Designing Smartcards for Emerging Wireless Networks

Pascal Urien[1] and Mesmin Dandjinou[2]

[1] ENST 37/39 rue Dareau, Paris 75014, France
[2] Université Polytechnique de Bobo-Dioulasso, Burkina Faso
Pascal.Urien@enst.fr
Mesmin.Dandjinou@voila.fr

Abstract. This paper presents our work relating to introduction of EAP smartcards in emerging wireless LAN like Wi-Fi or WiMax. We analyse basic characteristics involved in authentication protocols from feasibility and performances points of view. We shortly introduce our open Java architecture, and underline some observed interoperability issues. We present and analyze results obtained with five different smartcards, for two authentication scenarios: the first one works with an asymmetric algorithm (EAP-TLS, a transparent transport of the well known SSL standard), and the second method uses the EAP-AKA protocol, which is an adaptation of the symmetric Milenage algorithm. We introduce a new class of smartcard which acts as EAP server, and that has been successfully tested in operational networks. Finally we suggest a new way to manage and use smartcards, remotely and securely, by using Trusted EAP Modules.

1 Introduction

In 1999, the IEEE 802 committee ratified the 802.11 standard [3] which introduced the first wireless Ethernet network, later enhanced with 802.11i standard [9]. Wi-Fi technology became the foundation stone of cheap IP radio LAN. Two years after, an "*Air Interface for Fixed Broadband Wireless Access Systems*" was proposed in [6] that extends wireless IP connectivity at a campus scale. The emerging IEEE 802.16e [14] standard, "*Air Interface for Fixed and Mobile Broadband Wireless Access Systems*", provides enhancements to [6] in order to support subscriber stations moving at vehicular speeds.

But unlike the GSM network, no security module was specified in the so called *Wi-Technologies*. There is however a common denominator between [5] and [6], both of them supports the *Extensible Authentication Protocol* [10]. EAP is a light protocol, used prior to IP address allocation, that may transport multiple authentication scenari like EAP-TLS [2] or EAP-AKA [13].

An EAP dialog occurs between an EAP client, that wants to gain access to network resources, and an EAP authenticator which is the heart of an AAA (Authentication, Authorization and Accounting) system. An EAP session is a set of (server) requests and (client) responses; at the end of this exchange the server delivers either a success or a failure message. Upon success both EAP entities compute a shared secret referred as the AAA key. EAP messages are transported either by non IP protocols like EAPoL [5] or PKM-EAP [14] on the wireless media (between the access point / base station and the client), or by routable protocols such as RADIUS [8] or DIAMETER [17] over the Internet network (between the access point / base station and the AAA server).

J. Domingo-Ferrer, J. Posegga, and D. Schreckling (Eds.): CARDIS 2006, LNCS 3928, pp. 165 – 178, 2006.

In this paper we introduce smartcards associated to EAP clients and EAP servers. Section 2 shortly reviews basic services that must be supported by EAP cards. Section 3 describes the software architecture of an open implementation for Java cards. Section 4 presents experimental results with five kinds of Java cards. Section 5 defines EAP servers for Java cards and gives early results. In the last section we introduce the Trusted EAP Module (TEAPM), an innovative architecture that will improve usage and remote management of smartcards.

2 EAP Java Cards Issues

They are two main issues concerning the use of smartcards for computing the EAP protocol: the protocol complexity and the computing speed.

Protocol complexity must be compatible with Java cards computing resources. As an illustration, byte code size is about 20 kB for EAP-TLS implementation, and 16 kB for EAP-AKA implementation; that is well-suited with today Java cards characteristics.

The need for Java cards performance estimation is not new, for example *a performance comparison of Java cards for micro payment implementation* was discussed in [4]. More recently, an initiative for *open benchmark for Java card technology* has been launched by [12], in order to setup a missing and useful tool for the smartcards industry. A study of Java cards performances has been recently presented in [18].

In our approach we classify processing operations in three categories: data transfer, cryptographic operations, and software overhead. So, the processing time of an application can be written like:

$$T_{\text{Application}} = T_{\text{Transfer}} + T_{\text{Crypto}} + T_{\text{SoftwareOverhead}} \, . \tag{1}$$

Our applications embed test functions that are used to identify critical parameters.

