

# eSmartLock

By Yiannis Hatzopoulos  
Scientific Engineering Services  
www.ses-ltd.gr

**Abstract** : This project is a complex prototype, which demonstrates the use of a Java Card based smartcard dongle as an integrated anti-piracy module and networked DRM engine; plus offering a hardware toolkit that can enhance the security of SSL backed transactions; authenticate timeStamp receptors; function as a digital eSignature validator over commercial off-the-shelf software products - all in one: Forming an integrated system that allows users of an application to operate in a closed-user-group setting with their software producer or vendor; either online or even off-line.

Why? Apart from antipiracy security, the *eSmartLock* networked DRM model supports a diverse variety of billing options like: leasing, renting, TimeCrediting, pay-as-you-use ValueCrediting, remote feature unlock, full feature demo use. It can provide controlled crypto Web access to your eShop; or even secure CD/DVD offline content access. For high-security conscious users, *eSmartLock* can encrypt local file Save/Load operations with internal self-generated keySets; uniquely binding saved data to a specific *eSmartLock* card. It can be used in a Server - Client configuration (Trusted Third Party – Key Distribution Center), over a LAN or WAN (extranet) to authenticate other *eSmartLock* cards, establish encryption channels between *eSmartLocked* network nodes and assist the verification of signed content – all in a single JavaCard applet.

**A**nti-Piracy protection will always be a 'hot' issue in the Software industry. It is common knowledge in this business: "*there's hardly a lock that can't be unlocked*". This is mostly true for PC's; by the very fact that the exact same system has to host the protection method, validate the user's access rights and execute the application code. Any hacker that manages to capture the code in memory before its execution, can trace down the protection route, make patch calls or replay execution traffic and produce a cracked version of a software application; all in due time and effort...

*eSmartLock* belongs to a line of hardware based protection systems that can be broadly defined as 'dongles'. Dongles are connected to a USB or parallel port of a PC and function as QuestionAndAnswer-Sidekicks on the Application software. Hacking attacks on these systems are mainly focused at producing sets of drivers that replace the valid hardware dongle driver set and simulate the anticipated communication flow between application and dongle; fouling the application in to believing that all is Ok; or less frequently involve removal and patching of code that invokes dongle functions in the executable files.

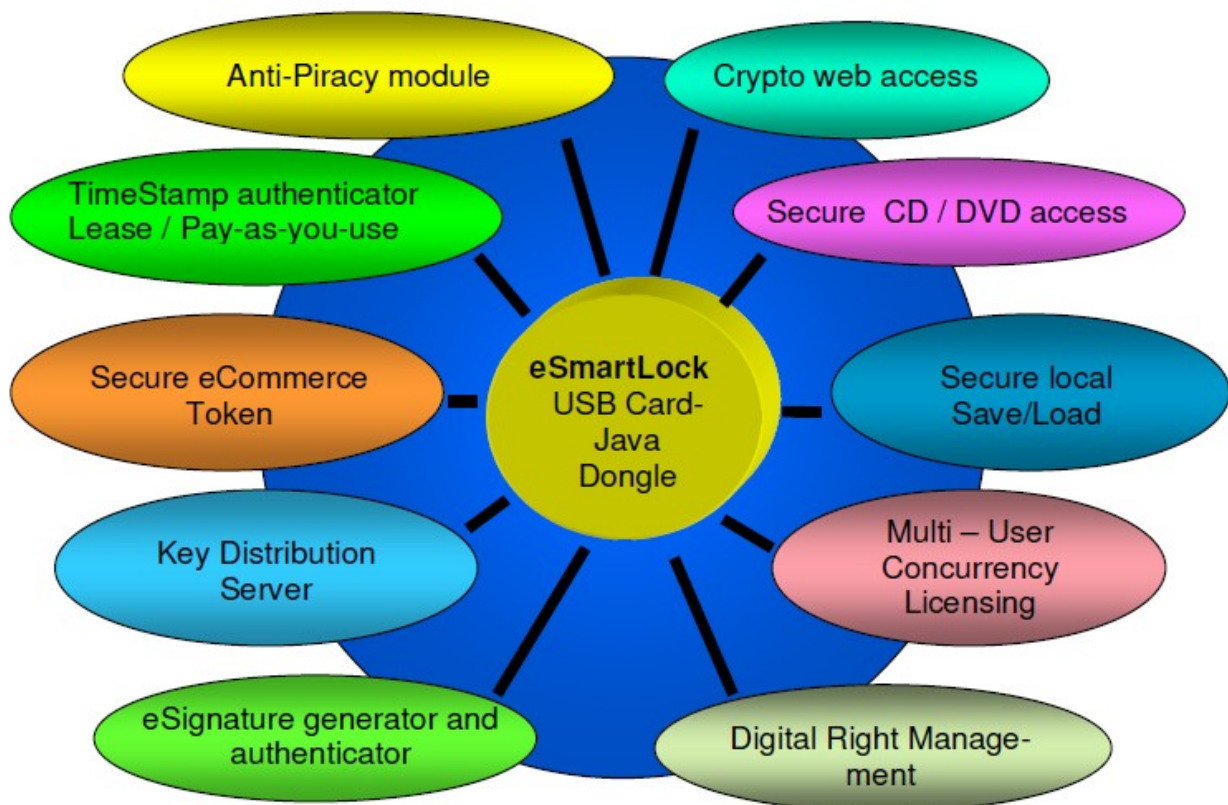
The state-of-the-art in the dongle world is smartcard dongles that either host inside them static executable code of a few kilobytes, written for instance in light versions of C or Basic( e.g Rocky5 or BasicCard) , or systems that encode executable code snippets at the developer site; running it dynamically piece-by-piece in a secure way inside the dongle (eg Sospita). These are quite secure, but since any dongle's processing speed is far smaller than the host PC, they can create a potential execution bottleneck. Moreover, the communication traffic produced may create identifiable patterns that can be exploited in playback attacks. Other approaches include smartcard file-based tokens; much like Aladdin's eToken and WIBU key, that indicate users' access-rights on the computational resources (key and certificate repositories); and also decrypt exe code segments at run-time - but do not host any executable code inside them.

*eSmartLock* is built on a standard JavaCard 2.1.1 *CyberFlex eGate* smartcard, which hosts static executable code written in java; with its inherent JavaCard spec restrictions. Nevertheless, the JavaCard environment is tamper-resistant and sufficiently flexible. The holly grail of software protection is to create a software application / system that is far less use-worthy (delivers far less functionality or ideally renders itself inoperable) when it has been illegally copied - or one that costs more in time and financial resources to crack than the marketing value it presents in the illegal software market. In the pages that follow, the approach used by *eSmartLock* to meet this difficult challenge will be analyzed.

***eSmartLock.java*** is a JavaCard applet that implements the project's functions. Upon personalization each *eSmartLock* card is given a unique Card ID (8 Bytes), which can be comprised of a serial number and an attribute set ( 4 + 4 bytes). All the necessary sets of secret keys are registered inside the card at the software producer's site by a personalization software application; before being shipped to the customer. Please note the description that follows is a descriptive rendering on the undergoing functions; and cannot substitute the java coding, that implements it in detail.

## Applet authentication and session ticketing

Security in *eSmartLock* is implemented in several layers; the first layer is a mutual cross-authentication via knowledge of a shared secret between Card and PC.



