CryptoManager++

An Object Oriented Class Library for Cryptographic Mechanisms

Thilo Baldin, Gerrit Bleumer
Department of Computing, University of Hildesheim

Abstract

This paper describes the general concepts of an object oriented class library for cryptographic mechanisms: CryptoManager++. Its application programming interface (API) provides security services of various symmetric and asymmetric cryptographic mechanisms. Nevertheless, these services can be used in an uniform and convenient way achieved by means of object oriented techniques such as inheritance and overloading. Especially, the provided services include cooperations of several cryptographic mechanisms (like e.g. hybrid encryption), allowing to treat such a cooperation as one (integrated) mechanism at the API. Further main design principles were easy integration of new mechanisms, individually adjustable security, support of bit stream operands, portability and efficiency. The implementation of the CryptoManager++ is done almost completely in C++.

Keywords: Security API, cryptographic mechanisms, hybrid mechanisms, object oriented, class hierarchy

1. Introduction

The CryptoManager++ is a C++ class library which offers services for a great variety of cryptographic mechanisms such as RSA, IDEA, DES, MD5, SHS and so on. In particular, services are provided for

- generating, storing and loading cryptographic keys
- encrypting plain texts / decrypting cipher texts
- generating / verifying digital signatures
- hashing messages
- generating pseudo random bitstrings
- setting security parameters individually
- partitioning large data operands

Hence, the CryptoManager++ provides all basic building blocks for the implementation of cryptographic protocols to achieve message confidentiality, integrity, authentication and non-repudiation or advanced cryptographic protocols such as secure exchange and distribution of cryptographic keys, digital coin toss, zero-knowledge proofs, multi party computations and so on [Bras_88, Schn_93].

Our most important goal in designing the CryptoManager++ was to provide an API as *uniform* and *intuitive* as possible. For that reason, we tried to structure the great variety of cryptographic mechanisms in a natural way which helps the application programmer to use cryptographic services without being distracted by unnecessary details.

We characterize cryptographic mechanisms by two orthogonal properties: *cryptographic purpose* and *key structure*. The former distinguishes, for example, between encipherment and authentication mechanisms (including digital signature schemes and message authentication codes). Furthermore, efficient implementation of cryptographic mechanisms requires pseudo random bitstring generators (PRBGs) for key generation and different kinds of hash functions. The latter distinguishes between cryptographic mechanisms with symmetric keys (classical cryptography, [Kahn_67]) and those with asymmetric keys (public key cryptography, [DiHe_76]). Our idea was to enable the application programmer to look at cryptographic mechanisms in this general way when using cryptographic services. Notice that the implementation of other cryptographic APIs like [FIPS_94, XOpe_94] can also take full advantage of the concept we are going to describe.

2. Object Oriented Design

2.1 Modelling Cryptographic Mechanisms as a Class Hierarchy

We capture cryptographic purposes and key structures by introducing *purpose classes* (encipherment, authentication, hash and PRBG mechanisms) and *key classes* (symmetric and asymmetric mechanisms) which provide *all* services of the CryptoManager++ API as abstract services. A purpose class offers certain (abstract) *purpose services*, e.g. the purpose class *encipherment mechanisms* offers, among others, the services *encrypt* and *decrypt*. At this level, the service does not determine its implementation. For example, there exist many possible implementations of the service *encrypt* which differ with regard to security, efficiency and so on. RSA and DES encryption are two possible implementations among many others. Analogously, a key class offers certain (abstract) *key services*. For example, the key class *asymmetric mechanisms* offers a service *generate private and public key*.

In order to characterize cryptographic mechanisms completely we combine purpose and key services in so called *crypto classes*. Crypto classes distinguish between the symmetric and asymmetric versions of encipherment, authentication, hash and PRBG mechanisms[1]. In general, each crypto class is inherited from a purpose class and a key class (multiple inheritance) which results in an abstract cryptographic mechanism offering the abstract key *and* purpose services of its base classes. A specific cryptographic mechanism then constitutes one possible implementation of exactly one crypto class. For example, RSA is a possible implementation of crypto class *asymmetric encipherment mechanisms*. This also applies to keyless cryptographic mechanisms like the hash functions RIPE-MD, MD5 and SHS. In this case, no key class is necessary and hence their crypto class is identical with their purpose class. Completing the concept, each specific cryptographic mechanism is implemented by a *mechanism class* which inherits all (abstract) services from the corresponding crypto class and overloads those services appropriately.

