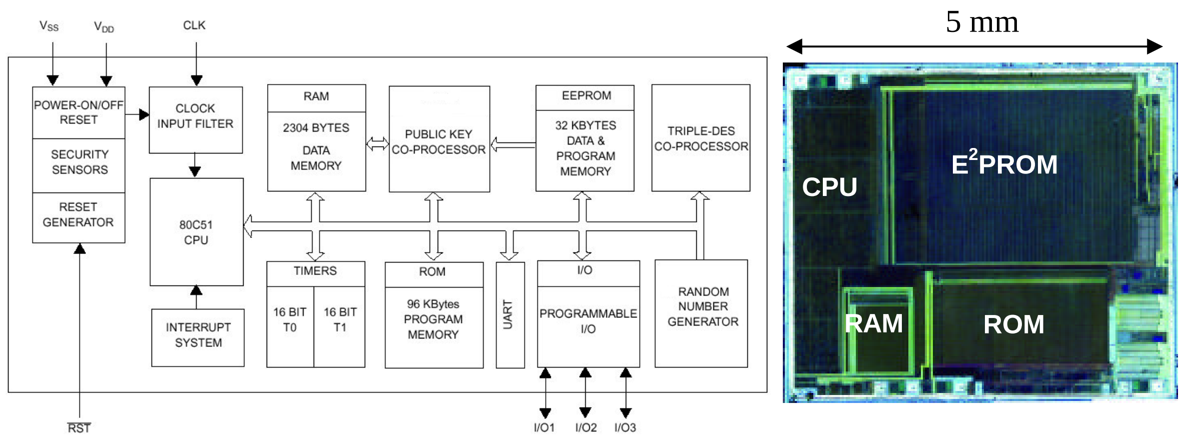


Figure 5. Device B functional diagram (left side) and device A die (right side)

TABLE I
MICROCONTROLLERS CHARACTERISTICS

Device	CPU	RAM bytes	ROM Kbytes	E^2PROM Kbytes	Max. Clock	Max Data Rate	RSA Processor	RNG
A	8 bits	2304	96	32	10 MHz	424 kbit/s	1088 bits	yes
B	8 bits	4096	96	34	8 MHz	1000 kbit/s	4032 bits	no

D.Basic performances.

We performed two basic tests in order to evaluate chips and operating systems performances. EAP-TLS application supports a family of Method Functions (as described in [12]) that are very practical in order to execute basic cryptographic operations like RSA calculations (encryption and decryption) or digest procedures (MD5 and SHA1).

As for both components, the RSA algorithm is handled by a co-processor, we observed short computing times, ranging between 100ms and 500 ms (see figure 6). Because TLS performs only three RSA calculations (with the CA public key, the server public key and the client private key), these operations consume around one second.

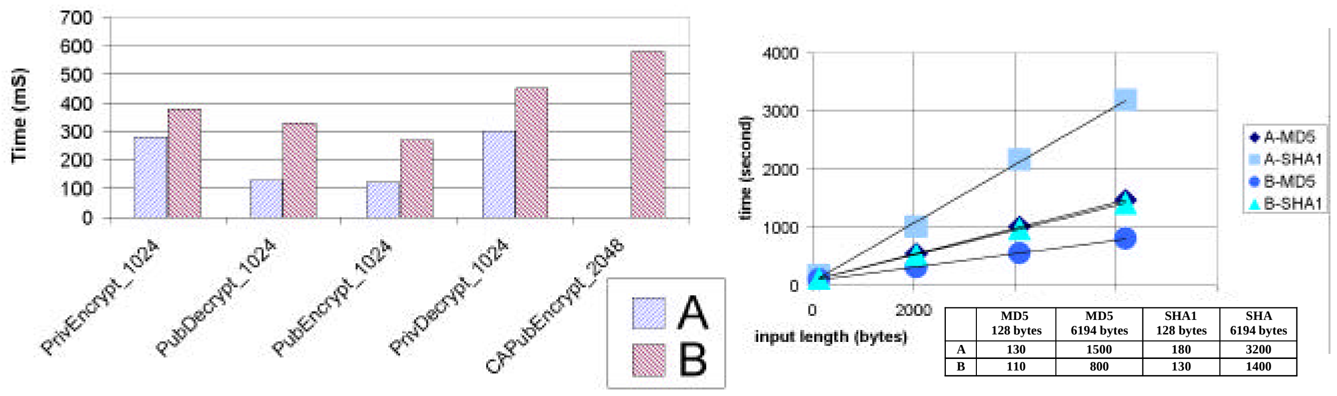
Digest algorithms are performed on blocks, whose size is 64 bytes for MD5 and SHA1 algorithms. Therefore, the computing time is dependant on the input data size. These functions are not hold by a co-processor, but stored in cryptographic libraries and executed by the smartcard CPU. Figure 6 shows that device B is two time faster than device A. We also notice that for both components, MD5 is faster than SHA1 (about 2 times). The average computing times (dual digests, MD5 & SHA1) are respectively (for device A and B):