2.1 Data Transfer

In protocols dealing with X.509 certificates like EAP-TLS, several kilobytes of data are sent/received to/from the smartcard. Due to the lack of RAM memory, these information are written or read in the non-volatile memory (E^2PROM, flash memory,…). Therefore we call data transfer, the time required for transporting EAP packets between a terminal that controls the smartcard and the application running in this device. From a practical point of view it is easy to measure this time, but we shall not try to estimate the different elements that make up this value, like transfer delays between terminal and reader, transfer duration between reader and smartcard, internal software delays (introduced for example by java operations) and time consumed by memories accesses (writing and reading). Named T_{Transfer}, it is expressed like:

$$T_{\text{Transfer}} = T_{\text{TransferReader}} + T_{\text{TransferSmartcard}} + T_{\text{SoftwareOverhead}} + T_{\text{MemoriesAccesses}} \, . \tag{2}$$

Figure 1 presents transfer characteristics measured with four smartcards labeled A, B, C and D; the reader is the same excepted for device A which integrates an USB interface. Reading and writing operations (respectively from and to a smartcard) require similar times. The transfer law is quite linear (T_{Transfer} = *a* + *b* x *Length*) with *a* corresponding to a factor around 50 ms and *b* to a factor around 0.6 ms/byte for A and D devices, and 1.7 ms/byte for B and C devices.

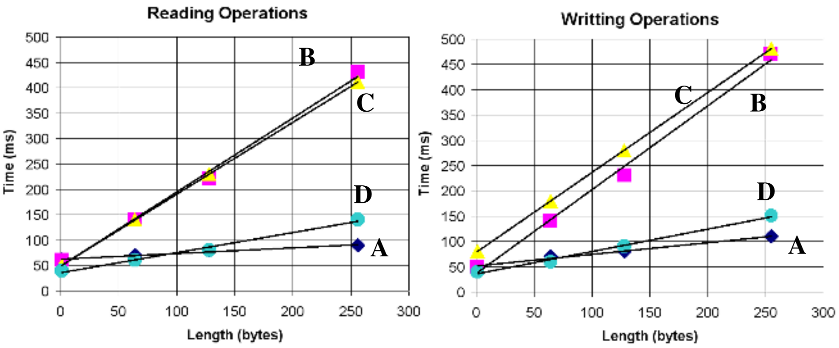

Fig. 1. Measured times for reading and writing operations

2.2 Cryptographic Operations

In a Java card context, cryptographic functions are invoked via specific APIs. For the authentication methods studied in this paper, the main cryptographic procedures are MD5, SHA1, RSA and AES.

Figure 2 shows MD5 and SHA1 speed; the time required by a digest operation is proportional to the number of computed blocs whose size is 512 bits. The time required by bloc is respectively (by alphabetical device name order) 15.3 ms, 8.5 ms, 10.2 ms, 3.0 ms for MD5, and 33.2 ms, 14.8 ms, 17.3 ms, 4.4 ms for SHA1. Because smartcards are usually optimized for RSA functions, these operations are rather “fast”. During the TLS protocol, three RSA calculations are performed: firstly during server certificate checking (public key decryption), secondly for pre-master key encryption (public key encryption), and thirdly for client’s authentication (private key encryption). As demonstrated by Table 1, these procedures consumed less than 500 ms.

In our experiments we only get one smartcard (device E) that supports the AES algorithm. We observe for this device a computing time of about 11.3 ms per bloc of 128 bits.

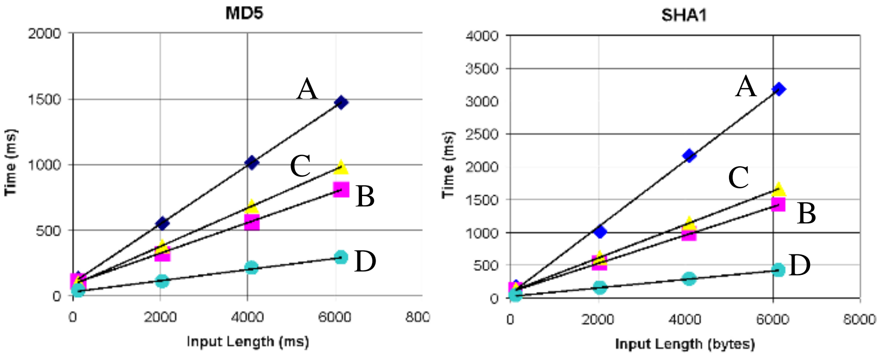

Fig. 2. Computing times for MD5 and SHA1 digests

Table 1. Estimation of RSA computing times

	RSA (1)+(2)+(3) ms	Private Key Encryption (1)	Public Key Decryption (2)	Public Key Encryption (3)	Private Key Decryption (4)
A	320	230	50	40	220
B	320	160	110	50	230
C	322	191	61	70	200
D	150	110	20	20	120

2.3 Software Overhead

All resources that are not available through APIs are supplied by the embedded (Java) application. This includes extra software needed for packets analysis, messages construction, additional cryptographic services like keyed MAC (HMAC), pseudo random functions (PRF), or some specific services like X.509 certificates parsing.

