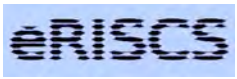
Coordinateur et Consortium :

SNT *Satellite Network Terminal*

Projet [1] **:** ***DGCIS- Pôle Compétitivité SCS***

Titre Delivrable **Spécifications Sécurité Communications SNT**

Partenaire responsable: **ERISCS** [2]**-Université de la Méditerranée**



Date de remise effective: **15/06/09**

Contact: **Traian MUNTEAN <muntean@univmed.fr>**

---

[1] *Ce travail a été partiellement financé par le CR-PACA dans le cadre du Projet SNT du Pôle de Compétitivité SCS*

[2] ***Groupe d'Etudes et de Recherche en Informatique des Systèmes Communicants Sécurisés***
ERISCS- Parc Scientifique de Luminy-ESIL , F-13288-Marseille ; (tél.+33(0)491828501; fax: 0491828511)

TABLE des MATIERES

1. PRÉSENTATION GÉNÉRALE

Le contenu de ce livrable représente un résumé des approches et des résultats des travaux menés par le Groupe de Recherche ERISCS (Groupe d'Etudes et de Recherche Informatique des Systèmes Communicants Sécurisés) de l'Université de la Méditerranée à Marseille dans le cadre du projet SNT.

L'objectif général est de proposer des solutions adaptées et innovantes pour la sécurité des communications satellitaires et des protocoles supportés par le terminal SNT.

L'approche est dirigée par les contraintes et les besoins des applications proposées dans le projet SNT par les partenaires industriels du projet proposé à la DGCIS, mais nous gardons un caractère générique plus général pour les systèmes de communications critiques. En particulier, les aspects et approche de sécurité proposés nous ont amené à développer des études plus générales pour les méthodes cryptographiques et algorithmes associés. De plus, pour couvrir les besoins de performances requis par les applications nous intégrons les solutions proposées dans le système embarqué dans le terminal (basé sur un noyau Linux libre distribution) pour offrir des interfaces support efficaces pour les applications et les services futurs.

La structure de ce livrable est comme suit : après ce préambule nous présentons en 2 l'approche générale et les choix initiaux proposés en conformité avec l'évolution des standards actuels ; en 3 nous présentons nos travaux sur le mode de chiffrement Galois/Counter que nous proposons d'intégrer à une CryptoBox générique qui est en cours de développement dans notre laboratoire (la partie utilisée dans le projet SNT sera présentée dans un livrable complémentaire) ; en 4 nous résumons nos travaux actuels et à venir sur le chiffrement par courbes elliptiques (ECC) et hyper-elliptiques et les perspectives d'utilisation; nous terminerons en 5 sur les travaux réalisés et les logiciels produits.

Remarques :

• ***Etant donné que le financement pour notre participation à ce projet n'a été disponible qu'en avril 2009, nos travaux couvrent aussi des efforts antérieurs sur ressources propres et continueront pour couvrir l'effort global proposé pour le financement du projet.***

• *Par conséquent notre collaboration avec les partenaires du projet devrait se poursuivre après la date de fin annoncée pour cette phase du projet SNT.*

• *A part cette présentation générale et de l'approche décrite en 2, le reste du document sera en anglais pour permettre une dissémination plus large des résultats.*

2. APPROCHE POUR LA SÉCURISATION D'UNE GAMME D'APPLICATIONS SATELLITAIRES

Il s'agit de sécuriser le transfert de certains flux de données transmises lors des communications supportées par le terminal SNT entre utilisateurs et applications ou entre services des diverses applications. Plus précisément, le flux de données transmises sont analysées et une partie de ces données est transmise en clair, une autre partie est chiffrée. En outre, ces données, que ce soit la partie transmise en clair ou la partie chiffrée, pourront être sécurisées du point de vue de leur intégrité.

Nous présentons ci-dessous dans les grandes lignes les mécanismes mis en place pour la protection des données qui vont transiter et assurer les fonctionnalités requises.
Le contexte architectural générique de développement expérimental et de prototypage actuel est le suivant : sur un « terminal type PC sous Linux », un utilisateur envoie un flux de données à travers une carte spécifique SNT vers un satellite. Un autre utilisateur muni d'un équipement analogue reçoit ce flux. Le travail consiste à sécuriser la transaction. Le chiffrement se fait entre l'utilisateur et la carte spécifique SNT du terminal. Bien entendu le déchiffrement se fait entre la carte du destinataire et le destinataire.
On considère ici pour simplification de la présentation qu'au départ on récupère un flux d'octets dont on ne se soucie pas de la structure et qu'on rend à l'arrivée le même flux d'octets.

La sécurisation va se faire avec les moyens les plus récents et pense-t-on, les plus performants de la cryptographie.
Les primitives et protocoles sont choisis et mis en place en tenant compte des standards du NIST et de l'ISO et des choix de l'étude européenne NESSIE sur les primitives cryptographiques de base.
En outre le texte récent de la NSA, suite B nous conforte dans ces choix:

1) AES (FIPS pub 197)
2) GCM (Galois/Counter mode) et GCMAC (800-38D)
2) SHA256 (FIPS pub 180-3)
3) ECDSA (FIPS pub 186-3)
4) EC-KEM (ISO- 18033-2) ou ECDH

Dans les grandes lignes, on chiffre les données et on les protège avec un code MAC en utilisant la primitive AES en mode Galois/Counter et GMAC. Les clés secrètes du mode précédent sont échangées (avec authentification des protagonistes) en utilisant un mécanisme KEM (Key Encapsulation Mechanism) utilisant une courbe elliptique suivant les recommandations de ISO-18033-2 et NIST 800-56 A. Les signatures, si nécessaire, sont

faites par de la cryptographie elliptique suivant le standard du NIST ECDSA.
Les nombres aléatoires d'initialisation sont générés avec le périphérique /dev/random de Linux, suivi d'un hachage SHA256. Les fonctions de dérivation de clés (KDF) sont construites à partir d'une valeur initiale ("seed") par application répétée de SHA256 comme décrit dans ISO- 18033-2.

2.1. LE CHIFFREMENT ET L'INTÉGRITÉ :

La primitive de chiffrement par bloc choisie est AES. Les tailles des clés secrètes sont 128 bits pour le mode sûr et 256 bits pour le mode « top level security ».

Le mode d'utilisation est le Galois/counter mode, qui assure d'une part avec le mécanisme « counter » une rapidité et une sécurité optimales, et d'autre part qui permet avec le mécanisme « Galois » de contrôler, si besoin est, l'intégrité des données, aussi bien celles qui sont chiffrées que celles qui transitent en clair.

Les algorithmes correspondants pourraient, si besoin est, être parallélisés pour des performances requises par certains systèmes ou réseaux embarqués.

L' « initial value » demandée par l'algorithme est tiré au sort à l'aide du générateur pseudo-aléatoire inclus dans la toolbox cryptographique. Ce générateur pseudo-aléatoire utilise un germe initial construit avec le périphérique /dev/random, puis construit ses valeurs successives grâce à la fonction de hachage SHA256 (voir les détails dans la documentation de la toolbox cryptographique).

2.2. L'ÉCHANGE DE CLÉ

La clé secrète utilisée pour le chiffrement des données est échangée par ECDH (elliptic curve Diffie Hellman). Ceci nécessite la mise en place d'une distribution de bi-clés : clé publique clé privée, adaptées au système elliptique correspondant.

Un rapport plus précis sur le travail fait (et à prolonger) sur la partie concernant la cryptographie elliptique se trouve ci-après dans §3. Cependant, nous tenons à rappeler ici qu'avec les tailles actuelles des systèmes à clé secrète (c'est-à-dire de 128 bits à 256 bits) la cryptographie RSA ou sur le groupe (Z/pZ)* n'est plus adaptée. D'un autre côté, si l'utilisation de RSA était relativement standard dans la mise en place, il n'en va pas de même de la cryptographie elliptique qui est actuellement en pleine évolution et qui permet de nombreuses variantes aussi bien dans le choix des courbes que dans le choix de l'implémentation des opérations sur ces courbes (cf. §3.).

2.3. SIGNATURE

Une fonctionnalité de signature numérique (qui n'entre pas dans les fonctionnalités requises par le projet) est cependant disponible sous forme d'une signature ECDSA. Le choix de la courbe et de l'implémentation de l'opération est celui de l'échange de clé.

2.4. GÉNÉRICITÉ ARCHITECTURALE

Le système mis en place **est évolutif** au moins dans les sens suivants :

1) l'implémentation logicielle peut être remplacée par une implémentation par matériel, éventuellement avec une architecture parallèle.

2) La cryptographie elliptique à clé publique, qui couvre ici l'échange de clés (et la signature si besoin est) peut évoluer en ce qui concerne les courbes choisies, les implémentations de l'opération (en particulier est à l'étude actuellement, la détermination de courbes données dans le modèle d'Edwards, ce qui permet une meilleure implémentation de l'addition sur la courbe). En outre une base de courbes disponibles pourrait être maintenue (le NIST ne propose qu'une seule(!) courbe de taille 256 bits et qu'une courbe de taille 512 bits). Il va de soi que cette situation, alors même que la cryptographie elliptique est devenue indispensable, n'est pas satisfaisante. Si des logiciels classiques sont capables de fournir de bonnes courbes pour la taille 256 bits, il n'en va pas de même pour la taille 512. Des modifications de logiciels existants, et notamment le calcul de tables plus étendues de polynômes modulaires ont dû être faites.

3) Les prototypes logiciels que nous livrons au projet SNT sont implémentés en C avec utilisation de la bibliothèque GMP. Nous réalisons en plus des interfaces appropriées pour l'usage de ces prototypes à partir d'application en Java à la demande des partenaires du projet SNT.

Ces logiciels sont couverts par des clauses de propriété intellectuelle et ne sont destinés qu'à l'usage de prototypage du projet. Des licences étendues d'utilisation seront proposées pour une utilisation plus large ou industrielle plus étendue des codes sources des logiciels.

3. SECURING SATELLITE PROTOCOLS

3.1. INTRODUCTION

Satellite has played an important role in telephony communication and TV broadcasting services since the birth of telecommunication satellites. It is less known that satellite also plays an important role in broadband and Internet services and will continue to play an important role in the future generation networks. This is due to the satellite characteristics that make a niche position for satellites in the global network infrastructure (GNI). Satellite networking is a special and important topic together with other networking technologies in recent years. Due to the nature of satellite links (long propagation delay, relative high bit error rate and limited bandwidth in comparison with terrestrial links, particularly optical links), some standard network protocols do not perform well and have to be adapted to support efficient connection over satellite. Satellite orbit directly affects the link characteristics and has a significant impact in satellite network design. It is the ultimate goal of satellite networking to support the many different applications and services available in terrestrial networks. These applications and services generate different types of traffic having different requirements in terms of network resources and quality of service (QoS), particularly the recent development of integration of telecommunication, broadcast and computer networks and integration of telephone, TV, computer and global positioning system (GPS) terminals.