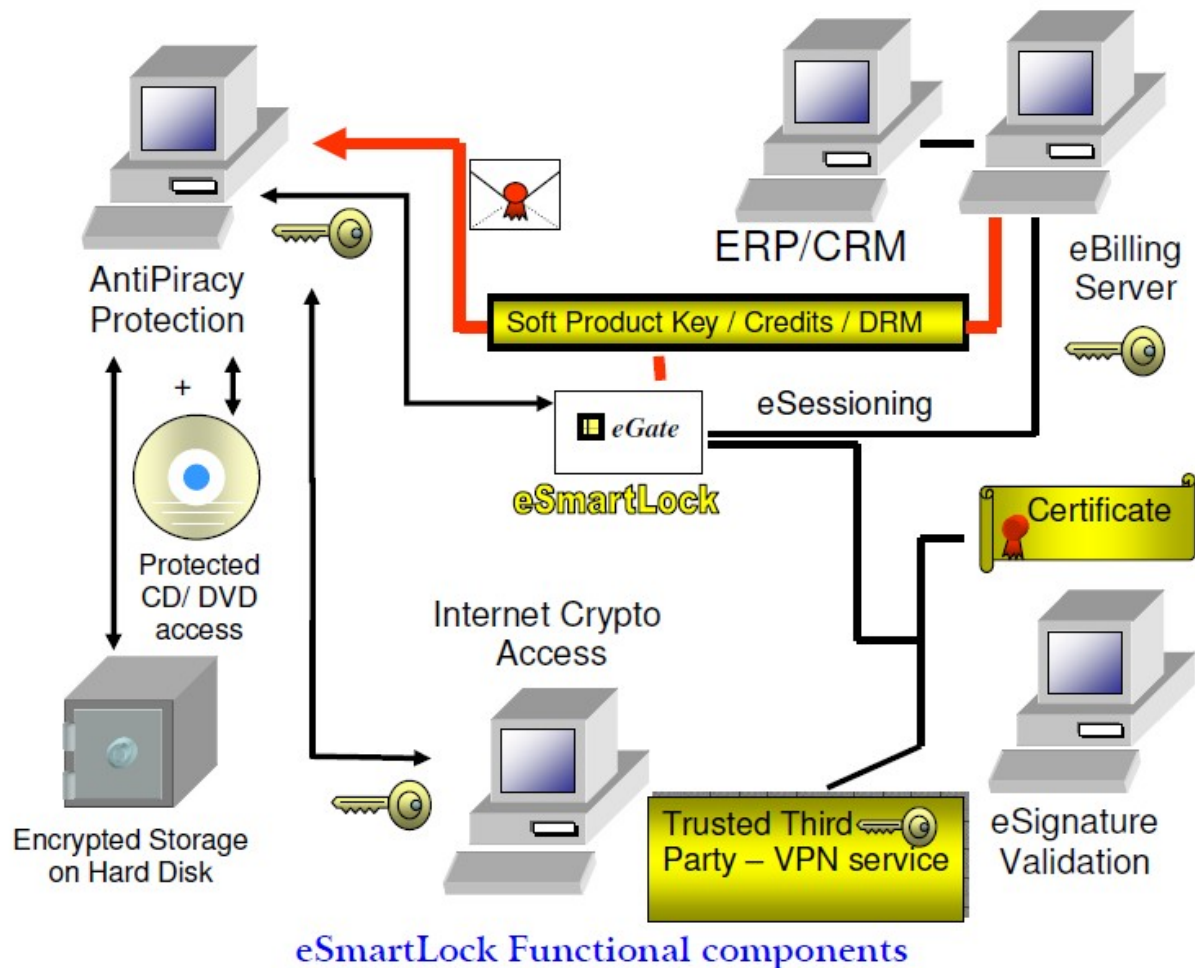
**eSmartLock API based service delivery**

- Authentication : Encryption Challenge - PC to Card ( or Card – to –Card )

This is how it works: The PC generates a random number of 8 bytes ( **R** ) and sends it to the card for encryption with a DES key known both to the card and the PC (shared secret). The card encrypts this 8 byte number ( **R** ) with its version of the secret DES key, producing **K( R )**, and sends the result back to the PC for validation. The secret key is of-course K.

The PC decipheres this encrypted number **K( R )** using a secret DES Key locally (identification via shared secret). In multi-user applications, a LAN server equipped with a Server *eSmartLock* card can be securely used for this validation ( card –to – card authentication). In either way, if the deciphered result **K<sup>-1</sup> ( R ) = R** matches the original random number challenged, this part of the authentication succeeds and the next phase of the authentication can proceed. The secret DES key shared by PC and Card can be a mathematical formula result over the executable code's digest, allowing the process to function as a fingerprinting service on the actual executable code.

If a remote authentication is not a feasible option (for stand-alone PCs) *eSmartLock* offers the capability of deciphering encoded pieces of the PCs memory with a secret key known only to the card. These deciphered memory blocks can be used as shared secret-pool repositories, that unlock their content only in the presence of the card– this feature forms an additional security layer on the code. From this pool, the necessary authentication shared-knowledge keys can be extracted by the PC; perhaps also combined with the.exe code digest to produce their final values, before being used in any authentication procedure. A similar authentication route is available using 3DES keys, which are stronger and function as an additional security layer.

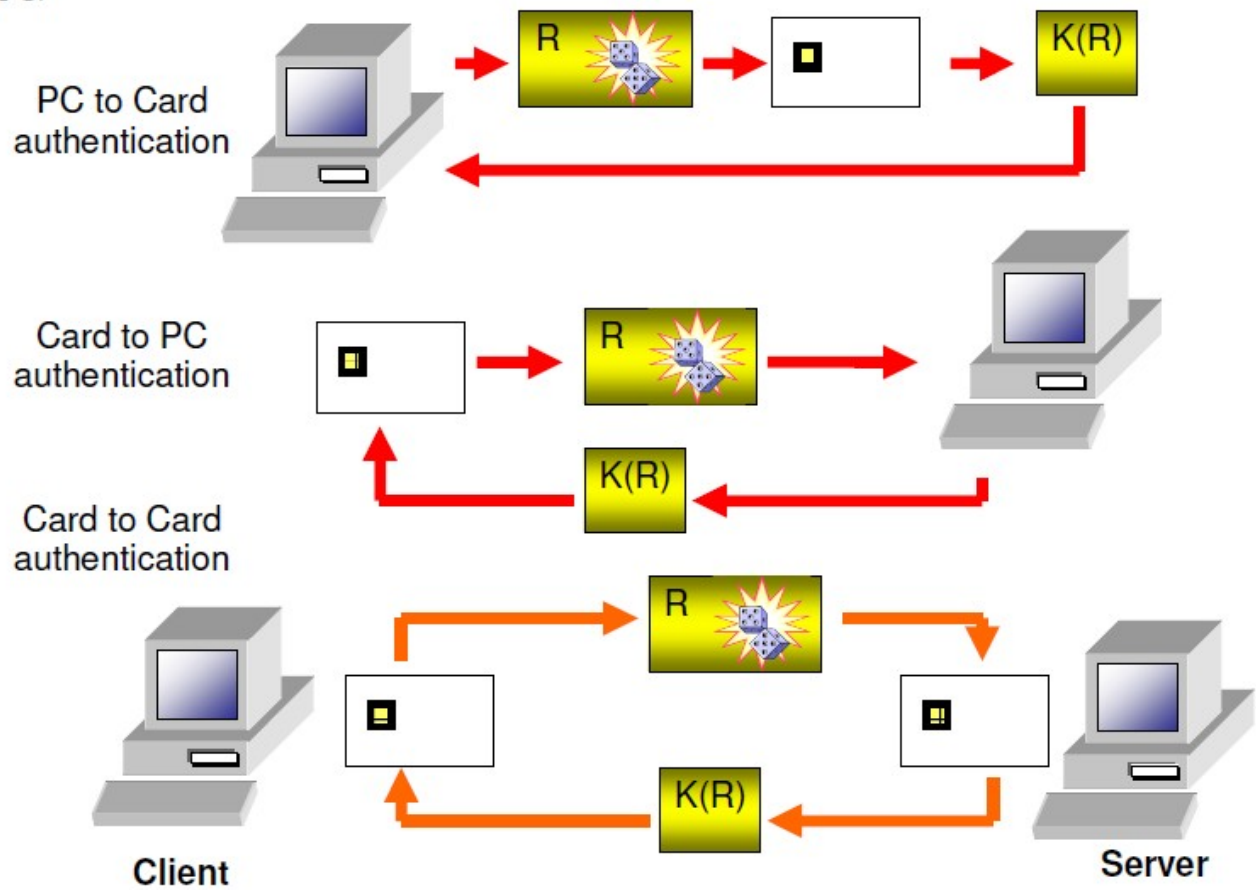


- Authentication : Decryption Challenge - Card to PC ( or Card – to –Card )**

This is the reversed authentication course (remember: this is a mutual authentication!). The card generates an 8 byte random number (**R'**) with its random number generator. It stores this number internally and gives this number to the PC for validation. The PC receives the challenge number ( **R'**) and encrypts it using a secret shared knowledge DES key, either locally or over a server service (LAN server with a Server *eSmartLock* card – which functions as a Secure Access Module - SAM ), producing a **K(R')** - an 8 byte number.

The encrypted output is given to the first card for decryption. The card internally decrypts **K<sup>1</sup>(R') = R''** and checks the outcome against the random number **R'** it presented the PC with. If **R'** and **R''** match, the authentication is a success.