2.4 Performances Issues

The timing constraints induced by smartcards usage in wireless environments are linked to EAP and DHCP [21] protocols requirements.

On the authenticator side, the EAP server sends requests and waits for responses before a timeout; and this waiting time called *txPeriod* lasts [5] 30 s by default (with 3 retries). If the smartcard computing time exceeds this value a retransmission occurs. On Windows platforms, DHCP is a parallel event, independent of EAP authentication, that starts once network interface comes up. If the IP client doesn't receive a DHCP acknowledgement within a reasonable period of time, usually 60 s, the terminal OS resets the network interface, and therefore restarts both DHCP and EAP processes.

In summary the two main timing requirements are:

- computing an EAP request in less than 30 s, *and*
- processing an authentication scenario in less than 60 s.

This last value also includes the time consumed by the user to enter, if necessary, its PIN code.

3 OpenEapSmartcard

The basic idea behind an open platform [16] [20] is to define a simple Java framework (whose APDUs interface is described in [15]) working with most of commercial Java cards, and supporting as many EAP methods as possible.

The software architecture comprises four Java components:

1. The EapEngine that manages several methods and/or multiple instances of the same one. It implements the EAP core, and acts as a router that sends and receives packets to/from authentication methods. At the end of authentication

process, each method computes a master cryptographic key (*AAA Key*) which is read by the terminal operating system.

2. The Authentication interface that defines all mandatory services in EAP methods, in order to collaborate with the EapEngine. The two main functions are *Init*() and *Process-Eap*(). First initializes method and returns an Authentication interface; second processes incoming EAP packets. Methods may provide additional facilities dedicated to performances evaluations.
3. The Credential objects that, each one associated to a method, encapsulate all the information required for processing the given authentication scenari.
4. The Methods that correspond to the specific authentication scenarios to process. Once initialized, the selected method analyses each incoming EAP request and delivers corresponding response.

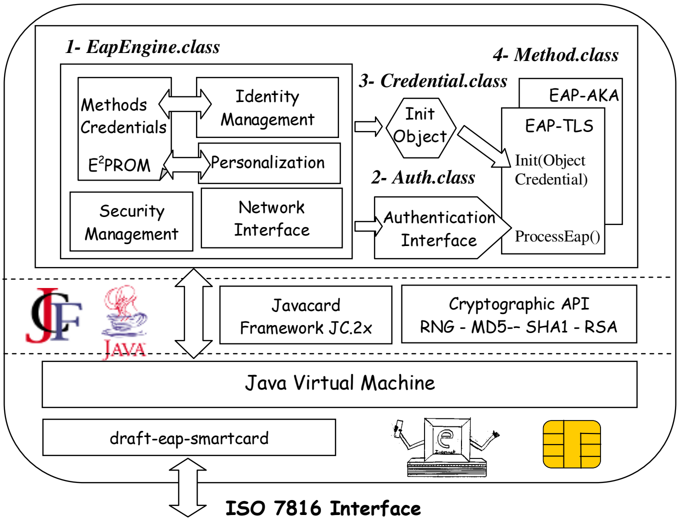

Fig. 3. OpenEapSmartcard software architecture

Due to the Java language universality, we could hope that the same code works with all smartcards; the reality is a little bit different because almost all devices present minor differences or even bugs. Here is a brief description of some observed interoperability issues:

- in TLS, the RSA algorithm is issued in conjunction with the PKCS#1 padding rules. Sometimes this functionality is not available (only NO_PAD option is working), and therefore an additional Java code is required;
- digests functions use *Update*() function for digest updating and *DoFinal*() procedure for digest closing. Sometimes *Update*() is not supported, and therefore it is necessary to concatenate all data in the non-volatile memory, in order to compute the output value;
- in some cases we observed erroneous values produced by the *Update*() method invoked with a “long” (a few thousand bytes) input value;

- with some components it is only possible to deal with one instance of MD5 or SHA1 object. As a result an interoperable application can only use one instance which implies multiple writings in non-volatile memory, so that the performances decrease (TLS produces three MD5 and SHA1 calculations).