2.2 Integrating Hybrid Cryptographic Mechanisms

There are cryptographic mechanisms which as *clients* use the services of other cryptographic mechanisms (*servers*). Such a cooperation of several different cryptographic mechanisms will be called a *hybrid cryptographic mechanism* (or simply *hybrid mechanism*) whereas a cryptographic mechanism using no services of others will be called an *elementary* one. Examples of hybrid mechanisms are:

(1) cooperation of a symmetric and an asymmetric encipherment mechanism in order to apply hybrid encryption [Garf_95]

(2) cooperation of a signature mechanism (asymmetric authentication mechanism) and a hash mechanism in order to truncate a message before generating or verifying a signature

(3) cooperation of an asymmetric encipherment mechanism and a hash mechanism in order to generate a redundancy predicate before encrypting a plain text (a redundancy predicate can be used to prevent chosen cipher text attacks, especially in case of RSA)

[1] There exist both asymmetric hash functions and PRBGs. Normally, these mechanisms look like symmetric ones because the secret key is often ignored. However, for some purposes it might be useful to have the secret key service available.

Like elementary mechanisms, hybrid mechanisms can also be characterized by the properties *cryptographic purpose* and *key structure* by distinguishing their dominating cryptographic purpose and their dominating key structure. For example, in (2), the dominating purpose is *authentication* and the dominating key structure is *asymmetric*. Therefore, hybrid mechanisms perfectly fit into our class hierarchy. The nice result is that at the API, a hybrid mechanism does not differ from any other elementary one of the same crypto class. The cooperation of different cryptographic mechanisms is hidden from the outside. Thus, the application programmer has to worry about neither the management of the different keys involved nor the combination of the different combined services.

Notice that a hybrid mechanism only determines the respective purpose or crypto classes of its servers but not the specific mechanisms. For example, in (2), any mechanism of crypto class *asymmetric authentication mechanism* and any mechanism of purpose class *hash mechanisms* is a possible server. Hence, a hybrid mechanism constitutes a *set* of possible implementations of one crypto class. For that reason, its mechanism class is realized as a template class inherited from a corresponding crypto class.

3. The Class Hierarchy

This chapter gives a more detailed overview of the class hierarchy resulting from the object oriented design outlined in chapter 2.

3.1 Purpose Classes

The CryptoManager++ offers four purpose classes inherited from a common base class *PURPOSE*:

(1) class ***ENC*** (encipherment mechanisms)

(2) class ***AUTH*** (authentication mechanisms)

(3) class ***HASH*** (hash mechanisms)

(4) class ***PRBG*** (pseudo random bitstring generators)

For any purpose class, certain purpose services are available. Mainly, these purpose services are:

(1) ***Encrypt / Decrypt*** (class *ENC*):
encrypting a specified plain text / decrypting a specified cipher text

(2) ***Sign / Verify*** (class *AUTH*):
generating / verifying a digital signature for a specified message

(3) ***Hash*** (class *HASH*):
hashing a specified message

(4) ***TickRandom*** (class *PRBG*):
generating a pseudo random bitstring of specified length

In addition, services for the support of memory management and the partition of large data operands are provided. Figure 1 illustrates the purpose classes of the CryptoManager++.