- 155 and 120 ms for small input size (128 bytes)
- 2350 and 1100 ms for larger input size (6 Kbytes).

The handshake protocol performs three dual hashes (MD5 & SHA1) of which input size is around 6kB. Therefore these digests computing times are estimated respectively (for device A and B) to 14.1 and 6.6 seconds.

E. Other cryptographic facilities.

As we mentioned it above, cryptographic facilities that are not natively supported by JavaCard APIs are delivered by the EAP-TLS application. The following cryptographic functions have been added:



	MD5 128 bytes	MD5 6194 bytes	SHA1 128 bytes	SHA 6194 bytes
A	130	1500	180	3200
B	110	800	130	1400

Figure 6. Basic performances, RSA computing times (left part), digest functions computing times (right part)

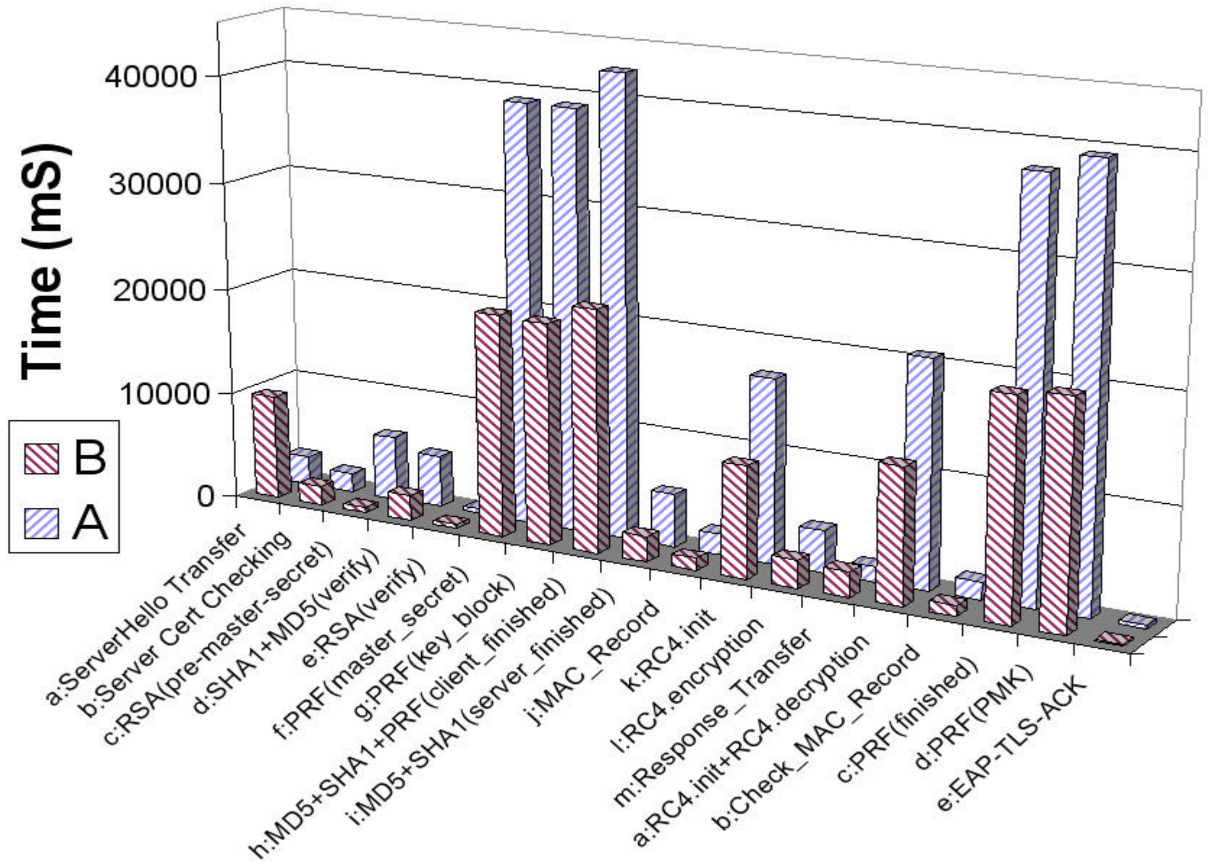


Figure 7. Performances measurement for step 2 (left part) and step 3 (right part)

- Two HMAC algorithms [17] for MD5 and SHA1 digests. These procedures work with two parameters: a key whose size is less than 64 bytes (in our TLS context) and a message whose size is less than 128 bytes. In these conditions, HMAC uses two digest functions and some additive operations. Computing time is about 2s for device A and 1s for device B.

- A TLS pseudo random function (PRF). When this algorithm outputs 80 bytes, it deals with 11 HMAC-MD5 and 9 HMAC-SHA1 procedures, whose key size is less than 64 bytes and message length less than 128 bytes. Consequently, execution time is about 40 sec for device A and 20 sec for device B. An EAP-TLS session invokes this algorithm 5 times, what respectively costs 200s and 100s.