Our EAP-TLS method takes into account these constraints. It works with RSA algorithm with no padding byte; it is compatible with single digest instance, and manages bugged or missing *Update*() methods.

4 Experimental Results

The same EAP-TLS application, including minor adaptations dealing with devices particularities (detailed in section 3), was downloaded in our four different Java cards A, B, C and D. EAP-AKA was tested with device E only.

4.1 With EAP-TLS

EAP-TLS [2] is a transparent transport of the TLS protocol [1] which has two working modes (see figure 4). The first one, referred as *full mode*, is asymmetric and uses a mutual authentication based on RSA, and that requires certificates exchange for both server and client. The second mode qualified *session resume* works according to a symmetric scheme and deals with a shared secret, the *master secret* computed during a previous full session identified by a *session-id* parameter. A detailed analysis of the EAP-TLS application was described in [11].

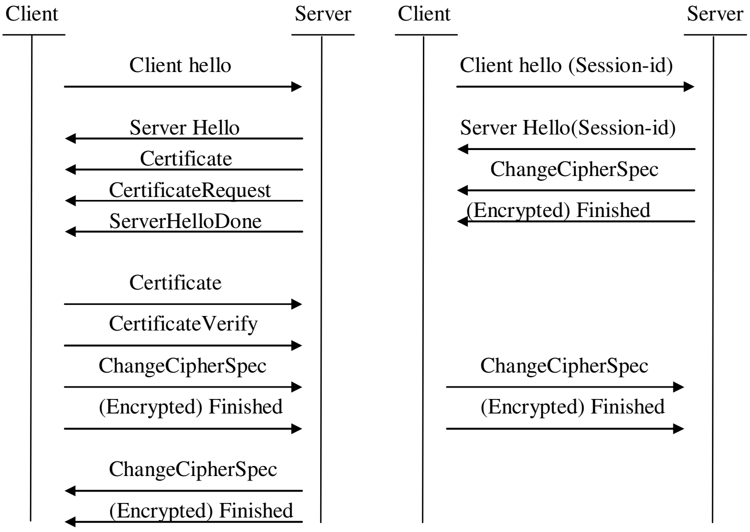

Fig. 4. TLS message exchange, *full* mode (left part) and *session resume* mode (right part)

With 1024 bits RSA keys, a *full* mode typically has the following characteristics:

- 2500 bytes of information are exchanged between the TLS client and the TLS server, for the duration of $T_{Transfer}$;
- three RSA calculations are performed (public key decryption, public key encryption and private key encryption) and need a total time T_{RSA};
- approximately 266 blocs of 512 bits are processed by MD5 and SHA1 functions. If we call T_{Digest} the average time for computing a bloc (T_{MD5}/2+ T_{SHA1}/2), these calculations cost 532 times $T_{Digest.}$;
- other operations, like X.509 certificate parsing, EAP and TLS messages processing are handled by Java procedures and consume a time $T_{SoftwareOverhead}$.

Because all cryptographic resources are seen from a practical point of view as APIs, we called T_{Crypto} the time consumed by these facilities and expressed it as:

$$T_{Crypto} = T_{RSA} + 532 \text{ x } T_{Digest} . \tag{3}$$

As a consequence, the time spent in EAP-TLS computing named $T_{EAP\text{-}TLS}$ can be split in three categories according to the following formula:

$$T_{EAP\text{-}TLS} = T_{Transfer} + T_{Crypto} + T_{SoftwareOverhead} . \tag{4}$$

Total computing time ($T_{EAP\text{-}TLS}$) and data transfer duration ($T_{Transfer}$) are obtained by direct measurements. T_{Crypto} is deduced from basic parameters presented in section 2.2. So, $T_{SoftwareOverhead}$ value can be deduced as:

$$T_{SoftwareOverhead} = T_{EAP\text{-}TLS} - T_{Transfer} - T_{Crypto} . \tag{5}$$

Table 2 presents experimental results, and a detailed comparison of Java cards performances is presented in *Appendix 1*; the reading of [11] may be useful for understanding these exhaustive comparisons.