A Boolean flag in the applet indicates if the mutual challenge layer has been passed. Unless successful, the card's functionality is disabled and subsequently the PC application is terminated, since it is deprived of necessary Card support to function properly. In all eSmartLock authentication layers, Card-to-Card authentication is available for Client-Server *eSmartLocked* workstations. The Server *eSmartLock* Card acts as a SAM, which authenticates Client *eSmartLocks*. All secret keys involved remain securely nested inside the cards.



### eSmartLock mutual authentication patterns

- Ticket Key generation between PC application and Card**

One of the most common attack techniques applied to dongles is simulating the communication traffic between them and the PC; fouling the application to 'think' that everything is OK. To bypass this problem, *eSmartLock* uses a Ticket key to encode all traffic between PC and Card. This ticket is discarded when the application session ends. In that respect, the communication traffic is never replicated and cannot be played back. The Ticket Key generation takes place when the *eSmartLock* applet is selected.

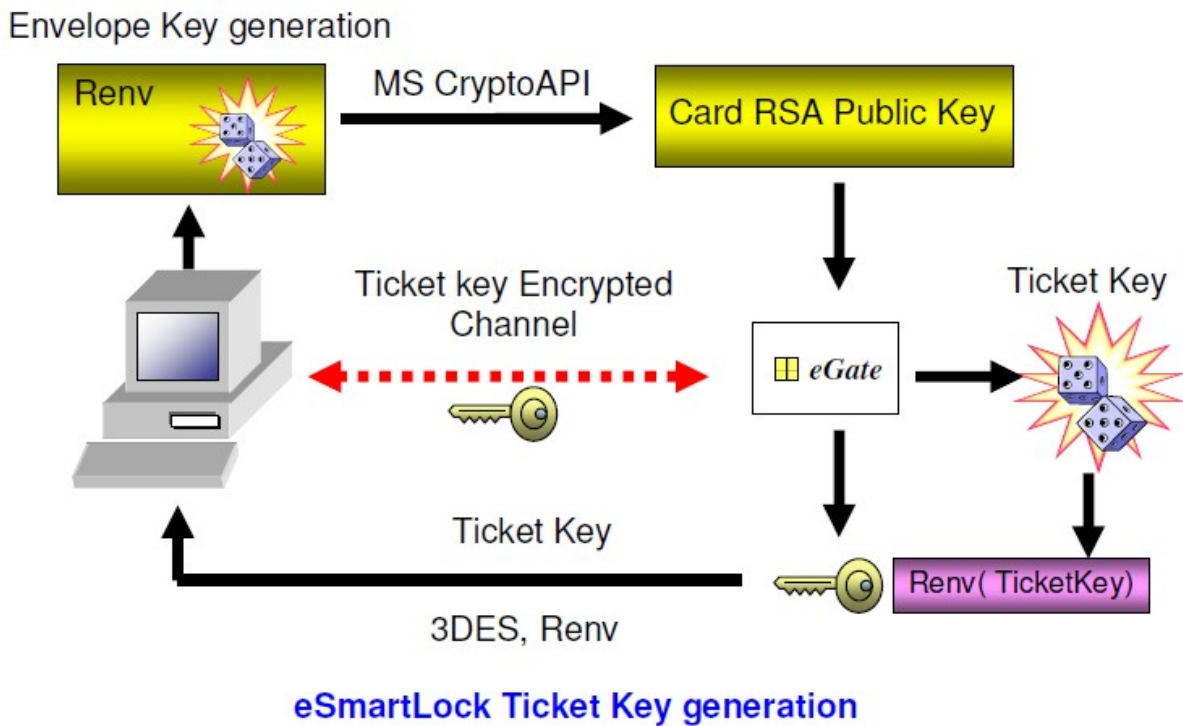
This is how it works: Using Microsoft CryptoApi and a random number generator, the PC application generates a 16 byte  $RK_{env}$  random sequence (*Random key envelope*). The PC encrypts this sequence using an RSA Public Key  $RSAPK(RK_{env}) = A$ , and forwards the encrypted data to the card; the Public Key of-course belongs to the card. The card receives the encrypted buffer, decrypts the  $RK_{env}$  using its private RSA key  $RSAPriv(A) = RK_{env}$  and generates a random 8 byte sequence, TicketKey, which will be used as a ticket key. The card encrypts (or actually envelopes) the Ticket Key using  $RK_{env}$  and a 3DES algorithm producing  $RK_{env}(TicketKey)$ . It returns the encrypted Ticketkey to the PC, which decrypts it, since it has knowledge of  $RK_{env}$ .

From that point on, all traffic between card and PC will use this TicketKey. Thus the traffic is unique each time since the TicketKey is changed on each session with an *eSmartLock* card. (Note that MS Cryptoapi does not facilitate decryption with a public key. Thus I had to use a temporary envelope key to allow the card to decide

on a ticket key and communicate it securely to the PC). No Session Key will be generated unless *eSmartLock* has passed the mutual authentication layer.

*A short note about Key Selections:* Please note that 512bit RSA cryptography was used in our example applet; which is supported by MS Base Crypto Service Providers found in all MS Windows Versions (also in WIN98). In commercial applications of *eSmartLock*, the length of the RSA keys used will be longer ( WIN XP/2000 Enhanced CryptoService Providers). The lengths of symmetric keys used in *eSmartLock* were selected to provide a fair compromise between security and card processing speed. However, in commercial versions of *eSmartLock* this length selection will be re-factored with longer keysets, in favor of security. Moreover, many important and frequent APDU commands to the *eSmartLock* are polymorphic. They do not follow a rigid CLA, INS, P1, P2 command structure; but rather respond to range checks and contain random data. The applet parses them before execution, to confuse a potential attacker who 'listens' to the traffic generated and tries to reverse engineer the system through the communication flow.

## Persistency – Card identity



The basic functions of the *eSmartLock* applet are:

- GetCardID

This command returns an 8 byte buffer, masked using the current Ticket key; it contains the Card's ID and ID attributes, as registered during the card's personalization at the vendor's site. The CardID is the unique identity of an *eSmartLock* card.

- **Write/Read Parameter File**

The parameter file is a protected, persistent unformatted byte-buffer structure, which has a size of 128 bytes in the current *eSmartLock* version. It is used as a protected repository file, where a PC application can securely store information that must be persistent. Typical examples might be date data, object IDs, encoded data, operation codes, temporary keysets, access levels, challenge data, timeStamps etc. All Read/Write traffic to the parameter file is encrypted using the current Ticket key. The data is intended to be formatted externally, using a Tag Length Value pattern. The parameter file content can be accessed on demand to unlock software features or instantiate major classes in the PC software. The Card denies access to the parameter file if the mutual authentication layer has not been cleared. If the PC application needs a more extensive Parameter File space, the file accessed will be paged in units of 128 bytes.