- An RC4 procedure. This method is expansive because it deals with an array of 2048 bits stored in the E2PROM. The encryption/decryption (plus key initialization) of 32 bytes respectively costs 21 and 12.5 sec. This function is called two times by the record layer, what respectively requires 42 and 23 sec.

F. Global performances

Figure 7 shows the repartition of computing times during step2 and step3. Device A processes the EAP-TLS protocol in about 266 sec (166 + 100) and device B in 151 sec (97 + 54). The time spent to transfer data to/from smartcard is respectively 4.5 and 12 sec.

As demonstrated by figure 8, most of computing time is consumed by the PRF function. It clearly appears that digest algorithms are the main issue for fast processing of the EAP-TLS protocol in smartcards.

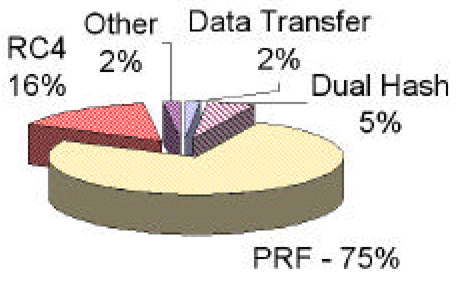


Figure 8. Repartition of computing times for device A and B

V. SOFTWARE OPTIMIZATION

As demonstrated before, the PRF function is a critical component for the EAP-TLS smartcard, therefore we tried to carefully optimize it.

The javacard language supports two kind of objects,

- persistent objects, stored in non volatile memory (E^2PROM) and associated with “slow” writing operations.
- transient objects, stored in volatile memory (RAM), associated with “fast” writing operations.