The *session resume* mode typically presents the following characteristics:

- no RSA calculation is performed;
- 230 bytes of information are exchanged between TLS client and TLS server, which require a time called $T_{Transfer}$;
- approximately 158 blocs of 512 bits are processed by MD5 and SHA1 functions. If we call T_{Digest} the average time for computing a bloc (T_{MD5}/2+ T_{SHA1}/2), these calculations cost 316 times T_{Digest};
- other operations, like EAP and TLS messages processing, are handled by Java procedures and consume a time $T_{SoftwareOverhead}$.

Because all cryptographic resources are seen from an applicative point of view as APIs, we called T_{Crypto} all the time consumed by these facilities and we expressed it as:

$$T_{Crypto} = T_{RSA} + 532 \text{ x } T_{Digest} . \tag{6}$$

Table 2. EAP-TLS *full* mode performances

	A	B	C	D
$T_{Transfer}$ (ms)	2492	5326	5219	1433
T_{Crypto} (ms)	13221	6507	7648	2117
$T_{SoftwareOverhead}$ (ms)	62618	21914	14784	6827
$T_{EAP\text{-}TLS}$ (ms)	78331	33747	27651	10377

As a result, the time spent in EAP-TLS computing named $T_{EAP\text{-}TLS}$ is shared in three categories:

$$T_{EAP\text{-}TLS} = T_{Transfer} + T_{Crypto} + T_{SoftwareOverhead}\,. \tag{7}$$

Table 3 shows experimental results, where $T_{SoftwareOverhead}$ is deduced as previously.

Table 3. EAP-TLS *session resume* mode performances

	A	B	C	D
$T_{Transfer}$ (ms)	140	450	460	110
T_{Crypto} (ms)	7663	3675	4352	1169
$T_{SoftwareOverhead}$ (ms)	41697	19675	8688	4221
$T_{EAP\text{-}TLS}$ (ms)	49500	23800	13500	5500

4.2 With EAP-AKA

EAP-AKA [13] is a quite transparent transport of the Milenage algorithm [7]. A full authentication session is made of one request and one response. The request message which is 68 bytes long includes three attributes: a random number RAND (16 bytes)

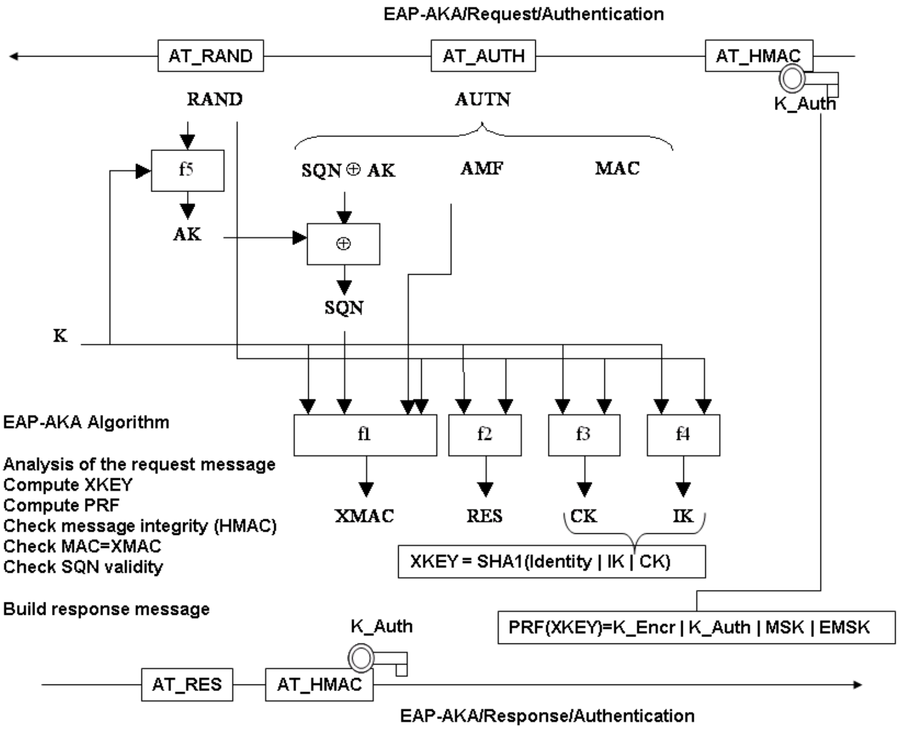
EAP-AKA/Request/Authentication
AT_RAND
AT_AUTH
AT_HMAC
K_Auth
RAND
AUTN
f5
SQN ⊕ AK
AMF
MAC
AK
⊕
SQN
K
EAP-AKA Algorithm
Analysis of the request message
Compute XKEY
Compute PRF
Check message integrity (HMAC)
Check MAC=XMAC
Check SQN validity
Build response message
f1
f2
f3
f4
XMAC
RES
CK
IK
XKEY = SHA1(Identity | IK | CK)
PRF(XKEY)=K_Encr | K_Auth | MSK | EMSK
K_Auth
AT_RES
AT_HMAC
EAP-AKA/Response/Authentication