- **Executable Code Repository**

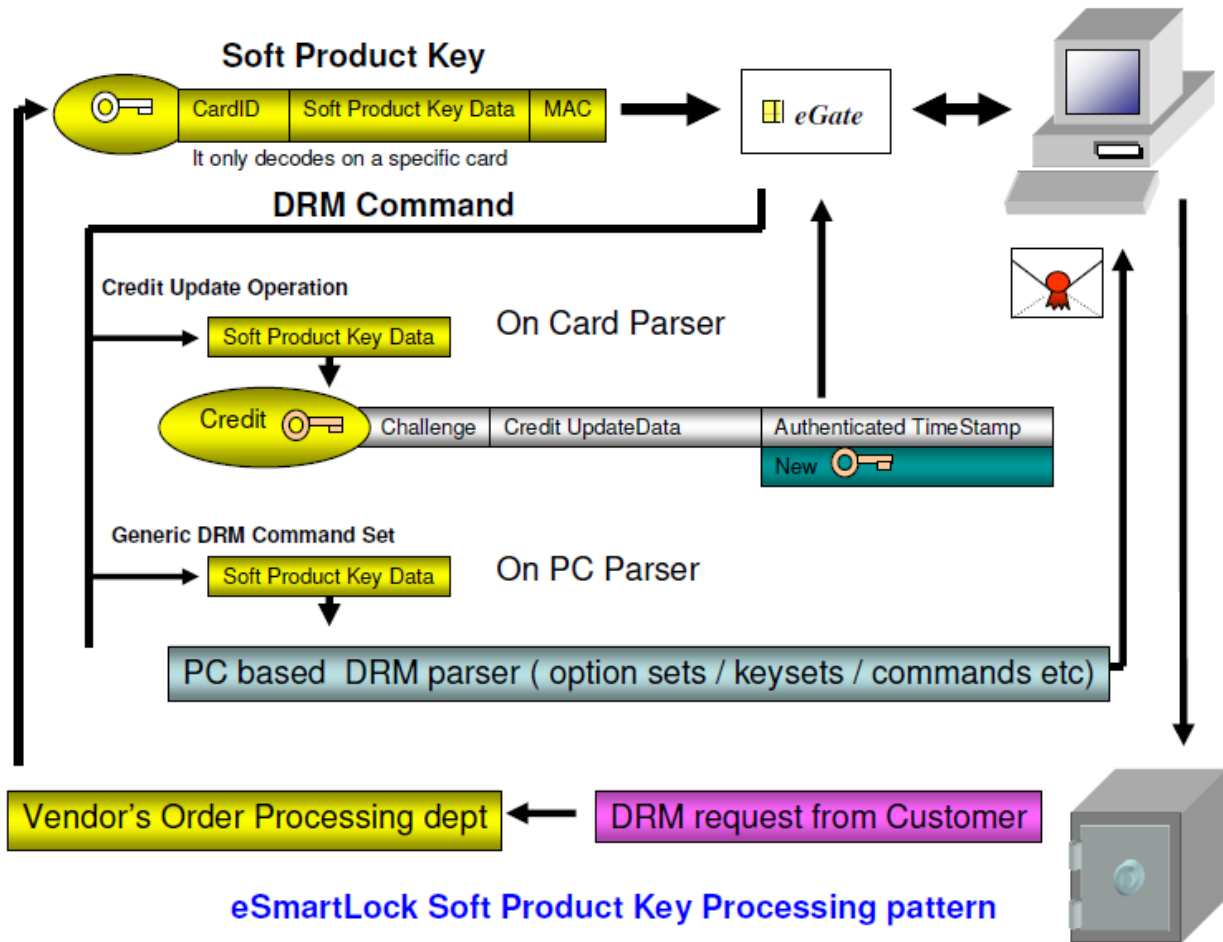
This is an additional security level over *eSmartLock*. A persistent byte buffer is used to store and retrieve executable code data belonging to the PC exe files. The data is placed back in memory before the actual code execution. This can be a dynamic process, where different pieces of the executable code can be transposed in the Card's persistent memory and back to the PC's hard disk at will. The information traffic involved is encrypted using the current ticket key. The technology used is similar to the one supporting the secure *eSmartLock* Parameter file access.

## **Digital Rights Management**

- **Soft Product Key decoding**

The Soft Product Key (SPK or simply Product Key) is an encrypted byte array, typically 24 - 128 bytes long, which can unlock or block specific features in a software application; and controls the software system's personalization and behavior. It is compiled at the order-processing-dept of the Vendor in response to a DRM request, and is subsequently sent to the Customer. The typical scenario is: The Customer receives his software package along with an *eSmartLock* card, labeled with its unique CardID ( Serial No + Card attribute set). Following the software's registration, the customer receives a Soft Product Key (eg a 48 byte number sequence), which is a match for that specific card. The Soft Product Key data is encrypted at the vendor's site, using a secret key known only to the Vendor and hosted inside the card (shared secret key).

The Soft Product Key is processed and decrypted inside that card. If the decrypted SPK's prefix matches this designated card's ID, and a MAC check clears, the Soft Product Key is exported to the PC, otherwise an exception is raised by the card and the Product Key is rejected. A copy of the last valid Soft Product Key processed is also stored inside the card and can be given to the outside world masked by the current ticket key.



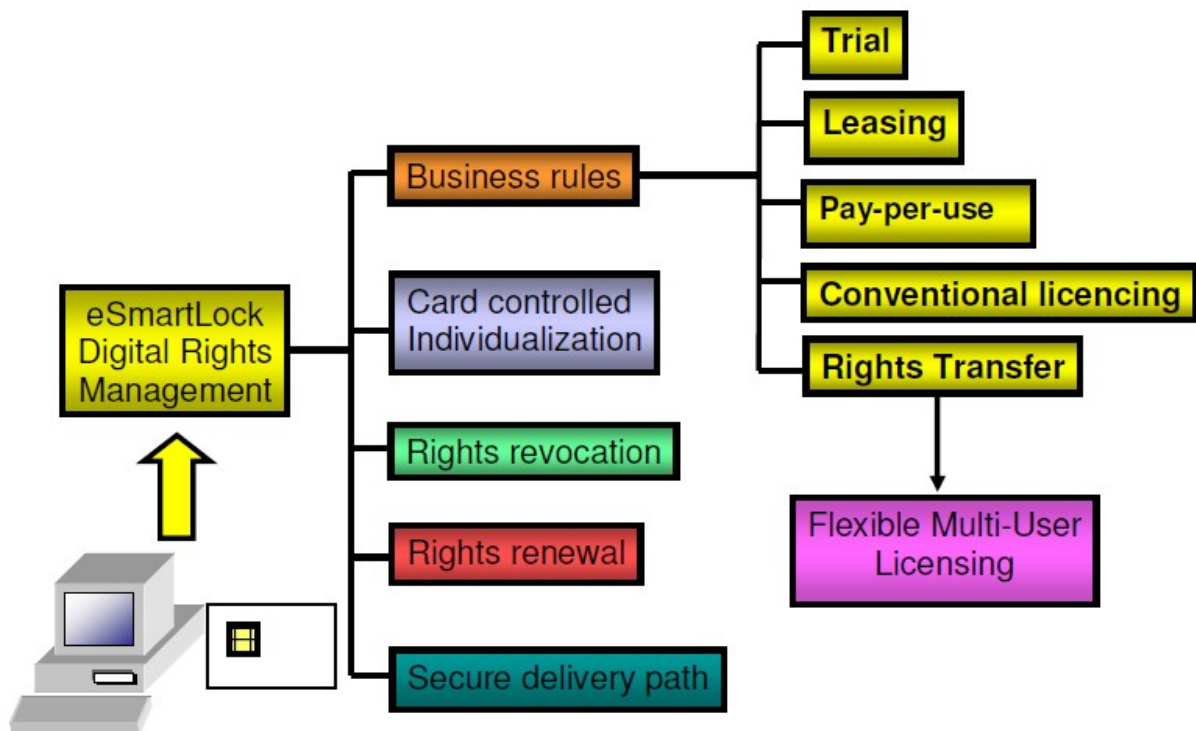
This means that a specific Soft Product Key only decodes inside one specific card, and may not be used elsewhere. Many commands that should reach the PC safely, concerning DRM functions get enveloped inside Soft Product Keys( SPK). The decrypted SPK data is parsed on the PC and subsequently the parameter file on the Card is can be safely updated. The card can be updated with time or credit value accordingly, or whatever info is needed to implement the requested business/ licensing model.

When the user requests an updated set of functionality from his application (feature unlock), a new Soft Product Key sequence is compiled by the vendor and given to the Card for validation, deciphering and processing. The process can be performed either online or even offline (eg faxed message with a new Soft Product Key sequence, an email message / eBilling session, clearance and auto-generation of a corresponding SPK). All necessary processing is out of reach for the PC application and hackers, since it is performed internally on the *eSmartLock* card.

## • Decode eLibrary Content

A secret symmetric key inside the card is used to decipher encrypted byte-buffers stored in CD/DVD or coming over a web connection (eg Database access, Voice over IP or even mp3 music). If the byte buffer size is small in size( typically couple tens of KBytes) the card can handle the decoding task internally in packets of 128 bytes; otherwise it must decode the 'track'-key necessary to decrypt the encrypted buffer externally; with this key, the bulk decryption task is handled by the PC which is far faster.

This feature is especially useful for streaming encrypted data flow, coming from an internet connection. This very same function can be used to enhance the security of the PC exe file. Data that is used to instantiate objects can be encrypted at the source level and be decrypted into functional values during the execution time. In fact, whole pieces of the exe file can reach the customer PC in encrypted layers and be decrypted by the card, just prior of their execution. Finally, a similar feature facilitates the installation of updates on *eSmartLock* applets. If the vendor needs to update the eGate applet on Customer Site by a setup application, the new applet should reach the customer PC in encrypted form and get decrypted by the card just prior to the installation. The new applet will contain the necessary keys to facilitate any subsequent update operation. The same setup file cannot update an applet more than once, because the decryption keys will get changed along with it on card.