Satellite networking has evolved significantly since the first telecommunications satellite, from telephone and broadcast to broadband and Internet networks. It has adapted during the advancement for ISDN, ATM, Internet, digital broadcast, etc. The evolution has also been reflected in research and development, including the recent studies of onboard processing, onboard switching and onboard IP routing. There are also new developments and new issues in satellite networking such as resource management, security and quality of service, new services and applications including VoIP, multicast, video conference, DVB-S, DVB-RCS and IPv6 over satellite. There are always many practical constraints, such as cost, complexity, technologies and efficiency of space and ground segments in design, implementation and operation. Often trade-offs have to be made to achieve an optimal solution.

3.1.1. BASIC NETWORK SECURITY MECHANISMS

Security in general is intended to protect the end-user identity (including their exact location), data traffic to and from the user, signaling traffic and also to protect the network operator against use of the network without appropriate authority and subscription. The basic mechanics in the Internet at network layer used to provide security include authentication using public key systems, privacy using public and secret key systems and

access control using firewalls and passwords. Internet security is a very important and also very difficult problem particularly in satellite networking, as the Internet covers the world across political and organizational boundaries. It also involves how and when communicating parties (such as users, computer, services and network) can trust each another, as well as understand the network hardware and protocols.

3.1.2. SECURITY APPROACHES

Security coding can proceed by two approaches:

- *Layer-to-layer approach*: in this case, a computer layer (usually, layer 3 – IP layer or layer 4 – TCP and UDP layer) receives an uncoded file from the above layers, encapsulates the file in a protocol data unit (PDU), and codes the whole frame before sending it to the other end. There, the corresponding layer of the peer entity will decode the PDU before sending the file to the higher layers. This requires, however, that those routers on the network are able to deal with completely coded frames.
- *End-to-end approach*: in this case, the files are coded directly at the application layer by the user, and a coded file is handed out to the lower layers for delivery. This means that only the data payload of the frames is coded (contrary to the previous case, where all the frames were coded).

In the second case, cryptography has only indirect consequences on network traffic, and this only if the coding algorithm has an effect on the size of the data to be transmitted. This is the case for hashing functions or algorithms like RSA.

In the first case, this kind of coding implies an overhead in the frames, thus decreasing the useful load of data carried. This kind of mechanism is implemented in IPv4 and IPv6, with different mechanisms.

In IPv4, cryptography is an option that is activated in the 'Options' field of the header (6th: 32-bit row), in IPv6, it is included as an 'extra header' (since the Options field is used in IPv6) of 64 bits.

Another possible consequence, besides the added headers and the variation of the size of data, is the apparition of messages for exchange of session keys, which never happens under normal (i.e. without cryptography) circumstances.

3.2. THE GALOIS COUNTER MODE OF OPERATION

In cryptography, the block ciphers operate on blocks of fixed length, usually of 128 bits long. Because encrypting the same block P with the same key K the result will always be the same this poses a threat to security of the encryption. An attacker can see for example if the plaintext contains repeating patterns. This is why cipher modes of operation are used.

The implementation in which each separate block is only ciphered with the key is called ECB (Electronic Codebook).

Some of the most used modes of operation nowadays are CBC (Cipher-block chaining), PCBC (Propagating cipher-block chaining), CFB (Cipher feedback), OFB (Output feedback) and CTR (counter)

3.2.1. THE CIPHER-BLOCK CHAINING

CBC mode of operation was invented by IBM in 1976. In the cipher-block chaining (CBC) mode, each block of plaintext is Xored with the previous ciphertext block before being encrypted. This way, each ciphertext block is dependent on all plaintext blocks processed up to that point. Also, to make each message unique, an initialization vector must be used in the first block.

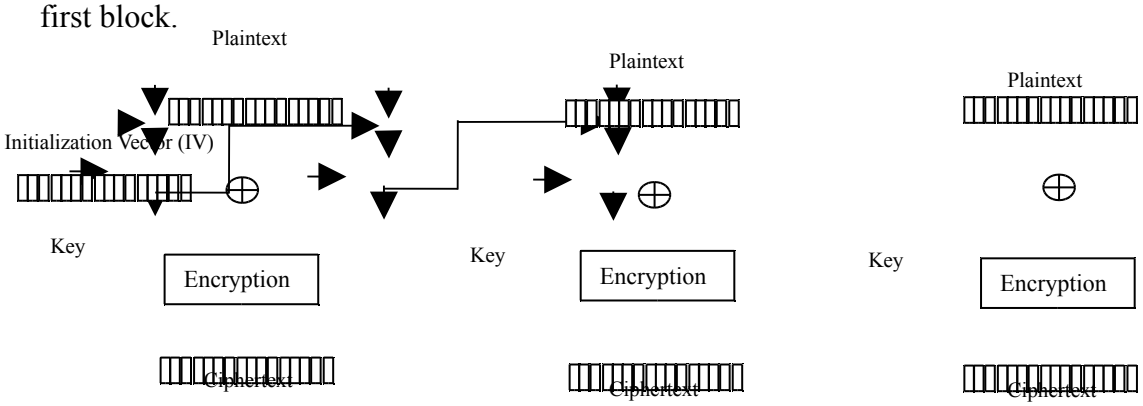


Fig.2.5 Cipher Block Chaining (CBC) mode encryption

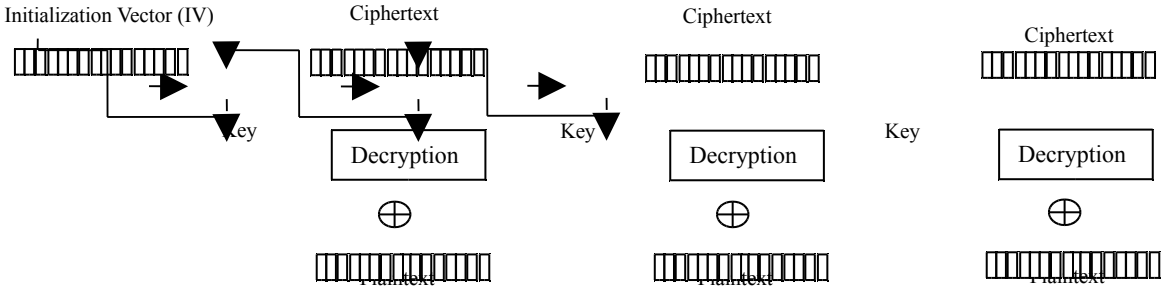


Fig.2.6 Cipher Block Chaining (CBC) mode decryption

If the first block has index 1, the mathematical formula for CBC encryption is

$$C_i = E_k(P_i \oplus C_{i-1}), C_0 = IV$$

while the mathematical formula for CBC decryption is

$$P_i = D_k(C_i) \oplus C_{i-1}, \quad C_0 = IV$$

CBC has been the most commonly used mode of operation. Its main drawbacks are that encryption is sequential (i.e., it cannot be parallelized), and that the message must be padded to a multiple of the cipher block size. One way to handle this last issue is through the method known as ciphertext stealing.

Note that a one-bit change in a plaintext affects all following ciphertext blocks. A plaintext can be recovered from just two adjacent blocks of ciphertext. As a consequence, decryption *can* be parallelized, and a one-bit change to the ciphertext causes complete corruption of the corresponding block of plaintext, and inverts the corresponding bit in the following block of plaintext.

3.2.2. THE COUNTER MODE

The counter mode turns a block cipher intro a stream cipher. The algorithm of encryption is applied to successive values of a "counter" and the result is then xored with the plaintext to produce the ciphertext.

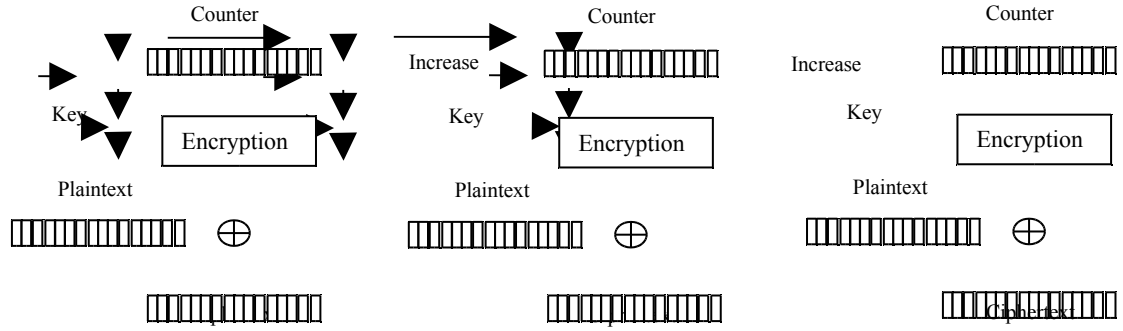


Fig.2.7 The counter mode encryption

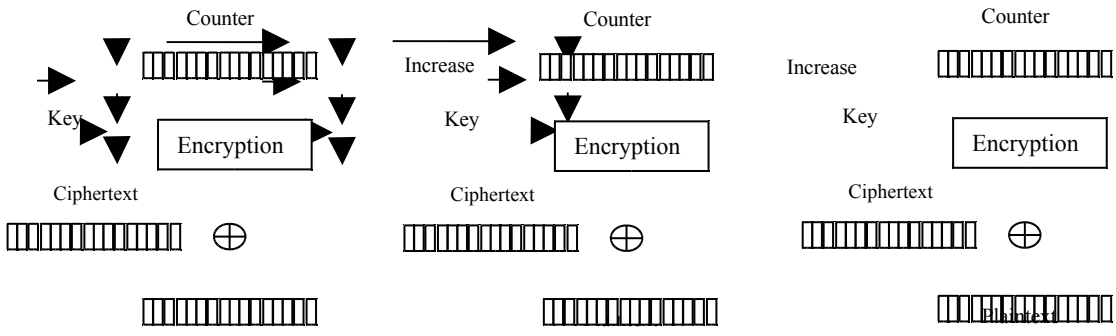


Fig.2.8 The counter mode decryption

The counter is initialized with an IV (initialisation vector) so that encrypting the same plaintext would produce different results. There are several implementations for the way the counter changes from one step to another but most of the times a simple increment is used. The main advantage of this mode is that it can be easy parallelized so that the encryption is performed very fast. Moreover, there is basically no difference between the encryption and the decryption operations, the algorithm is the same, only the input changes.