Fig. 5. EAP-AKA, Full authentication summary

an authentication value AUTH (16 bytes) and a HMAC-SHA1 trailer (20 bytes). Upon success, the response message whose length is 40 bytes returns two attributes: a signature RES (8 bytes) and a HMAC-SHA1 trailer (20 bytes). This exchange is summarized in figure 5.

The f_i functions (f_1 , f_2 , f_3 , f_4 , f_5) are invoked by the EAP-AKA application, and imply 5 AES calculations. HMAC-SHA1 requires processing of 9 blocs of 512 bits, while the XKEY estimation costs 4 blocs. The pseudo random function (PRF) works with a modified version of SHA1 using a null padding bytes algorithm; the production of 100 bytes requires the calculation of 5 blocs of 512 bits each. Because current versions of Java cards do not support this modified version of SHA1, the procedure is fully written in Java and generates an important software overhead.

In summary the EAP-AKA cost is given by the following expressions:

$$T_{\text{EAP-AKA}} = T_{\text{Transfer}} + T_{\text{Crypto}} + T_{\text{SoftwareOverhead}} \,. \tag{8}$$

$$\text{with } T_{\text{Crypto}} = 5 \text{ x } T_{\text{AES}} + 18 \text{ x } T_{\text{Digest}} \,. \tag{9}$$

But according to our full software implementation of the PRF function (which computes five modified SHA1 values), we get the formula:

$$T'_{\text{Crypto}} = 5 \text{ x } T_{\text{AES}} + 13 \text{ x } T_{\text{Digest}} + T_{\text{PRF}} \,. \tag{10}$$

Table 4. Experimental EAP-AKA performances for device E, T_{Digest} = 4.8 ms, T_{AES} = 11.3 ms

$T_{\text{EAP-AKA}}$ (ms)	T_{Transfer} 108 bytes (ms)	5 x T_{AES} *f1…f5* (ms)	13 x T_{Digest} HMACs and XKEY (ms)	T_{PRF} (ms)	$T_{\text{SoftwareOverhead}}$ (ms)
5950	<190	56	64	5650	>0

As shown in table 4, most of computing time is consumed by the PRF function. EAP-AKA should be very efficient, if this function was available via a cryptographic API. Under this hypothesis, the authentication time should be less than 350 ms.

5 EAP Server

According to the EAP protocol, clients process requests which are issued by servers. From a software point of view, the EAP server application is very close to the client one. The cryptographic load is quite the same, but messages processing is significantly different. As illustrated by figure 6, we designed [19] a first EAP-TLS server. This server works with real network, but needs a specific RADIUS implementation, that dispatches EAP messages encapsulated in RADIUS packets, to one or more EAP-Server smartcards. In this architecture EAP data are transported by various layers (802.11 frames, RADIUS), but the authentication dialog directly occurs between two EAP smartcards, acting as SAM (Secure Authentication Modules) components.

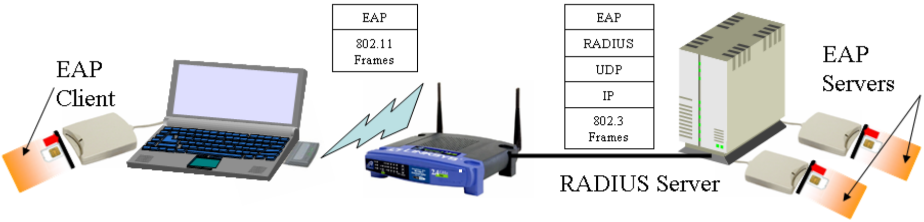

Fig. 6. EAP-Server deployment in real networks

Table 5. Comparison between EAP client and server performances

	B	D
$T_{EAP\text{-}TLS}$ Client (s)	33.8	10.4
$T_{EAP\text{-}TLS}$ Server (s)	45.2	13.0

Table 5 presents measured performances for B and D devices which are used alternatively as clients and servers. We observe that EAP-TLS servers require an additional time of about 30%. We attribute this difference to extra information transfers from E^2PROM to E^2PROM, needed for messages construction or data concatenation, induced by digest operations.