## Time and Value Credits

- Credit Updates

This feature is invoked when a new *Value Credit* ( much like an embedded eWallet - where monetary value is debited each time the user accesses specific software features - pay-as-you-use model ) or Time Bound function-Credit is requested (leasing, renting, end of an evaluation period)

The Credit Update Request can be channeled through a web service, following an *eSmartLock* backed eBilling Session; or offline by a faxed money order / credit card number submission to the order processing dept. of the vendor. Responding to a Credit Update request by a customer, a new Soft Product Key sequence is compiled, which encapsulates all the necessary information for the specific card intended, to allow it to

release Credit, or update the card's parameter files that control the software product's execution ( Digital Rights Management support).

In the case of a Credit Update Request, this compiled Product Key ( eg 48 encoded bytes sequence) embeds inside it a CreditUpdate control sequence, which is encoded using the Current Credit Update Key (Credit Update Key is a DES or 3DES key shared by Card and Vendor) for the specific card it is intended to be applied to ( this key is different for each card and can be changed remotely and securely in the manner described bellow).

The CreditUpdate control sequence contains in encrypted form the recipient Card's CardID, an encoded Credit Challenge, encrypted new value-sets for Time Credit / Value Credit, an encrypted Date Stamp and the new Credit Update Key, which will replace the current one on the Card. When installed, this new Credit Update Key will be used for any future credit update operation. Apart from the CreditUpdate control sequence, the Soft Product Key can also contain additional encrypted fields that manipulate the card's parameter file (check the parameter file access operation).

When the Soft Product Key reaches the card, it is decoded internally and gets processed, if only the card tracks a matching CardId contained in the Soft Product Key, comparing it with its own CardId and a valid MAC. Subsequently, the CreditUpdate control sequence is extracted which allows the card to perform a check on the CreditChallenge field contained in the Credit Update sequence: The Card deciphers the CreditChallenge using its current Credit Update Key ( DES or 3DES key); and checks the outcome against a static value it expects to find; usually a constant byte stream, known to the card and the software vendor ( shared secret ).

If the CreditChallenge is passed, the new Time/Value Credits and DateStamp are decoded and replace the old ones inside the card. Finally, the New Credit Key is decoded and replaces the current Credit Update Key. A boolean flag marks the success of the Credit update operation, which will unlock all of the Card's locked functionality in the case of an eg expired time lease.

This same product key sequence cannot be used on any other card, or be used again, because the Current Credit Update Key in the card gets changed on every successful Credit Update operation; and of-course the whole sequence clears only on a specific card. It is actually a procedure that is fairly similar to an a-la-cart cellphone crediting scheme; it is cryptographically secure; and can be applied securely over insecure channels either online ( web access, email ) or even offline ( eg faxed message/ or letter to the customer). All necessary processing is performed internally in the Card, and no relevant keys exist in the PC environment.

To be efficient, the whole procedure should be backed by an order processing SQL database: one that hosts customers' contact and contract data, the corresponding Card IDs each one is using, each *eSmartLock* Card's Current Credit Update Key and the current set of enabled options (feature unlock) each client has bought or leased. Needless to say, there should be a module that compiles Soft Product Key sequences based on these data! Links can be made to either ERP or CRM modules that allow an efficient management of the system; especially if the user database is extensive and the tasks have to be automated through a secure *eSmartLock* web access or eBilling session.

- **Get Current Credit**

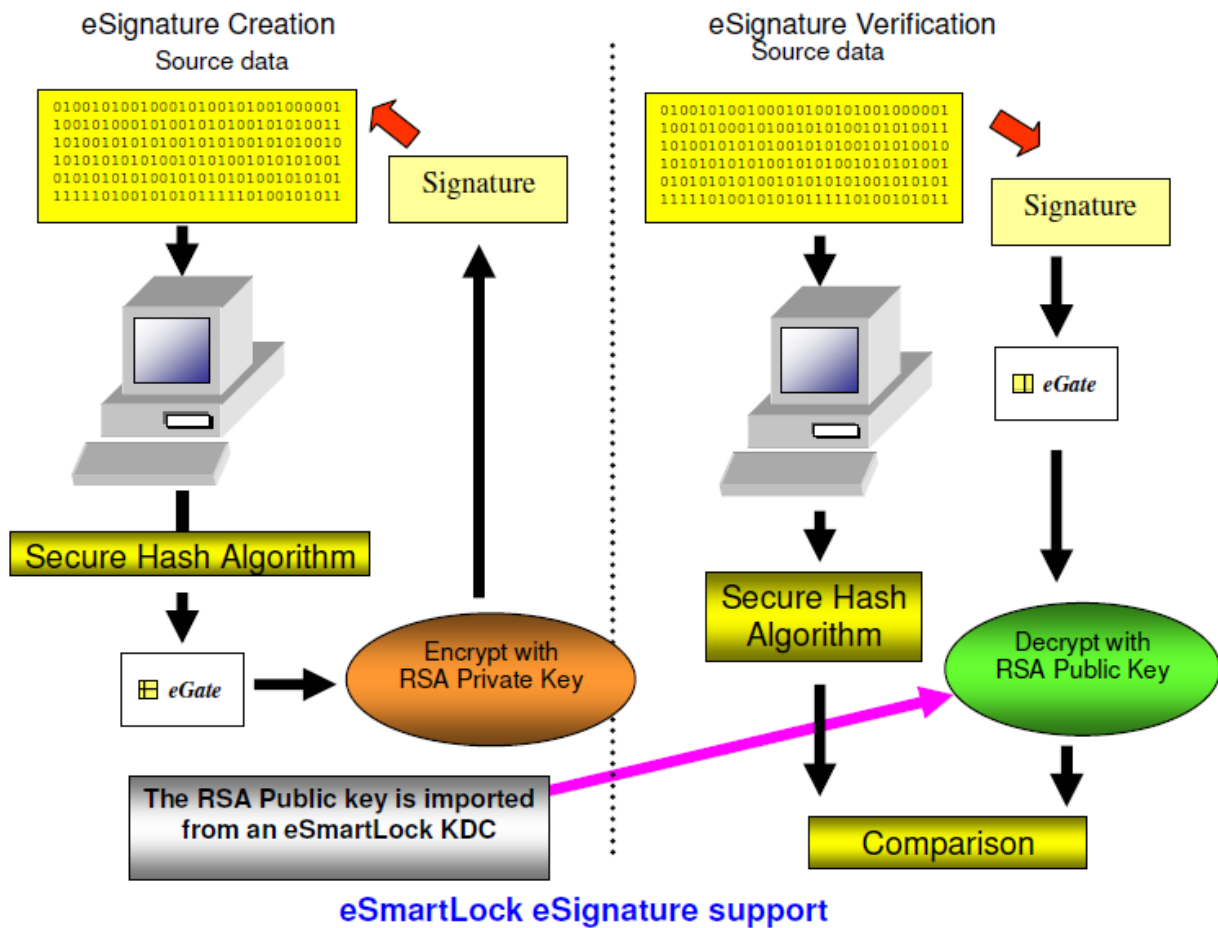
This command retrieves the Current Time and Value Credit on the Card. The communication traffic generated is encrypted using the current ticket key. In our *eSmartLock* applet, it returns 8 bytes: 2 bytes (JavaCard short number ) for Value Credits, 6 bytes (3 short numbers, T1,T2,T3) for time credit. Remaining Time Credit is attributed in the following manner.  $TimeCredit = ((JavaCard\ Short)\ T2 * (JavaCard\ Short)\ T3) + (JavaCard\ Short)\ T1$ . This pattern is used because JavaCard short is unable to host long numbers. The Value credit 2 byte Short was deemed reasonably big to host a value credit. A similar command returns the last authenticated DateStamp received by the card ( for check on end of lease period). Also masked using the current ticket key.

## • Reduce Credits

This command reduces the available Time/ Value credits by a specified byte value each time it is called. It is a polymorphic and random padded APDU command, which is being parsed by the *eSmartLock* APDU processor to make it difficult to be identified and bypassed by a hostile driver set patch. It reduces the remaining time or value credits, according to the charging model specified by the currently applied Soft Product Key.

When either value reaches zero, an exception is raised, which disables the card's functionality, only allowing it to receive a new Credit update (you can call it a credit refill). As far as TimeCrediting is concerned, *eSmartLock* allows either a date based TimeCredit (eg. allow full software functionality for 30 days; unless a new Soft Product Key has been entered into the system, fall back to a demo mode) or 'Time-Tick-Credit' ( eg allow 16 hours of full operation before falling back to a demo mode). The necessary date-data can be stored in the persistent *eSmartLock* Parameter file or the DateStamp store area; whereas the Time-Tick-Credit data is stored in the TimeCredit card buffers.