None of the operation modes mentioned above provide data integrity. This means that someone that doesn't know the key may modify the data so that the changes will not be detected. Securing data means both encrypting and providing integrity protection.

The Galois Counter Mode (GCM) is a block cipher mode similar to the Counter mode which provides authenticated encryption. This mode is very fast in hardware and through table-driven field operations it can achieve great speed in software also. This mode admits pipelined and parallelized implementations an can provide authenticated encryption at speed of 10gigabits per second. GCM is also free of IPRs. GCM can also be used just for providing stand-alone authentication, without encrypting any data.

3.2.3. DEFINITION OF THE GALOIS COUNTER MODE

Galois/Counter Mode (GCM) is a block cipher mode of operation that uses universal hashing over a binary Galois field to provide authenticated encryption. It can be implemented in hardware to achieve high speeds with low cost and low latency. Software implementations can achieve excellent performance by using table-driven field operations. It uses mechanisms that are supported by a well-understood theoretical foundation, and its security follows from a single reasonable assumption about the security of the block cipher.

This mode admits pipelined and parallelized implementations and has minimal computational latency, making it useful at high data rates.

GCM also has additional useful properties. It is capable of acting as a stand-alone MAC, authenticating messages when there is no data to encrypt, with no modifications. Importantly, it can be used as an incremental MAC: if an authentication tag is computed for a message, then part of the message is changed, an authentication tag can be computed for the new message with computational cost proportional to the number of bits that were changed. This feature is unique among all of the proposed modes.

Another useful property is that it accepts initialization vectors of arbitrary length, which makes it easier for applications to meet the requirement that all IVs be distinct. In many situations in which authenticated encryption is needed, there is a data element that could be used as a nonce, or as a part of a nonce, except that the length of the element(s) may exceed the block size of the cipher. In GCM, a nonce of any size can be used as the IV. This property is shared with EAX, but no other GCM proposed mode.

GCM uses block ciphers of 128 bits with AES.

3.2.3.1. AES

AES is one of the most popular algorithms used today in symmetric key cryptography. It was announced by the National Institute of Standards and Technology on November 26, 2001 after been selected most suitable from fifteen competing designs, five years earlier. The main advantages of AES are that it is fast both in software and hardware, it requires little memory and it is easy to implement.

3.2.3.2. Inputs and outputs

GCM uses block ciphers of 128 bits.

Inputs:

- Key K which is appropriate for the underlying block cipher.
- The initialization vector (IV), which can be of any size. For best performance 96-bit IV should be used.
- The plaintext P which can have between 0 and 2^{39}-256 bits.
- Additional authenticated data (AAD), which can have between 0 and 2^{64} bits.

Outputs

- The ciphertext C, with the same length as P
- The authentication tag T, with a length between 0 and 128 bits.

The decryption has the same inputs as the encryption, except of course the plaintext which is replaced by the ciphertext, plus the authentication tag T used to verify the integrity. The output is the plaintext.

The AAD is used to protect de information, but it is not encrypted. For example, when using GCM to secure a network protocol, the AAD could include ports, sequence numbers, etc.

The IV is used so that for distinct invocations of the encryption operation using the same key the results would not be the same.

The primary purpose of the IV is to be a nonce, that is, to be distinct for each invocation of the encryption operation for a fixed key. It is acceptable for the IV to be generated randomly, as long as the distinctness of the IV values is highly likely. The IV is authenticated, and it is not necessary to include it in the AAD field.

Both confidentiality and message authentication is provided on the plaintext. The strength of the authentication of P, IV and A is determined by the length t of the authentication tag. When the length of P is zero, GCM acts as a MAC on the input A. The mode of operation that uses GCM as a stand-alone message authentication code is denoted as GMAC.

Securing data means both encrypting and also providing integrity protection. None of the operation modes mentioned above provide data integrity. This means that someone that doesn't know the key may modify the data so that the changes will not be detected.
The Galois Counter Mode (GCM) is a block cipher mode, similar to the Counter mode, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf which provides authenticated encryption This mode is very fast in hardware and through table-driven field operations it can achieve quite significant speed in software too. This mode admits parallelized implementations and can provide authenticated encryption at speed of 10gigabits per second.

GCM can also be used just for providing stand-alone authentication, without encrypting any data. GCM is also free of intellectual property restrictions.

3.2.3.3. Notation

The notation follows that of the Recommendation for Block Cipher Modes of Operation [7]. The two main functions used in GCM are block cipher encryption and multiplication over the field $GF(2^{128})$. The block cipher encryption of the value X with the key K is denoted as $E(K,X)$. The multiplication of two elements $X, Y \in GF(2^{128})$ is denoted as X · Y, and the addition of X and Y is denoted as $X \oplus Y$. Addition in this field is equivalent to the bitwise exclusive-or operation, and the multiplication operation is defined in Section 2.5.3.5.

The function len() returns a 64-bit string containing the nonnegative integer describing the number of bits in its argument, with the least significant bit on the right. The expression 0^l denotes a string of l zero bits, and $A \| B$ denotes the concatenation of two bit strings A and B. The function $MSB_t(S)$ returns the bit string containing only the most significant (leftmost) t bits of S, and the symbol {} denotes the bit string with zero length.

3.3. IMPLEMENTATION

For implementing the GCM we have used the Java programming language and the library Gnu-Crypto. There are several ways the data can be passed to the program, but for testing purposes we used binary files from which we read the key, the IV and the plaintext. We also implemented a function for generating a random IV using SecureRandom.

We used byte arrays to represent the blocks of bits. The encryption of the plain text is quite easy to implement. It uses the xor() function, which is used for two byte vectors and the encrypt() function, used to cipher a 128-bit block with the key K. For the cipher we use AES-128, from the library Gnu-Crypto. The counter Y is incremented with another function.

$$
\begin{aligned}
Y_i &= increment(Y_{i-1}) \quad i = 1..n \\
C_i &= P_i \oplus E(K, Y_i) \quad i = 1..n-1 \\
C_n &= P_n \oplus MSB_t(E(K, Y_n))
\end{aligned}
$$

Regarding the structure of the program, we have two important classes: the Crypto class and the Utils class. The Utils class contains a few small functions like xor, increment and others that will be described later on. The Crypto class is the one that holds the functions for reading the key, the IV, the AAD and the plaintext and also the functions for the block cipher. While reading the AAD and the plaintext we compute the intermediate values of T so that little memory has to be used.
P is split into blocks of 128, but the length of P may not be a multiple of 128. In this case, the last block, P_n is xored with only the most significant t bytes of the ciphered counter.
Y_i denotes the value of the counter. Y_0 is initialized with a zero right padded version of the IV, if the IV has 96 bits. Otherwise Y_0=GHASH(H,{},IV). The function GHASH is described further on. It is also used to compute the authentication tag t.

$$
X_i = \begin{cases}
0 & ,i = 0 \\
(X_{i-1} \oplus A_i) \cdot H & ,i = 1..m-1 \\
(X_{m-1} \oplus A_m^* \,||\, 0^{128-v}) \cdot H & ,i = m \\
(X_{i-1} \oplus C_i) \cdot H & ,i = m+1..m+n+1 \\
(X_{m+n-1} \oplus C_m \,||\, 0^{128-u}) \cdot H & ,i = m+n \\
(X_{m+n} \oplus (len(A) \,||\, len(C))) \cdot H & i = m+n+1
\end{cases}
$$

$$GHASH(H, A, C) = X_{m+n+1} \text{ and } H = E(K, 0^{128})$$

GHASH(H,A,C)=X_{m+n+1}

Len() is a function implemented in the Utils class that returns the length of argument, contained in a vector of 64 bits. The function || concatenates the 2 arguments.

The multiplication in GF(2^{128})

A finite field is defined by its multiplication and addition operations which are of course commutative, associative and distributive. Both operations map two field elements onto another field element. For the multiplication we consider the 128-bit vectors as polynomials. The result of multiplying two polynomials X and Y is a 256-bit polynomial that has to be divided to the field polynomial .The field polynomial is fixed, and for the GCM it is $f = 1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{128}$.

The addition of two elements X and Y consists of adding the polynomials. It is identical to the bitwise exclusive-or of X and Y. Subtraction is identical to addition in GF(2). To show how the multiplication in GF(2) works we multiply an element X to the element P , where

$$P = \begin{cases} 1 & , i = 1 \\ 0 & , otherwise \end{cases}$$

P is the polynomial α and the multiplication can be written as:

$$\alpha \cdot (x_0 + x_1\alpha + x_2\alpha^2 + \ldots + x_{127}\alpha^{127}) = x_0\alpha + x_1\alpha^2 + x_2\alpha^3 + \ldots + x_{127}\alpha^{128}$$

If $x_{127} = 0$ then the result is a polynomial of degree 127 or less. Otherwise it has to be divided by the field polynomial f to find the remainder. We have:

$\alpha^{128} + a$ where $a = a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{127}\alpha^{127}$ (a is a polynomial of degree 127)

We need to find q and r so that $\alpha^{128} + a = q \cdot f + r$

$r = \alpha^{128} + a - f$, if q=1. This means that $r = a + 1 + \alpha + \alpha^2 + \alpha^7$

Therefore, the pseudo code for the multiplication is:

```
if x127=0 then
        Y←rightshift(X)
else
        Y←rightshift(X) ⊕ R
endif
```

The multiplication operation is defined as an operation on bit vectors in order to simplify the specification. Each element is a vector of 128 bits. The i-th bit of an element X is denoted as x_i. The leftmost bit is x_0, and the rightmost bit is X_{127}. The multiplication operation uses the

special element R = 11100001. The function rightshift() moves the bits of its argument one bit to the right. More formally, whenever W = rightshift(V), then $W_i = V_i{-}1$ for i = 1 .. 127 and $W_0 = 0$.

We implemented two methods for the multiplication. The first one is straightforward and uses the classical method to multiply two polynomials. The second is table-driven using precomputed tables to decrease the number of operations needed for this function. With the use of the tables the implementation is much faster and can be compared to AES.

For the first method, the algorithm is:

```
Z←0, V←X
for i=0 to 127 do
    if Yi=1 then
        Z←Z ⊕ V
    endif
   V←V ·P
end for
return Z
```

This algorithm is quite time consuming so its use is not recommended. The second method of multiplying two field elements X and Y takes into account that Y is constant during one invocation of the algorithm and is equal to H.