6 The Trusted EAP Module - TEAPM

Following the results obtained firstly about smartcards performances and secondly concerning OpenEapSmartcard environment for security improvement, and according the perspective of the future advances in smartcard technologies relatively to Moore's law, we suggest a new protocol stack which transforms the usage of smartcards by changing them to a kind of secure electronic pocket deposit box remotely manageable: the Trusted EAP Module.

As it appears in figure 7, EAP protocol and EAP-TLS or other EAP methods represent the heart of this protocol stack. Their presence make possible the mutual authentication establishment which can be followed by a secure exchange and storage of credentials like keys, certificates, account numbers, passwords, profiles, … in the OpenEapSmartcard-based smartcard. In this way, we offer to the users an pocket electronic component which functionally looks like the immutable TPM developed by TCG for trusted computing platforms [22].

With the ISO 7816-4 presence on the one hand of the application layer, we maintain the opening platform aspect by keeping compatibility with existing smartcard applications that use APDUs.

Finally, the choice of HTTP 1.1 and XML protocols on the other hand of the application layer welcomes the development of Web services, on either client side or/and server side.

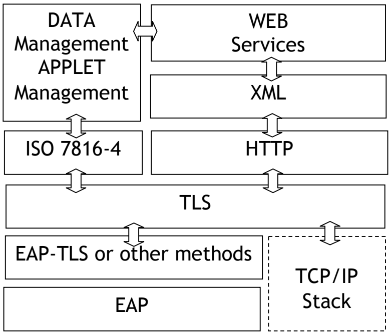

Fig. 7. The protocol stack of TEAPM

The implementation and test of this new platform on Java cards are going on. Our wish is to try later the same implementation on a SIM card, and right now nothing prevents from doing it. Surely, this will extend the capacities for secure remote management of services using smartcards, the "air" interface like in GSM network [23] [24], and Web services.

7 Conclusion

In this paper we have described a software architecture for EAP smartcards and experimental performances obtained with five devices. These results clearly demonstrate that today smartcards may be successfully introduced for enhancing security in emerging wireless networks. However authentication delays are yet very important in comparison with classical software solutions, probably because firstly some Java cards APIs are missing, and secondly more powerful components are needed, specially for EAP server. The lack of RAM memory leads to a slowdown of data storage in E^2PROM, for protocols that exchange several kilobytes of information, like TLS. But this architecture is working with standard Java cards, and it seems likely that performances will follow the Moore's law, and therefore that EAP smartcards will be more and more a credible alternative to traditional software. It is the reason why we propose the Trusted EAP Module, which will facilitate remote management and usage of network security services.

References

1. RFC 2246, “The TLS Protocol Version 1.0”, January 1999.
2. RFC 2716, “PPP EAP TLS Authentication Protocol”, B. Aboba, D. Simon. October 1999.
3. Institute of Electrical and Electronics Engineers, “Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”, IEEE Standard 802.11, 1999.
4. J. Castellà, J. Domingo-Ferrer, J. Herrera-Joancomartí, J. Planes, “A Performance Comparison of Java Cards for Micro payment Implementation”, Proceedings of the Fourth Working Conference on Smart Card Research and Advanced Applications, CARDIS 2000, September 20-22, 2000, Bristol, UK.