If the PC application keeps concurrent tags on the PC clock, against date log data in the persistent *eSmartLock* parameter file, the stored dateStamp on the Card and the actual TimeTick debiting that takes place during operation of the software, time spoofing attacks become quite difficult to perform.



## eSignature functions

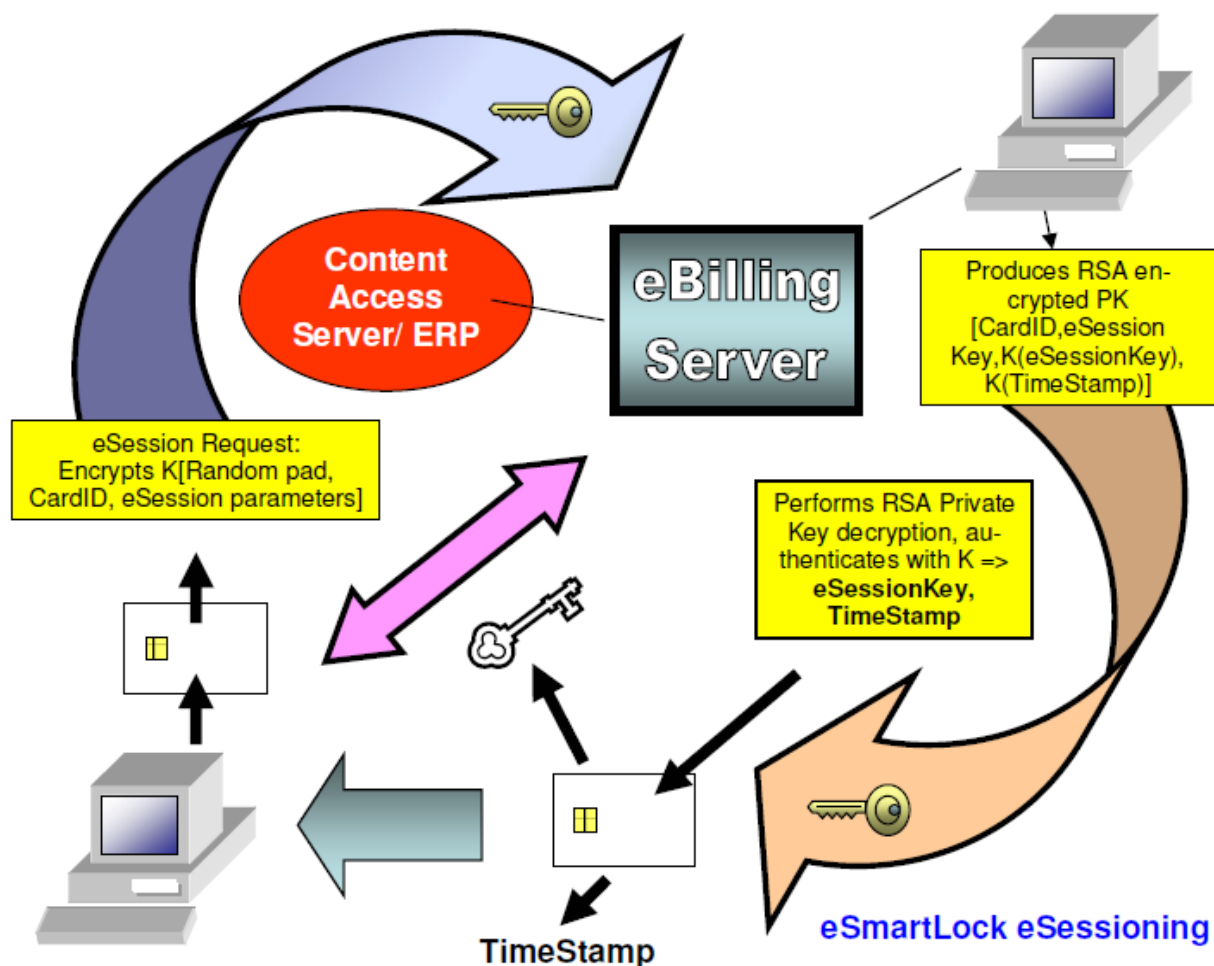
- eSignature Support.

eSignature creation and validation is a very important tool in upgrading the security status in any communicative transaction. It certifies that a byte buffer has not been tampered with by anyone, from the moment it has been created and a Signature Seal has been created on it.

This is how it works under *eSmartLock*: The information intended to be signed is contained in a byte buffer; let us call this buffer S. An MD5 ( SHA-1 / SHA -2 can be used as well – it makes no difference to the eGate Card) hash is calculated on S by the PC, which is very fast, producing a 16 byte digest (or more depending on the algorithm used ); We will call this digest D; therefore  $MD5(S) = D$ . D is given to *eSmartLock*, and gets encrypted using an RSA Private key producing  $RsaPrivKey(D) = SIG$ , which is exported to the outside world to know. This is actually the sealed signature on S. The whole information pack now becomes [S, SIG] and can be distributed freely.

The eSignature validation is quite simple. Assuming at another location an eSmaLock dongle needs to check if the [S, SIG] has been tampered with, what does it need to do?

Simple: The PC creates an MD5 digest on S, let us call this digest  $D' = MD5(S)$ . *eSmartLock* decipheres  $RSAPKdecrypt(SIG) = X$  with its public key ( decryption with a public key). The PC compares X with  $D'$ . If X is equal to  $D'$ , the PC can be confident that the information has not been tampered with; the whole process is based on RSA security and the fact that the keys used remain securely embedded inside the card. The hashing algorithm deployed is totally irrelevant to the card. Note that the hashes (and thus the algorithms used) are built on PC and not on card for the speed this approach offers; plus the fact that any length of byte buffer can be processed. The SmartCard technology offers the necessary security in handling the secret keys involved.



*eSmartLock* supports validity checks on distribution of any kind of content in electronic form. In a Client – Server *eSmartLock* system, all Clients register their public keys with the Server. Thus when Client A signs a piece of information with its private key, Client B can verify this signature by requesting A's authenticated public key from the *eSmartLock* KDC (key distribution Center – check below). In applications that do not support a KDC operation, all *eSmartLocks* belonging to the same workgroup share a common public/private key pair to assist signature checks (keyset values are being set by the personalization of each card).

***eSmartLock* Image signatures:** Future versions of *eSmartLock* will support bar-coded signature generation on image documents. The approach will involve PC image processing work on grayscale/ back-and-white document data. An image-hash will be created on the source image data. The hash will be encoded with a Private RSA Key on an *eSmartLock* card; the encoded hash data will be transformed into barcode data and will be attached under the original image. The image eSignature valuator will scan the printed document. The

scanned image will be separated from the barcode signature information. A hash will be calculated on the scanned image; the barcoded eSignature data will be decrypted by a public key on another *eSmartLock* card. The original image hash value (contained in the barcoded signature) will be compared against the hash calculated on the scanned image. The comparison will verify if the printed document has been altered, since it was signed (eg. banking/ corporate transactions/ government document workflows - tamper prevention).