We have: $$X = \bigoplus_{i=0}^{15} x_i \cdot P^{8i}$$

where x_i represents an element of the field that has its first byte equal to the i-th byte of X and all the other bytes are 0.

$$H \cdot X = \bigoplus_{i=0}^{15} x_i \cdot H \cdot P^{8i}$$

Here, there are two possible approaches, one that uses more memory, but which is faster, and the other one that uses less memory but is less efficient.

For the first one, we compute 16 tables, M0 to M15, where $Mi[x] = x \cdot H \cdot P^{8i}$. Each table holds 256 values, so we need 65536 bytes. The result can be expressed as:

$$H \cdot X = M0[x_0] + M1[x_1] + \ldots + M15[x_{15}]$$

For the second approach we use only two tables: the table M0 and another table to compute $x_i P^{128}$.

$$H \cdot X = \bigoplus_{i=0}^{15} x_i \cdot H \cdot P^{8i} = \bigoplus_{i=0}^{15} (x_i \cdot H) \cdot P^{8i}$$

which leads to the following algorithm:

```
Z←0
for i=0 to 15 do
   Z←Z ·P^8
   Z←Z ⊕ M0[x_i]
end for
return Z
```

For the multiplication $Z \cdot P^8$ we use again the decomposition

$$Z = \bigoplus_{i=0}^{15} z_i \cdot P^{8i}$$

So that

$$Z \cdot P^8 = \bigoplus_{i=0}^{15} z_i \cdot P^{8(i+1)}$$

The multiplication with $P^{8(i+1)}$ is easy to compute for i<15, being equivalent with a rightshift with 8(i+1) bits. For i=15 we must compute $z_i \cdot P^{128}$, for which we will use a second table R.

The algorithm to compute the table M0 uses a few optimizations. An entry that has an index that is a power of two contains a product of H times a power of P. The other elements are computed by summing together these elements. M[128]=H because 128=10000000.

```
M[128]←H, i←64
while i>0 do
        M[i] ←M[2i] ·P
        i ←[i/2]
end while
i ←2
while i<=128 do
        for j=1 to i-1 do
                M[i+j] =M[i] ⊕ M[j]
        end for
        i←2i
end while
M[0] ←0^128
```

The table R can be defined as a constant in the source of the program, as it is not key dependent. We used the results of another project to compute R, after which we took the

output and set a constant byte array. The table R only requires 512 bytes, because the result of the operation $x \cdot P^{128}$ only has the first two bytes non-zero.

Now, to compute the multiplication using the tables M and R, the algorithm is:

```
Z←H
for i=15 to 0 do
        A←z_i
        for j=15 to 1 do
        z_j=z_j-1
        end for
        Z=Z ⊕ R[A]
        while x_i
        Z←Z ⊕ M[x_i]
end for
```

3.4. REFERENCES

[1]. Zhili Sun, Satellite Networking Principles and Protocols, Wiley, 2005

[2] Eclipse.org website (http://www.eclipse.org/)

[3] Jtux website (http://basepath.com/aup/jtux/)

[4] Federal Information Processing Standards Publication 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES), NIST, 2001

[5] D. McGrew, J. Viega, The Galois/Counter Mode of Operation (GCM) (http://www.cryptobarn.com/papers/gcm-spec.pdf)

[6]T. Hall, The Galois/Counter Mode (GCM) and GMAC Validation System (GCMVS), NIST, Feb.11.2009 (http://csrc.nist.gov/groups/STM/cavp/documents/mac/gcmvs.pdf)

[7] M. Dworkin, Recommendation for Block Cipher Modes of Operation: Methods and Techniques, NIST Special Publication 800-38A.

[8] V. Shoup, On Fast and Provably Secure Message Authentication Based on Universal Hashing,Advances in Cryptology - Proceedings of CRYPTO '96, 1996.

[9] C. Parr, Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems. ECC '99.

[10]Emilia Kasper, Peter Schwabe, Faster and Timing-Attack Resistant AES-GCM (http://homes.esat.kuleuven.be/~ekasper/papers/fast_aes.pdf)

4. ELLIPTIC CURVE CRYPTOGRAPHY (ECC)

4.1. INTRODUCTION

The use of elliptic curves in cryptography was originally proposed independently by Koblitz [9] and Miller [12] in 1985, and has since become one of the central topics of research in cryptography. This is largely on account of the way the subject fuses deep and interesting mathematical problems from number theory with the provision of cryptographic primitives that are not merely applicable, but demonstrate a clear superiority to all previous methods.

This document aims to give a high-level account of elliptic curve cryptography, a guide to the selection of safe and efficient domain parameters, its relevance to satellite communications, and the current state and future directions of the Cryptobox implementation of elliptic curve cryptography. It should be noted that, while every effort has been made to eschew mathematical details as much as possible, the nature of the subject makes this difficult. A very basic knowledge of groups, fields and polynomials should suffice for the most part. In this document, all groups and fields will be finite, and all groups will be abelian (i.e. $P \oplus Q = Q \oplus P$ for all P and Q in a group whose operation is $\oplus$).

Readers looking for more thorough background material are invited to consult Washington's *Elliptic Curves: Number Theory and Cryptography* [21]. The main reference for this document is the *Handbook of Elliptic and Hyperelliptic Curve Cryptography* [5].

4.2. ELLIPTIC CURVES

We present here an introduction to elliptic curves and the properties associated to them that are of interest to cryptographers. A comprehensive introduction to the theory of elliptic curves can be found in Silverman's *The Arithmetic of Elliptic Curves* [17].

Let Fq be a finite field, where $q = p^d$ is a power of a prime number p . For our purposes, an *elliptic curve over* Fq is the set of all pairs (x, y) , where x and y are elements of Fq , which satisfy the polynomial equation

$$y^2 + a_1 x y + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6 . \qquad (2.1)$$

together with a special point called the *point at infinity*. Here a_1 , a_2 , a_3 , a_4 , and a_6 are elements of the field Fq for which the discriminant of the polynomial equation is nonzero. We will call a pair (x, y) that satisfies the polynomial equation a *point on the*

elliptic curve, and as such will often be denoted by P or Q. When $p>3$, Equation 2.1 can be put into the much simpler equivalent form

$$y^2=x^3+a_4x+a_6. \tag{2.2}$$

The finite field Fq is called the *ground field* of the elliptic curve. We denote the set of points (x,y) that satisfy the elliptic curve's polynomial equation by $\mathcal{E}(Fq)$. When the ground field is obvious from the context, or when it is not our primary concern in the exposition, we will denote the elliptic curve simply by $\mathcal{E}$.

The most important fact about the set $\mathcal{E}(Fq)$ is that it forms an group with the point at infinity playing the role of the identity element. Thus we can perform the familiar arithmetic operations of addition, subtraction and scalar multiplication on the points in $\mathcal{E}(Fq)$. Recall that scalar multiplication by n in a group means adding an element of the group to itself n times. Scalar multiplication by n of an element P in a group G will be denoted by $[n]P$, so

$$[n]P=\underbrace{P\oplus P\oplus\cdots\oplus P}_{n\,\text{times}}$$

where $\oplus$ is the group operation in G. Moreover, this group is finite (since there are only finitely many possibilities for the x and y coordinates of the points), and so in particular it is a product of finite cyclic groups (this is a basic theorem of group theory). Recall that a cyclic group C is one for which there is an element G such that every other element can be written in the form $[n]G$ for some integer n. Such an element G is called a *generator* of the group. In general, the *order of an element* P in a group G is the smallest number n for which $[n]P=0$, and the *order of a finite group* is just the number of elements in it. If C is a cyclic group of order ℓ, then any generator of C will have order ℓ, and in particular if ℓ is prime then any nonzero element of C will be a generator. Conversely, if a finite group has prime order, it is cyclic.

Now $\mathcal{E}(Fq)$ is a product of cyclic groups, and each cyclic group is a subgroup of $\mathcal{E}(Fq)$. In particular, the order of each cyclic group divides the order of $\mathcal{E}(Fq)$. Let C be a cyclic subgroup of $\mathcal{E}(Fq)$, and suppose that C has order ℓ and that $\mathcal{E}(Fq)$ has order n. Then ℓ divides n, and the quotient n/ℓ is called the *index* of the subgroup C in $\mathcal{E}(Fq)$.

4.3. ELLIPTIC CURVE CRYPTOGRAPHY

Cryptographic ciphers are usually divided into two categories, public key ciphers and private key ciphers. *Private key ciphers* use a single key to both encrypt and decrypt blocks of data. Private key ciphers tend to be a great deal faster than public key ciphers, making them suitable for encrypting large amounts of information. However, for the recipient of the encrypted data to be able to decrypt it, the key must be shared between the two parties, which raises the problem of how to exchange the key safely. Moreover, when many people need to send encrypted data to one another, the number of keys needed grows very quickly.

By contrast, *public key ciphers* use two distinct keys, a public key for encryption and a private key for decryption (not to be confused with the private key used in private key ciphers). They have the advantage that the public key can be distributed without compromising the private key. This means that there is no need to exchange any secret information before being able to send encrypted information, since anyone with access to a person's public key can send encrypted information to them. As public key ciphers tend to be much slower than private key ciphers, they are not practical for use with large amounts of data. For this reason, most systems employ a hybrid of the two, using a public key cipher to exchange a secret key which is then used in a private key cipher.

The security of a public key cipher is based on the intractability of a certain mathematical problem (or problems). For example, the most famous public key encryption scheme, RSA, is based on the difficulty of factoring the the product of two large prime numbers: the user selects two large prime numbers as the private key and publishes the product as the public key. While finding large primes and multiplying them together is relatively easy, finding the two primes given the product is very difficult.

4.3.1. THE DISCRETE LOGARITHM PROBLEM

Elliptic curve cryptography uses the intractability of the discrete logarithm problem on elliptic curves as the basis of public key ciphers. The *discrete logarithm problem (DLP)* is as follows: given two points P and Q in a cyclic subgroup of an elliptic curve $\mathcal{E}$, find a number n such that $[n]P=Q$. The discrete logarithm problem makes sense in any cyclic group, but if the ground field of the elliptic curve is large and the cyclic subgroup of the curve is a large prime, then the discrete logarithm problem in the subgroup is thought to be more difficult than the corresponding problem on the ground field's multiplicative group. This topic will be dealt with in more detail in §4.4.4.