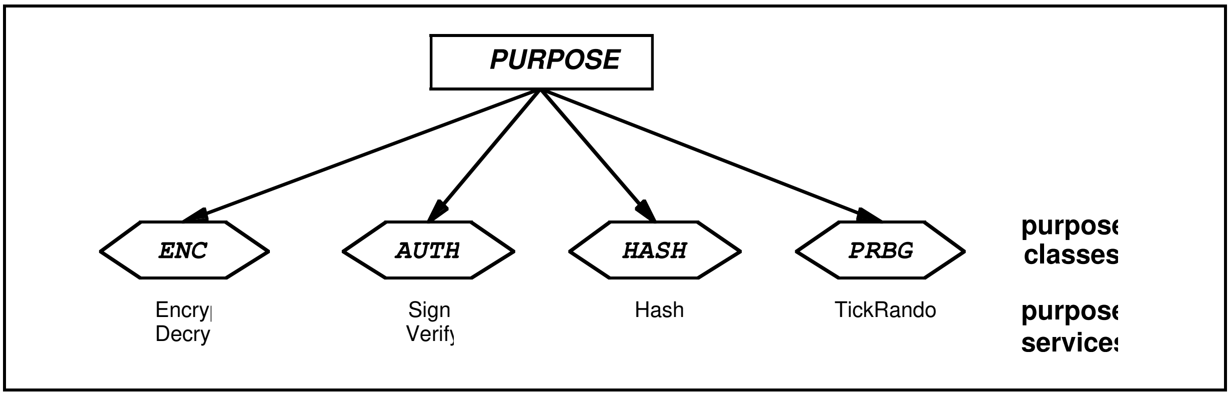

Figure 1: Purpose classes of the CryptoManager++

Each purpose service of a purpose class is realized by a set of methods of this class. One dedicated method (declared as public) makes this service available at the API whereas the others (declared as protected) are used internally to satisfy the service. At the layer of purpose classes, the methods only implement those parts of the desired service which are common to all cryptographic mechanisms being inherited from that particular purpose class. The common parts, for example, of stream ciphers based on block ciphers are transforming a plain text into a sequence of blocks of a fixed length, padding and applying modes of operation (e.g. ECB, CBC or PCBC [MeMa_82]). Methods satisfying those parts of the desired service which are not common to all inherited cryptographic mechanisms (e.g. encrypting one block) are not implemented but are declared as abstract (virtual) methods. They are overloaded at the layer of mechanism classes. Since they do not implement all their methods, purpose classes are abstract classes. As a consequence, objects of these classes can not (and need not) be created.

3.2 Key Classes

Two key classes are provided which are inherited from a common base class *KEYSTRUCTURE*:

(1) class ***SYM*** (symmetric mechanisms)

(2) class ***ASYM*** (asymmetric mechanisms)

Each key class offers certain key services, namely:

(1) ***GenerateKey*** (both class *SYM* and *ASYM*)
generating a secret key (*SYM*) or a private and a public key (*ASYM*) respectively

(2) ***StoreKey*** / ***LoadKey*** (both class *SYM* and *ASYM*):
storing / loading secret key (*SYM*) or public and / or secret key (*ASYM*) respectively

Further services are available concerning the support of memory management. The key classes of the CryptoManager++ are illustrated in figure 2.

Again, each key service is realized by a set of methods of the corresponding class with one dedicated method (declared as public) making this service available at the API and auxiliary methods (declared as protected) used internally to satisfy the desired service. Methods concerning the management of actual keys (which representations depend on the actual cryptographic mechanism) are declared as abstract methods. They are overloaded at the layer of mechanism classes. However, methods concerning the management of common key information are implemented at the layer of key classes. Like purpose classes, key classes are abstract classes.

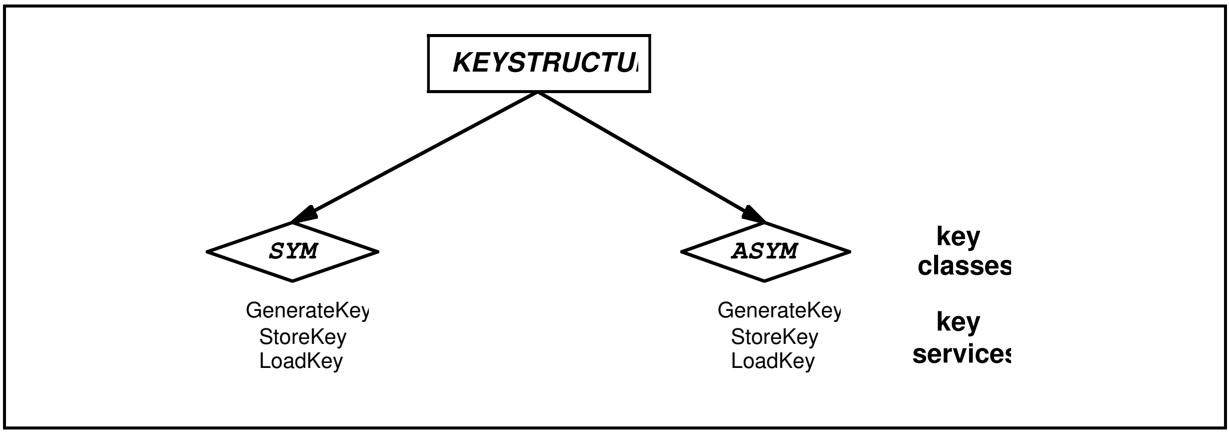

Figure 2: Key classes of the CryptoManager++

3.3 Crypto Classes

Crypto classes have been introduced as a combination of the general properties *cryptographic purpose* and *key structure*. For that reason, a crypto class is generally inherited from a key class *K* and a purpose class *P* (multiple inheritance) and therefore provides key services of *K and* purpose services of *P*. The methods realizing these services are simply inherited from the (abstract) base classes and no further implementation is necessary at the layer of crypto classes. As a result, crypto classes are also abstract classes.