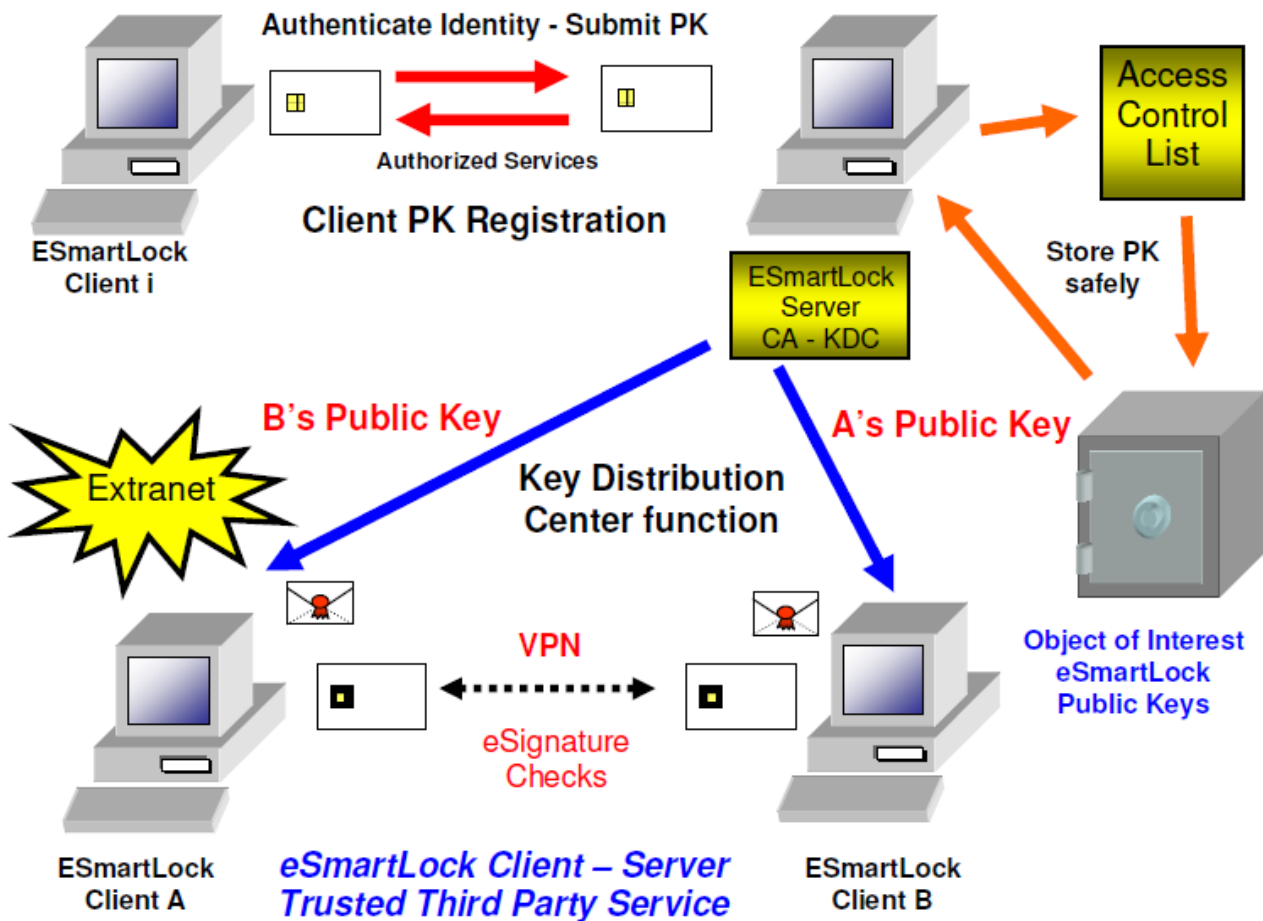
## Secure Save/Load Operations

- Secure File Save And Load operations

This is an important feature, designed especially for software that handles highly sensitive information; which is expected to support Save/Load operations on a hard disk, or on any kind of removable media: Upon instantiation of any *eSmartLock* applet, a long random number sequence is generated and stored inside the card - this is in fact a randomly selected key - we will call this sequence **KR**. Being totally random, this card self-generating keyset, never leaves the card environment and is known to this specific card and nobody else!

On Save operations, the PC application splits the digital content into packets of 100 – 128 bytes. These are given to card, which encrypts them piece by piece, using **KR** on a 3DES algorithm, returning the encrypted data sequence to the PC for external storage. At Load operations, the same process is repeated in a similar manner, with the card set at 3DES decrypt mode using **KR** as a key. The card functions as the single and only hardware key that can unlock the information; because nobody other than this exact same card that encrypted the data, has access to **KR** key. Unless the user is using this exact card, the data cannot be retrieved.

Under the *eSmartLock* Client – Server model, this operation supports the secure storage of authenticated public keys on Hard Disk, in the *eSmartLock* Key Distribution Center operation. Thus we can be certain that nobody else other than the *eSmartLock* Server card can access or manipulate the KDC public key database.



## eCommerce – eBusiness Support

- eSessioning and authenticated DateStamp reception

The eSessioning functions provided by *eSmartLock* are the basis for the internet eBilling functionality offered by this model, plus the provision of secured and controlled internet access.

eSessioning can be executed securely on *eSmartLock* over a plain internet socket connection, or can be performed over a Secure Socket Layer connection, adding an additional security layer over SSL, enjoying the interoperability of software that supports SSL (web servers, browsers). In the former case, no commercial certification authority is necessary. The applet is in full control of the secure exchanges; all the necessary secrets are self-contained in the smartcard. In the latter case, the *eSmartLock* applet must coexist with a smartcard SSL certificate container applet, instantiated by the SLB COVE application; the certificate must be provided by a commercial certification authority. eGate is a multi-application card and can contain multiple applets! One can actually enjoy the best of both worlds!

In either case, the PC application will enjoy services such as secure software downloads that replace components on the executable PC files, vendor server based software execution over a crypto channel, authorized access to knowledge bases, automatically updatable Soft Product Keys, automated credit card transaction clearance, reception of authenticated timeStamps etc. The lengths of the keys used and the supporting crypto-algorithms are under the absolute control of the Software Vendor.

This is how the whole approach functions on the card's side: Using a secret 3DES card key, let's call this eComK ( eCommerce Key - it is a shared knowledge secret key, known to card and the vendor's server ) an *eSmartLock* card encodes its own CardID and the type of access it requests from the vendor's internet server (eg eBilling, Renewal of Credits, secure eLibrary access, web site access in a protected zone etc). This sequence is prefixed with an 8 byte random pad; CBC mode encryption performed on this sequence ensures the encrypted result's uniqueness each time.

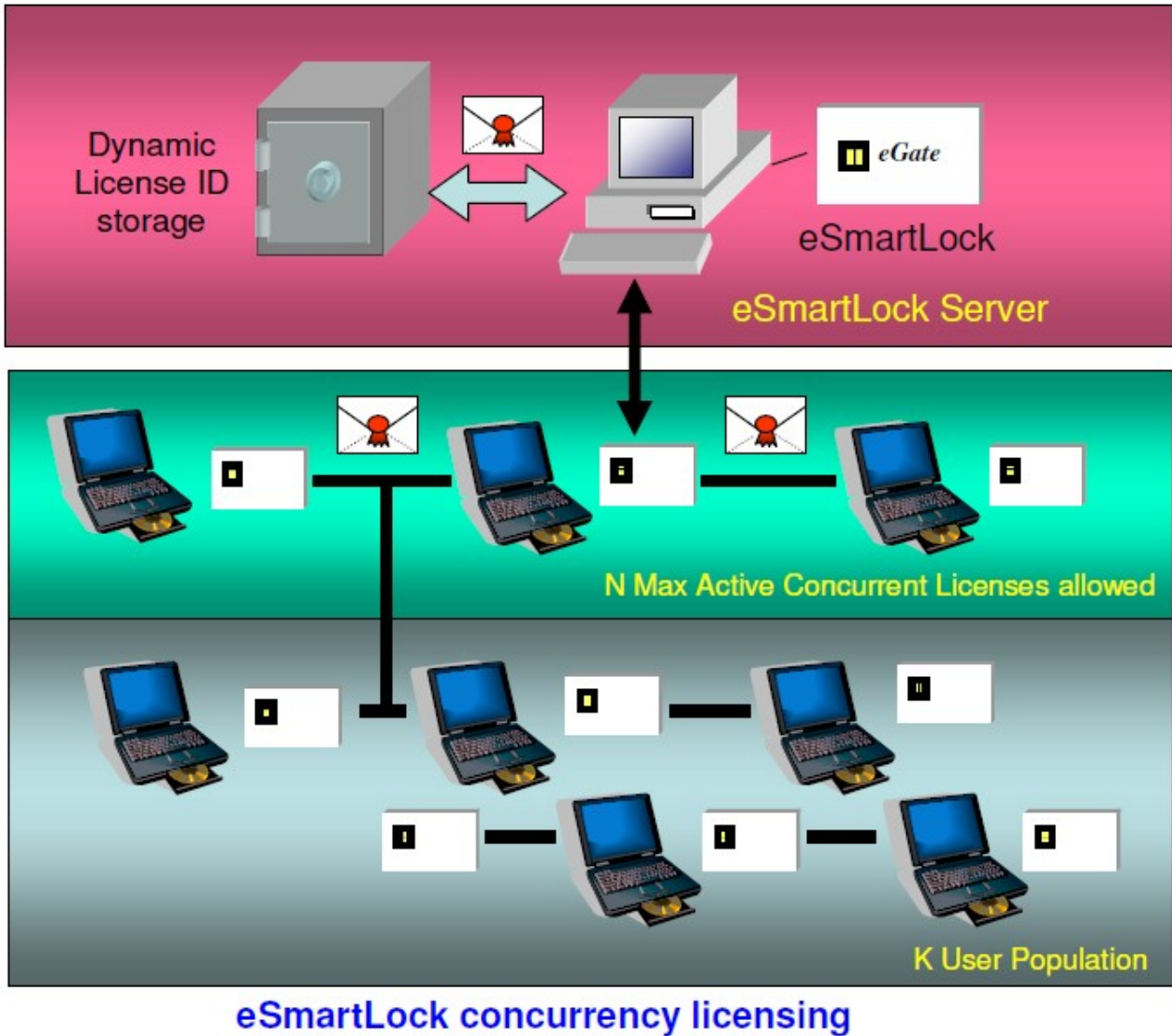
The encrypted sequence is sent to the Server. The Server deciphers the sequence, since it has knowledge of eComK ( it is a shared secret after-all). It identifies the customer using the CardID associated with him. Concurrently, checks if the access service requested is valid for this CardID, via access to an SQL customer service or an ERP module.