5. Institute of Electrical and Electronics Engineers, “Local and Metropolitan Area Networks: Port-Based Network Access Control”, IEEE Standard 802.1X, September 2001.
6. Institute of Electrical and Electronics Engineers, “IEEE Standard for Local and Metropolitan Area Networks, part 16, Air Interface for Fixed Broadband Wireless Access Systems,”, IEEE Standard 802.16, 2001.
7. 3GPP TS 35.206 V5.0.0, “3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Specification of the MILENAGE Algorithm Set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 2: Algorithm Specification”, 3GPP, June 2002.
8. RFC 3559, “RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP)”, B. Aboba, P. Calhoun, September 2003.
9. Institute of Electrical and Electronics Engineers, “Supplement to Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification for Enhanced Security”, IEEE standard 802.11i, 2004.
10. RFC 3748, “Extensible Authentication Protocol, (EAP)”, B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, H. Levkowetz, Ed. June 2004.
11. P. Urien, M. Badra, M. Dandjinou, “EAP-TLS Smartcards, from Dream to Reality”, 4th Workshop on Applications and Services in Wireless Networks, ASWN’2004, Boston University, Boston, Massachusetts, USA, August 8-11, 2004.
12. J.-M. Douin, P. Paradinas, C. Pradel, “Open Benchmark for Java Card Technology”, e-Smart’2004, Sophia Antipolis, France, September 22-24, 2004.
13. Internet Draft, .“Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA)”, draft-arkko-pppext-eap-aka-15.txt, December 2004.
14. Institute of Electrical and Electronics Engineers, “Approved Draft IEEE Standard for Local and metropolitan area networks part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems Amendment for Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands”, IEEE 802.16e, December 2005.
15. Internet Draft, “EAP-Support in Smartcard”, draft-eap-smartcard-09.txt, October 2005.
16. P. Urien, M. Dandjinou, “The OpenEapSmartcard project”, short paper, Applied Cryptography and Network Security 2005, ANCS 2005, Columbia University, June 7-10, New York, USA, 2005.
17. RFC 4072, “Diameter Extensible Authentication Protocol (EAP) Application”, P. Eronen, T. Hiller, G. Zorn, August 2005.
18. V. Guyot, “Smartcard, a mobility vector”, Phd defense, September 30th 2005, University of Paris 6, Paris, France.
19. P. Urien, M. Dandjinou, M. Badra, “Introducing micro-authentication servers in emerging pervasive environments”, IADIS International Conference WWW/Internet 2005, Lisbon, Portugal, October 19-22, 2005.
20. OpenEapSmartcard WEB site, http://www.enst.fr/~urien/openeapsmartcard
21. RFC 2131, “Dynamic Host Configuration Protocol, DHCP”, March 1997.
22. TCG, “TPM Main Part 1: Design Principles, Specification Version 1.2 Revision 85”, February 2005.
23. 3GPP TS 11.14, “Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit (SAT) for the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface”, 2003.
24. 3GPP TS 03.48, “Digital cellular telecommunications system (Phase 2+); Security mechanisms for the SIM Application Toolkit; Stage 2”, 2001.

Appendix 1 – Details of EAP-TLS Operations

EAP-TLS Message	**Operation Class**	**Smartcard**			
		A	**B**	**C**	**D**
First Message					
Request/Start	Transfer	510	601	321	151
Response/ClientHello	Transfer	30		120	20
Second Message					
Request/ServerHello, 1st fragment	Transfer	210	491	491	130
	Transfer	140	451	470	110
	Transfer	131	450	471	120
	Transfer	140	450	461	110
	Transfer	130	461	480	130
Response/ACK	Transfer	220	410	411	100
Third Message					
Request/ServerHello, 2nd fragment	Transfer	20		50	10
Response/ClientFinished					
Certificate Checking	RSA.pub.decrypt+ Other	2524	1312	931	390
VERIFY	RSA.pub.encrypt+ Other	8192	6400	1012	541
SHA1+MD5 (VERIFY)	DualHash(Verify)	1863	1121	1212	381
RSA(VERIFY)	RSA.priv.encrypt+ Other	530	361	460	261
PRF(MasterSecret)	PRF(MasterSecret)	9825	3294	3005	1162
PRF(KeyBlock)	PRF(KeyBloc)	12628	4166	3825	1472
MD5+SHA1+PRF(ClientFinished)	DualHash+PRF(Finished)	6099	2503	2524	901
MD5+SHA (ServerFinished)	DualHash(ServerFinished)	2002	1222	1322	410
HMAC-MD5	HMAC-MD5.compute	1011	451	450	251
RC4-INIT	RC4.init	5818	691	802	371
RC4-ENCRYPT	RC4.encrypt	1813	821	450	310
	Transfer	140	480	431	161
	Transfer	141	470	421	90
	Transfer	140	471	420	90
	Transfer	130	310	321	61
Fourth Message					
Request/ServerFinished					
RC4-INIT +RC4-DECRYPT	RC4.init + RC4.decrypt	7110	1292	1031	441
CHECK HMAC-MD5	HMAC-MD5.check	741	311	381	160
PRF(ServerFinished)	PRF(Finished)	4267	1332	1342	521
PRF(PMK)	PRF(PMK)	11416	3144	3685	1372

Fig. 8. Detailed EAP-TLS application performances for various smartcards

Response/ACK	Transfer	290	140	160	80
		60		50	40
Fifth Message					
GET PMK KEY	Transfer	60	141	141	30
Total Time		78331	33747	27651	10377

Fig. 8. (*continued*)