4.3.2. PROTOCOLS

The two main protocols that employ elliptic curve-based public key ciphers are the Elliptic Curve Diffie-Hellman key agreement scheme (ECDH), and the Elliptic Curve Digital Signature Algorithm (ECDSA). To get a taste of what these protocols are like, we will describe the basic version of the ECDH. Suppose Alice and Bob wish to establish a shared secret (for use with a private key cipher, for example). They must first agree on the domain parameters, namely

- the prime p and the positive integer d that define the ground field, Fpd ;
- the coefficients a_1 , a_2 , a_3 , a_4 and a_6 of the elliptic curve ξ (see Equation 2.1);
- a point G on ξ which generates a (large) cyclic subgroup of $\xi(Fpd)$;
- the order ℓ of the cyclic subgroup generated by G ; and

(Advice on secure choices of these parameters will be described in §4.5.2, advice on efficient choices in §4.5.3.) In order to establish a shared secret using ECDH, Alice and Bob proceed as follows:

- They must each generate a *key pair*, consisting of a *private key* k which is a random integer between 1 and $\ell-1$ and a *public key* $P=[k]G$. (This public key can be reused safely across multiple invocations of the cipher). Let Alice's key pair be (P_A,k_A) and Bob's key pair be (P_B,k_B) .
- Alice and Bob must be in possession of each others public keys, which are safe to transmit over an insecure channel.
- Alice computes the point $(x_A,y_A)=[k_A]P_B$ and Bob computes the point $(x_B,y_B)=[k_B]P_A$. These points are in fact equal, since

$$(x_A,y_A)=[k_A]P_B=[k_A][k_B]G=[k_B][k_A]G=[k_B]P_A=(x_B,y_B),$$

  and so the shared secret is the number $x_A=x_B$.

The protocol is secure since only the public keys are disclosed, and in order to break the protocol, an attacker must find k_A from $[k_A]G$ or k_B from $[k_B]G$, that is, they must solve the discrete logarithm problem on the curve.

Two things should be noted here. The first is that, although public knowledge of the public keys is allowed, indeed it is necessary for the scheme to work, some care must be taken when exchanging the public keys. For example, it is necessary to ensure that the public key is really that of the person to whom one is sending the (potentially sensitive) information. There are at least two mechanisms by which to ensure authenticity of the public keys: the use of a certificate authority, or the use of a web-of-trust.

The second thing to note is that ECDH allows two parties to agree on a shared secret, but this secret is fixed for any two parties. As it is usually undesirable to reuse a private key in most symmetric ciphers, usually one employs a "key derivation function" to generate a secure key that is suitable for use in a symmetric cipher from the shared secret.

4.4. ADVANTAGES OF ELLIPTIC CURVE CRYPTOGRAPHY

The most important choice to make when considering the use of a cipher is the key size. This is, obviously enough, a measure in bits of the size of the keys used by the cipher. It is important to note, however, that it is distinct from the cryptographic security of the cipher, which is a (logarithmic) measure of the fastest known computational attack on the algorithm: if the best known computational attack on the cipher requires $O(2^n)$ operations, then the cipher is said to have n -bit security. The security of a cipher cannot exceed the key size (since any cipher is susceptible to a brute force attack), however it can be smaller. While most modern private key ciphers manage to attain a security measure equal to the key length, there are no known public key ciphers that have this property. However, of all known public key ciphers, those based on the DLP on elliptic curves come the closest to attaining this property.

The fastest known algorithms to solve the discrete logarithm problem on a general elliptic curve are in fact the generic algorithms that work with any finite group (such as Shank's baby-step giant-step algorithm and Pollard's rho algorithm) which require $O(\sqrt{\ell})$ steps (where ℓ is the size of the subgroup of $\xi(Fq)$ being used). Thus in general, to attain n -bit security from an elliptic curve, the size of the ground field should be roughly twice the chosen security parameter, i.e. $2n$. For example, when using ECDH to transmit a 128-bit AES private key, one should select a ground field for the elliptic curve of approximately 256-bits. Compare this to cryptography over a finite field, such as the finite field Diffie-Hellman protocol, which would require 3072-bit public keys and 256-bit private keys, or using a cipher based on integer factorisation (such as RSA) which would require 3072-bit public and private keys. If we would like to use 256-bit AES keys, then we need a ground field of only 512 bits for ECDH, whereas NIST recommends 15360-bit (!) public and private RSA keys [3, §5.6.1].

Thus the keys used in elliptic curve cryptography can be chosen to be much shorter for a comparable level of security compared to existing methods based on integer factorisation and multiplicative groups of finite fields.

4.5. SELECTING APPROPRIATE ELLIPTIC CURVES

4.5.1. DOMAIN PARAMETERS

In general, there are two methods for selection domain parameters: either one selects a special curve that is known in advance, or one selects a curve at random and verifies that it has suitable properties. There is continuous, and often heated, debate as to which of these approaches is more secure, so the following exposition will cover both cases. We refer the reader to Koblitz *et al.* [8] for a thorough exposition of the history and issues behind this debate.

It is important to note that unless the domain parameters come from a trusted party (examples might include the National Institute of Standards and Technology [1], the Standards in Efficient Cryptography Group [2] or using a special curve that is well-documented), they must be validated before use. In particular, when using randomly generated curves from an unknown source, it is good practice to select curves whose coefficients are suitably sized *hashes* of random numbers in order to demonstrate that the curve could not have been selected to have any undesirable special properties.

The domain parameters were listed in §4.3.2, and we will use the same notation throughout this section.

4.5.2. SECURE DOMAIN PARAMETERS

The most basic security property to consider is the order of the subgroup of the elliptic curve we are considering using, since it is the order of this subgroup that determines the difficulty of the DLP. For a curve to have a subgroup with sufficiently many points (and hence a certain number of bits of security) it is necessary that the curve be defined over a sufficiently large field. This is because the number of points on the curve is bounded by the Hasse-Weil bound. That is, if $\mathfrak{E}$ is an elliptic curve over Fq ($q=p^d$), then the number of points $\#\mathfrak{E}(Fq)$ satisfies

$$|\#\mathfrak{E}(Fq)-(q+1)|\leq 2\sqrt{q}.$$

This means that the number of points on the curve will be roughly the same as the number of points in the ground field (since there are q elements in Fq).

The following conditions must be met if a given curve is to be considered secure. We assume as before that $\mathfrak{E}$ is an elliptic curve defined over the finite field Fq, where $q=p^d$ is a prime power, ℓ is the order of a subgroup C of $\mathfrak{E}(Fq)$.

Then to achieve n -bit security using C ,

- ℓ must be at least 2^{2n} , i.e. it must be at least a $2n$ -bit number;
- ℓ must be prime, otherwise the Chinese Remainder Theorem can be used to reduce the DLP in C to the DLP in its prime order subgroups [13];
- ℓ must not equal p , to avoid vulnerabilities associated with so-called anomalous curves [14, 15, 16, 18];
- if k is the smallest number such that $\ell \mid (q^k-1)$, then k should be at least 1000 to avoid vulnerability to the MOV attack [11]; and
- d must be either 1 (in which case q is prime) or a large prime. In the latter case the number 2 should have a large order modulo d (i.e. if $2^t \equiv 1 (md)$, then t should be large). This condition avoids Weil descent attacks [6, 7].

4.5.3. EFFICIENT DOMAIN PARAMETERS

We now wish to consider choices of domain parameters that can speed up computations associated with elliptic curve cryptography. We will provide a summary of [5, §23.3].

There are two main areas that can be targeted when addressing the question of efficiency: the choice of ground field, and the choice of the curve equation.

Choice of ground field

The first choice to make regarding the ground field F_{p^d} is whether d should be 1 and p large, or p small and d a prime (always keeping in mind the recommendations of §4.5.2). In the former case, a speed advantage can be gained for the modular reductions in F_p if one chooses p to be of low Hamming weight, that is, select p of the form $2^{wn}+c$ where w is the word size of the computer and c is small.

In the latter case, where p is small and d is a prime, there are certain choices that can speed up arithmetic for certain applications or on certain platforms. For example, selecting p such that elements of the subfield F_p fit within one computer word means operations in this field are particularly efficient. A choice of $p=2$ may be better than $p=3$ since then elements can be represented by a single bit, whereas when $p=3$ the elements require 2 bits (though there are some applications which require $p=3$; see [5, §24]).

Choice of curve equation

Certain choices of the coefficients of the elliptic curve can permit faster arithmetic on the curve. We will present two examples here.

The first example is that of the “anomalous binary curves” or “Koblitz curves”. One takes the ground field F_{2^d} , and then selects either

$$y^2+xy=x^3+x^2+1$$

or

$$y^2+xy=x^3+1,$$

both of which are defined over $F2$. Such a curve admits significant speedups to the basic arithmetic functions, especially scalar multiplication [10, 19].

The second example is that of Edwards curves. An Edwards curve over Fpm (with $p\neq 2$) has the special form

$$x^2+y^2=1+dx^2y^2,$$

where d is in Fpm and d is not a square in Fpm (i.e. there is no number x in Fpm such that $d=x^2$). The relationship between Edwards curves and elliptic curves is somewhat technical, so we will be content to remark that one can often use an Edwards curve (or a so-called "twisted Edwards curve") as a model for an elliptic curve, and that they permit very fast arithmetic operations on elliptic curves [4]. In particular, unlike most other models of elliptic curves, they admit a single defining equation for the addition operation on the curve, which not only speeds up computation, but also makes implementation easier.

It should be noted that improving the efficiency of elliptic curve cryptography is a very active area of research, and many improvements can be expected in the coming years. Recent work on parallelising the algorithms seems particularly promising as it can take better advantage of the recent proliferation of multi-core processors.

4.6. ELLIPTIC CURVE CRYPTOGRAPHY IN CRYPTOBOX

4.6.1. AT PRESENT: NIST RECOMMENDED CURVES

When deciding on a which curve to use in elliptic curve cryptography, one has the choice between using an existing published recommendation from an established authority such as NIST [3], or generating a new curve. At present, Cryptobox uses the NIST recommended 256-bit and 512-bit elliptic curves for its implementation of the ECDH and ECDSA protocols.

A problem that arises when one selects a curve published by NIST is that it is impossible to benefit from recent research. For example, the NIST recommendations do not include any mention of Edwards curves, simply because they were not well known at the time the recommendations were published. Thus in order to enjoy the considerable benefits of Edwards curves, we must generate these curves from scratch. We will now outline the work that has been done to this end.

4.6.2. THE FUTURE: EDWARDS CURVES

As explained in §4.5.2, the most important security parameter is the number of points in the cyclic subgroup that is being used. The way one determines the number of points is to use a point counting algorithm. Such algorithms come in two varieties, the p -adic algorithms and the ℓ -adic algorithms. The p -adic algorithms are generally faster than the ℓ -adic algorithms, but the ℓ -adic algorithms have an additional property that makes them more suitable to our purposes: they provide a "short circuit" mechanism by which the calculation can be aborted early if the number of points is found to have certain divisors. This is particularly important in light of how slow these algorithms tend to be in general. In the case of Edwards curves, we know *a priori* that the number of points on the curve will have a divisor of 4. So when we count the number of points on an Edwards curve, we would like to abort the computation if the curve has a divisor *other than* 4. This will ensure that there is a large cyclic subgroup of prime order.