We introduced a transient byte array, and used it as often as possible in the TLS class. We observed a strong decrease of processing time, for device B it was reduced by a factor three (from 151s to 45s), and we measured the following performances,

- Step1, hello message transfer to smartcard,	10s
- Step2, processing of the server hello message,	24s
- Step3, smartcard response transfer,	2,5s
- Step4, processing of last server (TLS) message,	8,5s

It is important to notice that in the optimized implementation, each EAP message is processed in less than 30s, the maximum default value specified in [4]. Although performances should be improved, it clearly appears that today smartcards are able to fully compute the EAP-TLS protocol, in a duration compatible with the IEEE 802.1X constraints.

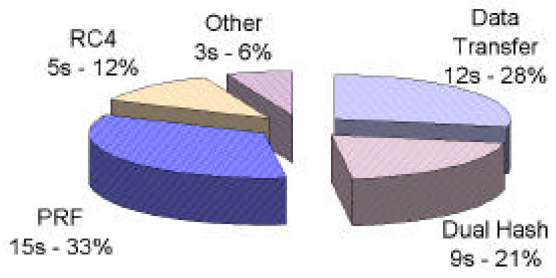


Figure 9. computing times distribution for device B (optimized software)

Figure 9 shows the observed repartition of computing times for the optimized applet.

PRF functions require 46 HMAC calculations (which should cost, according to section III.D, around 46x0,12 = 5,5s); the measured time (15s) clearly illustrates the software additive cost for this facility.

Three dual hash functions, operating on about 21 Kbytes (3x7) of data require 9 seconds (which should be equal, according to III.D, to 0,00018x6x7000 = 7,6 s).

RC4 algorithm performance has been greatly improved by the use of transient objects, that speed up memory accesses.

The data transfer factor depends on three factors, the data size (around 7 Kbytes), the serial link speed (at least 9600 baud/s) and the writing time needed for by non volatile memory.

VI. CONCLUSION

This paper describes the first prototype of an EAP-TLS smartcard. It demonstrates that today smartcards capacities, in terms of memory sizes and computing power, are sufficient for designing standalone TLS applications. However, it is necessary to improve performances; for example by introducing new Java APIs and by incorporating digests functions in smartcards cryptographic co-processors.

REFERENCES

[1] Arbaugh W, Shankar N, and Wan Y, "Your 802.11 Wireless Network has No Clothe" http://www.cs.umd.edu/~waa/wireless.pdf, 2001.
[2] RFC 2284, "Extensible PPP Authentication Protocol (EAP)", March 1998.
[3] Extensible Authentication Protocol (EAP), draft-ietf-eap-rfc2284bis-09.txt.
[4] IEEE P802.1X Approved Draft, "Port based Network Access Control", June 2001
[5] IEEE Std 802.11i/D5.0 (2003), "Draft Supplement to standard for Telecommunications and Information Exchange, Between Systems LAN/MAN Specific Requirements Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications: Specification for Enhanced Security"
[6] "Wi-Fi Protected Access (WPA) ", version 2.0, April 29 2003.
[7] RFC 2716, "PPP EAP TLS Authentication Protocol", October 1999.
[8] RFC 2246, "The TLS Protocol Version 1.0", January 1999
[9] "EAP SIM Authentication" draft-haverinen-pppext-eap-sim-12.txt, October 2003
[10] "Protected EAP Protocol (PEAP) Version 2", draft-josefsson-pppext-eap-tls-eap-07.txt, October 2003
[11] "Microsoft EAP CHAP Extensions", draft-kamath-pppext-eap-mschapv2-00.txt, September 2002
[12] ISO 7816, "Cards Identification - Integrated Circuit Cards with Contacts".
[13] http://www.cartes.com/en/frameset_dyn.htm?URL=I_trophee/I4_gagnants.htm
[14] "EAP support in smartcards”, draft-urien-eap-smartcards-05.txt, May 2004.
[15] P. Urien, M. Loutrel, "The EAP Smartcard, has tamper resistant device dedicated to 802.11 wireless networks.", ASWN’2003, Berne, Switzerland, July 2003.
[16] Zhiqun Chen, "Java Card Technology for Smart Cards”, ADDISON-WESLEY.
[17] RFC 2104, "HMAC: Keyed-Hashing for Message Authentication", September 1997.