Assuming this check clears, the server uses the *eSmartLock* card's Public Key (this public key is known to the server – and actually common to all *eSmartLocks* that belong to the SameWorkGroup) to encode the following pieces of information: Recipient card CardID, Server Generated eSessionKey ( a random sequence actually ), a 3DES encryption of the eSessionKey with eComK; let's call this eComK(eSessionKey) and an 8 byte TimeStamp generated by the vendor server encrypted with eComK, will call this eComK(TimeStamp). This encoded information pack is sent to the card for processing.

The *eSmartLock* Card receives this RSA encrypted sequence and decrypts it using a corresponding secret private RSA key. The first check is performed on the decoded Recipient CardID; if this is a match for the card's internal CardID, the process can continue. Subsequently, eComK (eSessionKey) is decrypted; and since *eSmartLock* has knowledge of eComK, a check is made between eSessionKey and  $eComK^{-1}(eSessionKey)$ . If the comparison clears, it is valid proof that the legitimate vendor eBilling server produced this eSessionKey and not some impostor; since it had knowledge of the embedded secret. Otherwise an exception is raised and the operation is aborted.

The eSessionKey is given to the PC, which encrypts/decrypts all subsequent data exchanged with the eBilling server, adding an additional protection layer on important pieces of information, like Credit Card Numbers, Order requests, customer IDs, downloaded information etc. The authenticated TimeStamp received, supports date bound TimeCredit operations. It makes no difference if the user rolls the PC real time clock back. The PC application can easily identify if the PC Clock has been tampered with.

In the manner described, following a single request-response sequence, a cross authentication between *eSmartLock* and eBilling server takes place, and the server generated timeStamped eSessionKey can be securely identified and decrypted by the card. The PC application can submit credit card and ordering data or in-fact any kind of data to the server with a very high degree of security. This service model is complementary to all of the previous functions described so far; and implements the *eSmartLock* eBusiness model. The Customer can purchase and unlock software functionality from the Vendor in a secure and transparent way 24h/day, 7days/week.



## Key Distribution Services

- **eSmartLock** Server/Client –Trusted Third Party - Key Distribution Service

As mentioned in preceding segments of this paper, *eSmartLock* functions not only over stand-alone PCs; but also in a client - Server configuration. The Server *eSmartLock* applet can authenticate Client *eSmartLock* applets ( Card-to-Card encryption / decryption challenges). The keys involved remain securely nested inside the cards. Moreover, an *eSmartLock* server applet can provide an additional and important service: It can act as a Trusted Third Party on a network; securely distributing the authenticated public keys of client *eSmartLocked* nodes; allowing the exchange of session keys for secure point – to – point communication; or verification on the integrity of information that has been digitally signed by an *eSmartLocked* Client on the network. This is how the approach works.

Two phases can be identified: *Registration* and *Operation*.

During the registration phase, the *eSmartLocked* Server polls the *eSmartLocked* Clients and requests their public keys.

All *eSmartLocks* applets share knowledge of a ServerLock 3DES communication Key ( we will this key SLK = ServerLock Key – unlike the Daimler Benz model!), and also an secret ID, a ServerLockID which indicates that

they belong to the same Trusted Third Party 'jurisdiction'. Each client card receives or self-generates a private-public RSA key pair during the applet's instantiation process at the vendor's site.

Each Client *eSmartLock* sends to the Server an SLK CBC encryption of the following data: SLK ( Random pad, ServerLockID, ClientCard ID, ClientCard Public Key, MAC on this whole data packet). Moreover, each Client card is requested to submit its network ID ( eg workstation network identity), signed with its private RSA key.

The Server receives this sequence from each client card, decrypts it using SLK and accepts this submission package only if it tracks a matching enveloped ServerLockID compatible to its internal ServerLockID registration and a valid MAC code. Then, the *eSmartLock* Server stores this data on a database it controls, which contains the following fields: Client CardID, signed Client Network ID, authenticated Card Public Key, MAC Signature.

Moreover, these database fields are saved/loaded securely using a standard *eSmartLock* secure Save/Load operation, to guarantee that they cannot be altered by external intervention; and are only accessible through this specific ServerLock *eSmartLock* card.

During the Operation phase, assuming that *eSmartLock* Client A needs to connect to Client B, A should contact the *eSmartLock* server ( KDC - Key Distribution Center) and submit a connection request with B. The KDC, will receive this request and will scan the Public key database on the PC hard disk, where a matching CardID for this request will be sought (CardID\_B). If this search succeeds., the Client B authenticated netID, CardID, Card Public Key and MAC will be sent to A.

A can verify if the received CardID is a match for the attached public key, based on the MAC check. Moreover, A can be sure that the information came from the KDC because the information he receives is encrypted using SLK, a shared secret in the workgroup. Furthermore, A can be sure that B's netID is valid because he can verify the signature on B's NetID with B's public key!

Now A can use B's authenticated NetID and public key to either establish a secure channel with the latter, or verify some additional piece of information, that is allegedly signed by B. The *eSmartLock* Client/Server configuration serves as a proprietary – hardware based Certification Authority, using the embedded secrets hosted in *eSmartLock* cards; over the LAN or over an extranet( eg. Serving the needs of mobile users, or users who need remote secure access, over an insecure internet connection). It can be actually extended to support LDAP services or X.509 certificates that can be hosted on web browsers to allow secure access to corporate web-based content, or support secure email communication.

- ***eSmartLock* Personalization**

*eSmartLock* encapsulates all this diverse behavior in a single JavaCard applet. All *eSmartLock* cards share the same JavaCard code. What makes them each one different, is the KeySet each one receives – or self generates – during its instantiation phase by the Vendor. The *eSmartLock.java* applet contains an appropriate code segment, which allows a post-production personalization of a Card with application specific keysets and identity flags (Client, Server, KDC, StandAlone).

- **Antidebugging techniques - What about these?**

*eSmartLock* is based on a standard JavaCard 2.1.1 platform which mandates that the executable java code inside an *eSmartLock* card is static – unless one updates the applet regularly, which is rather slow. Therefore we need to implement some basic security measures inside the PC code to make hacking attacks on the system difficult.

The basic measures used were the following: Encryption of object data at the source level and executable code encryption; both of which decrypt and execute normally only in the presence of the *eSmartLock* dongle. Timetraps that check for abnormal execution duration patterns, which indicate a tracing attack. Trapping and constant redirection of interrupts used by software debuggers (int 3h and 68h in PCs). Code that checks if a debugger is executing along the PC application. Use of dumb threads to delay and confuse a tracing attack. Logic bombs inside the exe code that ‘detonate’ if the.exe code has been patched ; introduction of memory leaks in case an attack has been detected.

A quite efficient protection measure has been the embedding of PC application java-equivalent code routines in the card (‘surrogate’ external application code on card). Testing this approach, I ended up nesting application specific java code as a co-existing applet along with *eSmartLock* on the same eGate card. In any case, an attacker would need to simulate the embedded application’s code having as a hint only a collection of input – output vectors of the card’s USB communication traffic, which is never replicated since it is also masked with a ticket key. Believe me, this not a trivial task!

As a conclusion, it must be noted that JavaCard technology is very promising indeed. The functionality and flexibility *eSmartLock* offers compares favorably - and in many areas surpasses – most of the dongle products currently marketed worldwide. It is actually the strength of JavaCard that made the difference.

Thank you for your attention! - Yiannis Hatzopoulos