Part of the author's ongoing work is to generate secure Edwards curves for use with Cryptobox. This will take the form of a patch to the open source **PARI/GP** computer algebra system [20], which will allow users to have fine-grained control over which divisors will cause the point counting to short circuit and which divisors are permitted. This will allow the easy generation of Edwards curves with large cyclic subgroups, and thus allow us to enjoy the benefits of very fast arithmetic operations without the risk of compromising security.

4.7. REFERENCES

[1] Digital Signature Standard (DSS). Technical Report FIPS 186-2, National Institute of Standards and Technology, 2000. http://csrc.nist.gov/publications/fips/ fips186-2/fips186-2-change1.pdf.

[2] Recommended Elliptic Curve Domain Parameters. Technical Report SEC 2, Standards for Efficient Cryptography Group (SECG), 2000. http://www.secg.org/download/aid-386/sec2_final.pdf.

[3] Recommendation for Key Management—Part 1: General (Revised). Technical Report SP 800-57, National Institute of Standards and Technology, 2006. http://csrc.nist.gov/publications/ nistpubs/800-57/SP800-57-Part1.pdf.

[4] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, Advances in cryptology—ASIACRYPT 2007, 13th international conference on the theory and application of cryptology and information security, Kuching, Malaysia, December 2–6, 2007, volume 4833 of Lecture Notes in Computer Science, pages 29–50. Springer, December 2007.

[5] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, editors. Handbook of elliptic and hyperelliptic curve cryptography. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2006.

[6] Steven D. Galbraith and Nigel P. Smart. A cryptographic application of Weil descent. In Cryptography and coding (Cirencester, 1999), volume 1746 LNCS., pages 191–200. Springer, Berlin, 1999.

[7] P. Gaudry, F. Hess, and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. J. Cryptology, 15(1):19–46, 2002.

[8] Ann Hibner Koblitz, Neal Koblitz, and Alfred Menezes. Elliptic curve cryptography: The serpentine course of a paradigm shift. Journal of Number Theory, DOI: 10.1016/j.jnt.2009.01.006, 2009.

[9] Neal Koblitz. Elliptic curve cryptosystems. Math. Comp., 48(177):203–209, 1987.

[10] Neal Koblitz. Cm-curves with good cryptographic properties. In CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology, pages 279–287, London, UK, 1992. Springer-Verlag.

[11] Alfred J. Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. IEEE Trans. Inform. Theory, 39(5):1639–1646, 1993.

[12] Victor S. Miller. Use of elliptic curves in cryptography. In Advances in cryptology—CRYPTO '85 (Santa Barbara, Calif., 1985), volume 218 of LNCS., pages 417–426. Springer, Berlin, 1986.

[13] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. IEEE Trans. Information Theory, IT-24(1):106–110, 1978

[14] Takakazu Satoh and Kiyomichi Araki. Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves. Comment. Math. Univ. St. Paul., 47(1):81–92, 1998.

[15] Takakazu Satoh and Kiyomichi Araki. Errata to the paper: "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves" [Comment. Math. Univ. St. Paul. 47 (1998), no. 1, 81–92; MR1624563 (99e:11083)]. Comment. Math. Univ. St. Paul., 48(2):211–213, 1999.

[16] I. A. Semaev. Evaluation of discrete logarithms in a group of p -torsion points of an elliptic curve in characteristic p . Math. Comp., 67(221):353–356, 1998.

[17] Joseph H. Silverman. The arithmetic of elliptic curves, volume 106 of Graduate Texts in Mathematics. Springer-Verlag, New York, 1986.

[18] N. P. Smart. The discrete logarithm problem on elliptic curves of trace one. J. Cryptology, 12(3):193–196, 1999.

[19] J. A. Solinas. Efficient arithmetic on Koblitz curves. Des. Codes Cryptography, 19(2-3):195–249, 2000

[20] The PARI Group, Bordeaux. PARI/GP, version 2.4.3, 2009. available from http://pari.math.u-bordeaux.fr/.

[21] Lawrence C. Washington. Elliptic curves: Number theory and cryptography. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, second edition, 2008.

4.8. HYPERELLIPTIC CURVE CRYPTOGRAPHY (HCC)

We give an overview of the current status of hyperelliptic curve cryptography and point out some directions in which progress is desired.

4.8.1. DEFINITIONS

A hyperelliptic curve (in imaginary quadratic form) H over a field K is defined by

$$y^2+h(x)y=f(x), h, f\in K[x]$$

with $d(h)\leq g$ and[3] $d(f)=2g+1$. We require the curve to be *non-singular* which means there are no pairs $(x,y)\in\overline{K}\times\overline{K}$ which satisfy both the curve equation and the two partial derivative equations

$$2y+h(x)=0, h'(x)y=f'(x).$$

The integer $g\geq 1$ is called the *genus* of the curve. Elliptic curves are examples of hyperelliptic curves of genus 1 . When the characteristic of K is different from 2 , one can "complete the square" and work rather with a simpler curve equation which has $h(x)=0$.

The set of K -rational points is

$$H(K):=\{(x,y)\in K\times K \mid y^2+h(x)y=f(x)\}\cup\{\infty\}$$

where ∞ is an additional point called the *point at infinity*.

In contrast to genus one, the set of K -rational points $H(K)$ does not form a group in higher genus. For $g\geq 1$ there exists a smallest abelian group $J_H(K)$ in which $H(K)$ embeds, called the *Jacobian* of H . With this definition, we deduce that an elliptic curve is equal to its Jacobian.

The number of F_q -rational points on J_H lies in the Hasse-Weil interval

$$(q^{1/2}-1)^{2g}\leq \#J_H(F_q)\leq (q^{1/2}+1)^{2g}.$$

In particular $\#J_H(F_q)\approx q^g$ and hence hyperelliptic curves over F_q of higher genus produce larger groups than those of lower genus.

---

[3] If $d(f)=2g+2$ the defining equation describes a hyperelliptic curve of genus g in real quadratic form. For simplicity we shall not be considering these equations.

The best algorithm for solving DLP in a generic abelian group G is Pollard's rho method [1, §19.5] which requires $O(\sqrt{\#G})$ group operations, that is, *exponential* in $l(\#G)$. This means that the DLP in $J_H(F_q)$ can always be solved in $O(q^{g/2})$ group operations. For genus greater than 2, index calculus algorithms [1, Ch. 21] can solve the DLP significantly faster than Pollard's rho method. Thus for cryptographic purposes we are left with elliptic curves (well understood) and genus 2 hyperelliptic curves.

4.8.2. PARAMETER CHOICES

The majority of attacks on hyperelliptic curves are straightforward generalisations of attacks on elliptic curves. Let H be a genus 2 hyperelliptic curve over a field F_q of characteristic p, where $q=p^d$. Let r be the largest prime factor of $\#J_H(F_q)$. This determines a unique subgroup G of order r for which we can construct a public key cryptosystem based on the hardness of discrete logarithm problem: given points P,Q in G, find a number n such that $[n]P=Q$. We list the conditions placed on the parameters to avoid known attacks on hyperelliptic curves:

- The prime r must be large and $\#J_H(F_q)/r \le 16$ is small. This is because the attack of Pohlig-Hellman [1, §19.3] reduces DLP on $J_H(F_q)$ to subgroups of prime order.
- We require $r \neq p$, for otherwise one can map the DLP across to the additive group $(F_p,+)$ which is trivial to solve.
- The smallest integer k satisfying $q^k \equiv 1 (m r)$ is greater than 1000. This nullifies any potential advantage of mapping the DLP across to the multiplicative group $F_{q^k}^{\times}$ via the Tate-Lichtenbaum pairing [1, §22.2].
- Either q is prime, or $q=2^d$ with d prime. This is to avoid an effective Weil descent [1, §22.3]: in certain situations, $J_H(F_q^d) \hookrightarrow J_X(F_q)$ where X has larger genus and index calculus methods are quicker.

4.8.3. ALGORITHMS

The best algorithms for performing arithmetic on genus 2 hyperelliptic curves make use of the associated Kummer surface, which is a generalisation of Montgomery arithmetic for elliptic curves to the genus 2 case [3]. While the number of bit operations is larger than that of elliptic curves, this is counterbalanced by only requiring a field with half the logarithmic size for comparable security. The efficiency very much depends on the implementation; if we can construct hyperelliptic Jacobians of cryptographic size, there is no reason to favour genus 1 over genus 2 hyperelliptic curves.

The main hurdle for hyperelliptic curves with $g>1$ is that point counting algorithms are slower. While there do exist polynomial time algorithms , they are impractical for fields of large prime characteristic. At present a minimum of 96 bits is suggested for hyperelliptic curve cryptosystems [8], which means that the group order should be larger than 2^{192} . The genus 2 hyperelliptic curve point counting record is due to Gaudry and Schost [4] who computed a 254 -bit Jacobian over F_p where $p=2^{127}-1$, but the largest prime factor has only 168 bits so it is insecure and inefficient to use this curve for cryptography. Currently the best known method for constructing hyperelliptic curves of cryptographic size is the CM method [1, Ch. 18].

The basic idea of the CM method is to work backwards: starting with the group order, construct the curve. The number of points on a genus 2 hyperelliptic curve over F_q and its Jacobian is encoded by the characteristic polynomial of Frobenius

$$f_q(X)=X^4-tX^3+sX^2-tq+q^2,$$

namely we have that $\#C(F_q)=q+1-t$ and $\#J_H(F_q)=f_q(1)$. We use precomputed invariants called *Igusa class polynomials* which describe genus 2 Jacobians over the complex numbers whose endomorphism ring is an order in a quartic CM field K . If we can find an element of K which has minimal polynomial f_q for some prime q , then the roots of the Igusa class polynomials (called *CM points*) mod q can be used to construct genus 2 hyperelliptic curves having the same Frobenius characteristic polynomial. A choice of quartic CM field K determines an infinite set of polynomials f_q and in this way, we have some control over the group order.

4.8.4. RESEARCH DIRECTIONS

Finding methods for computing Igusa class polynomials is an active area of research. An improved understanding of endomorphism rings, explicit isogenies and isogeny graphs in genus 2 is highly desirable to extend the database of Igusa class polynomials [7]. These polynomials have very large coefficients (the same occurs with the analogous Hilbert class polynomials in genus 1). An alternative set of CM quartic invariants could produce simpler polynomials (analogous to Atkin polynomials [1, §17.2.3.c] in genus 1). This will be investigated

New and improved algorithms for computing Humbert surfaces will be devised. Points on Humbert surfaces describe genus 2 Jacobians whose endomorphism ring contains a real quadratic order $Z[x]/(x^2+bx+c)$ of discriminant $D=b^2-4ac>0$. All CM points lie

on a Humbert surface and this fact can be used to speed up endomorphism calculations and algorithms to compute Igusa class polynomials [6, §7.9]. Also, from the definition, a genus 2 Jacobian corresponding to a point on a Humbert surface has a $\sqrt{D}$ -endomorphism for some $D>0$. Work of Gaudry [2] shows that if this endomorphism can be made efficiently computable, it can then be used to speed up point counting on the Jacobian over a finite field. Also, the GLV method [1, §15.2.1] can then be used to speed up scalar multiplication.

Equations for Humbert surfaces are currently known only for discriminants less than 70 (see [5]). Our goal is to extend the list of discriminants past 200 .

4.8.5. REFERENCES

[1] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, editors. Handbook of elliptic and hyperelliptic curve cryptography. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2006.
[2] Pierrick Gaudry. Genus 2 formulae based on theta functions and rm kummer surfaces. Talk given at ECC, 2007.
[3] Pierrick Gaudry and David Lubicz. The arithmetic of characteristic 2 kummer surfaces and of elliptic kummer lines. Finite Fields and Their Applications, 15(2):246 – 260, 2009.
[4] Pierrick Gaudry and Éric Schost. Hyperelliptic curve point counting record, June 2008.
[5] David Gruenewald. Humbert surface data. http://echidna.maths.usyd.edu.au/ ∼ davidg/thesis.html.
[6] David Gruenewald. Explicit algorithms for Humbert surfaces. PhD thesis, The University of Sydney, 2009.
[7] David Kohel. Echidna database - complex multiplication class invariants in genus 2. http://echidna.maths.usyd.edu.au/ ∼ kohel/dbs/complex_multiplication2.html.
[8] Tanja Lange. private communication.

5. LOGICIELS LIVRÉS

5.1. CRYPTOBOX: FONCTIONS DE LA LIBRAIRIE KTB

(C) ACRYPTA & ERISCS Group / 10 Juin, 2009 / (Librairie KTB CryptoBox)
Contact: Traian.Muntean@univmed.fr

5.1.1. REMARQUES PRÉLIMINAIRES

L'utilisation de la bibliothèque nécessite de noter les remarques suivantes :

(1) **Allocation mémoire.**Les procédures fournies par la bibliothèque ne font pas l'allocation de la mémoire (sauf pour leurs propres variables locales). Il est de la responsabilité de l'utilisateur de faire les allocations mémoire nécessaires.

Par exemple, quand une procédure P fournit un résultat dans une zone mémoire appelée destination : *P(unsigned char *destination,unsigned char *origine)* cette zone mémoire doit être allouée (malloc par exemple) avant l'appel de la procédure.

(2) **État de la mémoire.** En sortie, les procédures remettent à *null* les variables locales statiques allouées sur la pile, remettent à *null* les variables locales dynamiques allouées sur le tas (zones mémoires allouées par la procédure (*mallo*c etc.) pour son fonctionnement interne) et libèrent ces zones alloués. Ceci est fait pour effacer de la mémoire tout renseignement résiduel (une clé secrète par exemple). Attention : comme l'optimiseur du compilateur peut décider que cette remise à zéro faite juste avant la libération de la mémoire ne sert à rien, il peut la supprimer. En conséquence une procédure spéciale a été prévue, pour une version ultérieure, pour que la "zéroisation" ait toujours lieu.

(3) **Terminaison des chaînes.** Sauf dans le cas très particulier où les chaînes ont vocation à être affichées, les procédures ne rajoutent jamais un *null* à la fin d'une zone mémoire. Donc l'utilisateur n'a pas à prévoir un octet de plus pour les zones à réserver, sauf dans quelques cas qui seront signalés dans la documentation.

(4) **Les entiers.** Les entiers sont considérés comme stockés en *little endian*.

5.1.2. LES STRUCTURES SPÉCIFIQUES

1. La structure pseudo-aléatoire. — Le noyau de ktb contient un générateur pseudo-aléatoire destiné essentiellement à produire des clés ou des valeurs initiales. Ce générateur fonctionne de la façon suivante : lors de son lancement un germe est tiré au sort en utilisant un générateur physique spécifique système (/dev/random sous Linux). Ensuite un système de génération pseudo-aléatoire prend le relais du générateur physique, celui ci ne pouvant en général pas suivre la demande en raison du temps de remplissage de son réservoir d'aléa.

Cette structure appelée **ktb_seed_structure** contient

- un germe initial **unsigned char secret_seed[32]** gardé secret, de taille 256 bits,
- un compteur de 128 bits **unsigned char seed_counter[16]**,
- un état unsigned char seed_state[32] de 256 bits.

Voici la déclaration de la structure :

```
typedef struct {
  unsigned char secret_seed[32]; //256 bits (32 octets)
  unsigned char seed_counter[16]; //128 bits (16 octets)
  unsigned char seed_state[32]; //256 bits (32 octets)
}
ktb_seed_structure;
```

Dans les grandes lignes voici le fonctionnement. On initialise la structure (utilisation de la fonction *ktb_init_seed256*) en tirant une fois pour toutes un germe de 32 octets soit 256 bits (utilisation de /dev/random) qui est mis dans la zone *secret_seed*. Le compteur *seed_counte*r (zone de 16 octets soit 128 bits) est mis à zéro. Le haché des 48 octets obtenus par la concaténation du germe et du compteur donne les 32 octets de *seed_state*. Quand on veut un aléa de 256 bits on utilise la fonction *ktb_draw_seed256* qui incrémente le compteur, concatène le germe *secret_seed*, l'état *seed_state* et le compteur *seed_counte*r (soit 80 octets) hache les 64 premiers octets de la suite de 80 octets pour en faire l'aléa tiré au sort, et hache la suite complète des 80 octets pour en faire le nouvel état *seed_state*.

Nous verrons plus en détail au §3 les fonctions qui manipulent cette structure :

ktb_seed_256
ktb_init_seed256
ktb_inc_seed_counter256
ktb_draw_seed256
ktb_rd
ktb_seed_close

Cependant on peut dire qu'en utilisation normale, seules les fonctions

```
ktb_init_seed256 /* initialisation de la structure
ktb_draw_seed256 /* tirage au sort d'un alea de 256
bits ktb_rd /* tirage au sort d'un entier <n
ktb_seed_close /* destruction de la structure
```

seront appelées directement, les autres étant à usage interne.

2. Structure adaptée au chiffrement aes en mode Galois-Counter.

```
typedef struct {
   unsigned char  ktb_counter[16];
   unsigned char  key[16];
   unsigned char  msubkey[16];
   u32 crk[44];
}
ktb_gcm128_state;

typedef struct {
   unsigned char  ktb_counter[16];
   unsigned char  key[32];
   unsigned char  msubkey[16];
   u32 crk[60];
}
ktb_gcm256_state;
```

Le champ **ktb_counter** contient l'initial value sur 96 bits et le compteur proprement dit sur 32 bits. Le champ **key** contient la clé secrète. Le champ **msubkey** contient le chiffré du bloc de 128 bits tous à 0. Le champ **crk** contient les clés de ronde de chiffrement. Il est rempli quand on initialise le chiffrement grâce à la fonction **ktb_aes_KeySetupEnc** (appelée par la fonction **ktb_gcm128_cinit**, donc on n'a pas à s'en occuper en dehors de l'appel à **ktb_gcm128_cinit**. Remarque: la fonction de déchiffrement n'est pas utilisée par ce mode.

5.1.3. LES FONCTIONS

1. ktb_seed256. — Les fonctions de ce composant sont les suivantes :

```
void ktb_seed256(unsigned char *seed256);
void ktb_init_seed256(ktb_seed_structure *rstruct);
void ktb_inc_seed_counter256(ktb_seed_structure *rstruct);
void ktb_draw_seed256(ktb_seed_structure *rstruct, unsigned char *seed256);
void ktb_rd(ktb_seed_structure *ss, mpz_t n, mpz_t *result);
void ktb_seed_close(ktb_seed_structure *ss)
```

En principe les seules fonctions que l'utilisateur doit appeler sont **ktb_init_seed256**, **ktb_draw_seed256**, **ktb_rd**et **ktb_seed_close**. L'utilisateur déclare une structure ktb_seed_structure puis l'initialise avec ktb_init_seed256 et l'utilise quand il en a besoin avec ktb_draw_seed256. Il récupère ainsi 32 octets dans une zone mémoire v qu'il aura déclarée et allouée. La fonction **ktb_rd** permet de tirer au sort de manière équiprobable un nombre entier x inférieur à un nombre donné n ($0 \leq x \leq n-1$).

La fonction **ktb_seed_close** permet de finir proprement sans laisser de trace.

```
ktb_seed_structure ss;
ktb_init_seed256(&ss);
char* v;
v=malloc(32*sizeof(char));
ktb_draw_seed256(&ss,v);
```

2. ktb_aes. — Les fonctions de ce composant sont les suivantes :

```
int ktb_aes_KeySetupEnc(u32 rk[/*4*(Nr + 1)*/],
        const u8 cipherKey[], int keyBits);
int ktb_aes_KeySetupDec(u32 rk[/*4*(Nr + 1)*/],
        const u8 cipherKey[], int keyBits);
void ktb_aes_Encrypt(const u32 rk[/*4*(Nr + 1)*/],
        int Nr, const u8 pt[16], u8 ct[16]);
void ktb_aes_Decrypt(const u32 rk[/*4*(Nr + 1)*/],
        int Nr, const u8 ct[16], u8 pt[16]);
```

Ces fonctions marchent avec des clés de 128 bits, 192 bits et 256 bits. Les nombres de tours correspondants (*Nr*) sont 10 tours (128 bits), 12 tours (192 bits), 14 tours (256 bits). En principe, sauf si l'utilisateur veut utiliser un autre mode que le mode GaloisCounter, il n'aura pas à appeler lui-même ces fonctions. Cependant, elles sont au cœur du chiffrement, si bien qu'on en donne tout de même le fonctionnement.

– La fonction *ktb_aes_KeySetupEn*c est une fonction préparatoire qui prend en entrée une clé *cipherKe*y de taille *keyBits*, renvoie une valeur entière qui redonne le nombre de tours et remplit un tableau *r*k avec les *N*r +1 clés de tour pour chiffrer. Il faut donc prévoir 16 octets pour les clés de 128 bits ainsi que la zone *r*k de 44 octets. Pour les clés de 192 bits il faut 24 octets et une zone *r*k de 52 octets. Pour les clés de 256 bits il faudra 32 octets et une zone *r*k de 60 octets.
– La fonction *ktb_aes_KeySetupDe*c fait la même chose que la fonction précédente mais pour le déchiffrement.
– La fonction *ktb_aes_Encrypt* prend en entrée la zone *N*r remplie par la première fonction, un nombre de tours *Nr*, ainsi qu'un bloc de texte clair (*ct*) et renvoie le bloc chiffré *pt*.
– La fonction *ktb_aes_Decrypt* fait la même chose pour le déchiffrement.
Attention il faudra zéroiser les zone utilisées par la clé K par les tableaux *rk*, les chiffrés *ct*.

3. ktb_general. — Les fonctions de ce composant sont les suivantes :

```
u64 os2u64(unsigned char *orig);
u32 os2u32(unsigned char *orig);
u16 os2u16(unsigned char *orig);
```

*void u64tostr(unsigned char *dest, u64 orig);*
*void u32tostr(unsigned char *dest, u32 orig);*
*void u16tostr(unsigned char *dest, u16 orig);*
*u32 ktb_concat(unsigned char *dest, unsigned char *append,*
u32 ldest, u32 lap);
*u32 ktb_shr(unsigned char *dest,unsigned char *orig,*
u32 size, u32 n);
*u32 ktb_shl(unsigned char *dest,unsigned char *orig,*
u32 size, u32 n);
*void ktb_xor(unsigned char *X,unsigned char *Y,*
*unsigned char *result,u32 n);*

Les fonctions **os2u64, os2u32, os2u16, u64tostr, u32tostr, u16tostr** sont respectivement des fonctions qui font des traductions de suites d'octets en entiers de type *u*64 (64 bits), *u*32, *u*16, et réciproquement. C'est juste une, recopie de la mémoire dans exactement le même ordre des octets. On effectue donc un transtypage avec recopie.

Les fonctions **ktb_concat, ktb_shr, ktb_shl, ktb_xor** réalisent la concaténation, la rotation à droite, la rotation à gauche, le ou exclusif bit à bit. Dans l'appel de chaque fonction on précise les zones mémoires concernées et les tailles de ces zones.

En outre, la longueur de la nouvelle suite d'octets est renvoyée en tant que valeur de retour de la fonction.

4. ktb_hash256. — Les fonctions de ce composant sont les suivantes :

*void ktb_sha256_memory(unsigned char *buf, int len, unsigned char *hash);*
*int ktb_sha256_file(unsigned char *filename, unsigned char *hash);*

Ces fonctions concernent le hachage en un bloc de 256 bits.

La fonction **ktb_sha256_memory** hache un buffer en mémoire **buf** dont on donne la taille **len** en octets pour en faire une empreinte **hash** de 256 bits. La fonction **ktb_sha256_file** hache un fichier pour en faire une empreinte **hash** de 256 bits.

5. ktb_hash512. — Les fonctions de ce composant sont les suivantes :

*void ktb_sha512_memory(unsigned char *buf, u64 len, unsigned char *hash);*
*int ktb_sha512_file(unsigned char *filename, unsigned char *hash);*

Ces fonctions concernent le hachage en un bloc de 512 bits. La fonction **ktb_sha512_memory** hache un buffer en mémoire **buf** dont on donne la taille **len** en octets pour en faire une empreinte **hash** de 512 bits. La fonction **ktb_sha512_file** hache un fichier pour en faire une empreinte **hash** de 512 bits.

6. ktb_kdf256. — La fonction de ce composant est:

*void ktb_kdf256(unsigned char *seed, u32 len, unsigned char *mask);*

Il s'agit là d'une fonction de dérivation de clé qui prend un germe de 256 bits, et en fait une chaîne **mask** de longueur **len** octets.

7. ktb_gcm_gene. — Les fonctions de ce composant sont les suivantes :

*void ktb_multelem(unsigned char *X, unsigned char *result);*
*void ktb_multFF(unsigned char *X,unsigned char *Y, unsigned char *result);*
*void ktb_inc(unsigned char *X); void ktb_ghash(unsigned char *H, unsigned char *X, unsigned char *Y,u32 m);*

8. ktb_gcm128. — Les fonctions de ce composant sont les suivantes :

*void ktb_gcm128_cinit(ktb_gcm128_state *cs, ktb_seed_structure *ss, unsigned char *K);*
*void ktb_gcm128_cprocess(ktb_gcm128_state *cs, unsigned char *pt, unsigned char *aad, u32 nbocts_pt, u32 nbocts_aad, unsigned char *ct, unsigned char *init_v, unsigned char *tag);*
*void ktb_gcm128_dinit(ktb_gcm128_state *cs, unsigned char *K, unsigned char *init_v);*
*u32 ktb_gcm128_dprocess(ktb_gcm128_state *cs, unsigned char *ct, unsigned char *aad, u32 nbocts_ct, u32 nbocts_aad, unsigned char *pt, unsigned char *tag);*
*void ktb_gcm128_cfile(ktb_gcm128_state *cs, unsigned char *ptfilename, unsigned char *aad, unsigned char *ctfilename);*
*u32 ktb_gcm128_dinit_file(ktb_gcm128_state *cs, unsigned char *K, unsigned char *ctfilename);*
*u32 ktb_gcm128_dfile(ktb_gcm128_state *cs, unsigned char *ctfilename, unsigned char *ptfilename, unsigned char *aad);*
*void ktb_gcm128_direct_cprocess(ktb_gcm128_state *cs, unsigned char *pt, u32 nbocts_pt, unsigned char *ct, unsigned char *init_v);*
*u32 ktb_gcm128_direct_dprocess(ktb_gcm128_state *cs, unsigned char *ct, u32 nbocts_ct, unsigned char *pt);*

9. ktb_gcm256. — Les fonctions de ce composant sont les suivantes :

*void ktb_gcm256_cinit(ktb_gcm256_state *cs, ktb_seed_structure *ss, unsigned char *K);*
*void ktb_gcm256_cprocess(ktb_gcm256_state *cs, unsigned char *pt, unsigned char *aad, u32 nbocts_pt, u32 nbocts_aad, unsigned char *ct, unsigned char *init_v, unsigned char *tag);*
*void ktb_gcm256_dinit(ktb_gcm256_state *cs, unsigned char *K, unsigned char *init_v);*

*u32 ktb_gcm256_dprocess(ktb_gcm256_state *cs, unsigned char *ct, unsigned char *aad,*
*u32 nbocts_ct, u32 nbocts_aad, unsigned char *pt, unsigned char *tag);*
*void ktb_gcm256_cfile(ktb_gcm256_state *cs, unsigned char *ptfilename,*
*unsigned char *aad, unsigned char *ctfilename);*
*u32 ktb_gcm256_dinit_file(ktb_gcm256_state *cs, unsigned char *K,*
*unsigned char *ctfilename);*
*u32 ktb_gcm256_dfile(ktb_gcm256_state *cs, unsigned char *ctfilename,*
*unsigned char *ptfilename, unsigned char *aad);*
*void ktb_gcm256_direct_cprocess(ktb_gcm256_state *cs, unsigned char *pt, u32 nbocts_pt,*
*unsigned char *ct,unsigned char *init_v);*
*u32 ktb_gcm256_direct_dprocess(ktb_gcm256_state *cs, unsigned char *ct, u32 nbocts_ct,*
*unsigned char *pt);*

Dans le mode GaloisCounter, on transmet des données à laisser en clair et des données à chiffrer, le tout étant protégé par un tag d'intégrité. Le système est souple et peut ne contenir que des données à chiffrer ou que des données à laisser en clair et peut aussi être utilisé en mode "direct", sans contrôle d'intégrité.

La fonction **ktb_gcmXXX_cinit** initialise le processus de chiffrement et d'intégrité en GaloisCounter mode. En particulier la structure **ktb_gcmXXX_state** est initialisée. Lafonction **ktb_gcmXXX_cprocess** effectue le chiffrement et le calcul du tag d'intégrité. Lesfonctions **ktb_gcmXXX_dinit** et **ktb_gcmXXX_dprocess** sont les fonctions correspondantes relatives au déchiffrement et au contrôle d'intégrité. Les fonctions **ktb_gcmXXX_cinit** (la même que précédemment), **ktb_gcmXXX_cfile** sont relatives au chiffrement et au calcul d'un tag d'un fichier, les fonctions **ktb_gcmXXX_dinit_file**, **ktb_gcmXXX_dfile** sont relatives au déchiffrement et au contrôle d'intégrité d'un fichier. Enfin, les fonctions **ktb_gcmXXX_direct_cprocess** et **ktb_gcmXXX_direct_dprocess** sont utilisée (en collaboration avec les fonctions d'initialisation **ktb_gcmXXX_cinit**et **ktb_gcmXXX_dinit**) pour chiffrer sans contrôle d'intégrité.

5.2. INTERFACE JNA POUR LA BIBLIOTHÈQUE LIBKTB.SO (CHIFFREMENT CRYPTOBOX)

(C) ERISCS Group / 10 Juin, 2009 / (Java/JNA pour CryptoBox)
Contact: Traian.Muntean@univmed.fr
Auteur: Gabriel Risterucci

5.2.1. FICHIERS FOURNIS

Dans le répertoire src se trouvent les sources Java pour les classes suivantes:

- clibs.libktb.Libktb contenant l'interface vers libktb.so
- clibs.libktb.Utils contenant des fonctions de conversion de chaines C
- clibs.test.Test_* contenant des exemples d'utilisation

En plus des sources Java, les fichiers suivants sont fournis:

- jna.jar contenant la bibliothèque JNA
- run.sh permettant de rapidement lancer les exemples
- achiffrer.txt fichier de test utilisé dans les exemples.

5.2.2. UTILISATION

La classe clibs.libktb.Libktb se suffit à elle même pour appeler les fonctions de libktb.so.

Si ce fichier n'est pas dans le chemin de recherche des librairies, il faut préciser son emplacement avant de l'utiliser, par exemple avec:

```
System.setProperty("jna.library.path", "/opt/lib");
```

Les fonctions de la librairie sont accessible via l'objet clibs.libktb.Libktb.INSTANCE, comme illustré dans les exemples.

Le fichier run.sh permet de compiler et lancer les exemples depuis la ligne de commande.

Fin de document