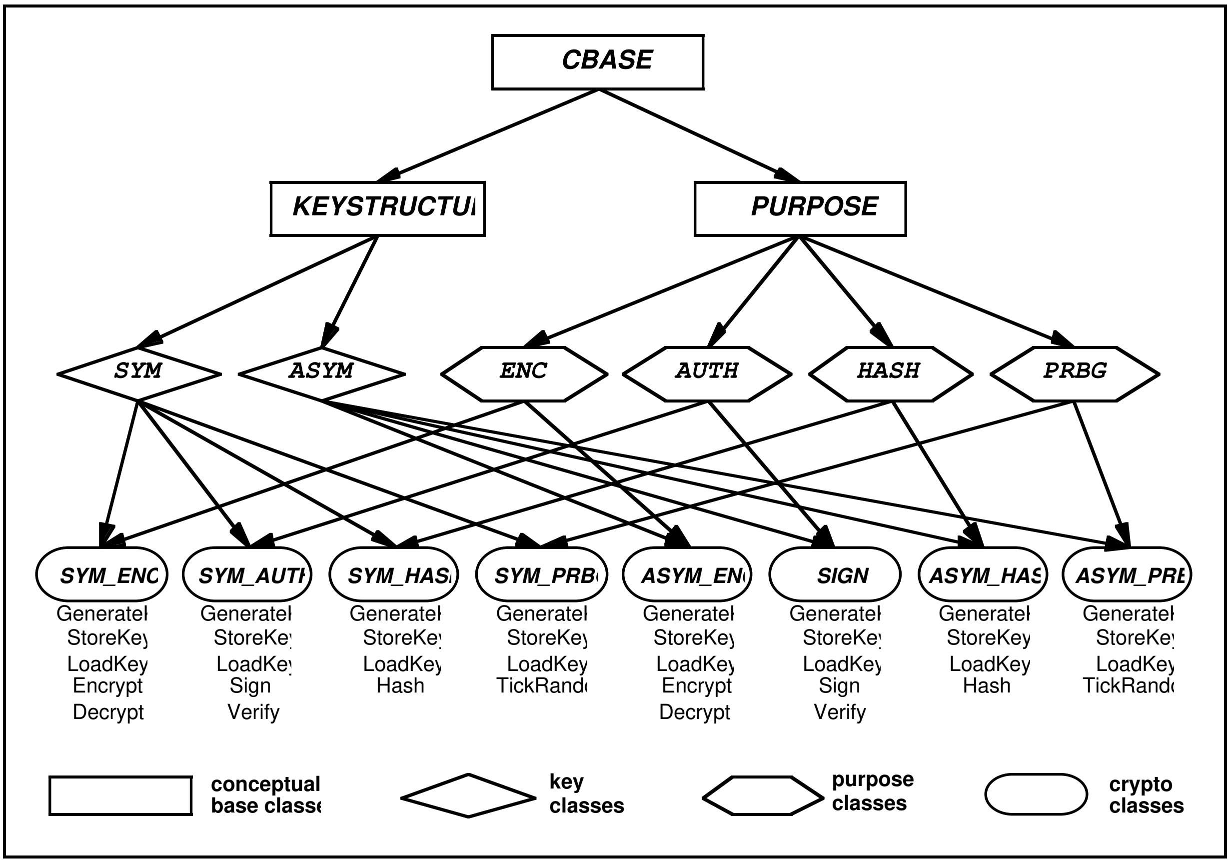

Figure 3: Inherited crypto classes of the CryptoManager++

Figure 3 illustrates the crypto classes of the CryptoManager++ and the base classes from which they are inherited. The classes *KEYSTRUCTURE* and *PURPOSE* are inherited from a common base class *CBASE*.

In detail, the following eight crypto classes are inherited:

(1) class ***SYM_ENC*** (symmetric encipherment mechanisms)

(2) class ***SYM_AUTH*** (message authentication codes)

(3) class ***SYM_HASH*** (symmetric hash mechanisms)

(4) class ***SYM_PRBG*** (symmetric pseudo random bitstring generators)

(5) class ***ASYM_ENC*** (asymmetric encipherment mechanisms)

(6) class ***SIGN*** (digital signature schemes)

(7) class ***ASYM_HASH*** (asymmetric hash mechanisms)

(8) class ***ASYM_PRBG*** (asymmetric pseudo random bitstring generators)

3.4 Mechanism Classes

We regard a specific cryptographic mechanism as one possible implementation of some crypto class *S*. Hence, its mechanism class inherits the key and purpose services of *S* and overloads those parts which are specific for that particular cryptographic mechanism (e.g. encrypting one block). Since all inherited abstract methods are overloaded specifically at the layer of mechanism classes, corresponding objects can be generated. In fact, all services of the CryptoManager++ API specified at the layer of purpose and key classes are provided without exception through objects of mechanism classes.

Table 1 gives an overview of the elementary mechanism classes implemented by the time this paper is written[2]. Notice that the keyless hash functions RIPE-MD, MD5 and SHS are directly derived from their purpose class *HASH* as explained in chapter 2.1.

mechanism class	**cryptographic mechanism**	**inherited from crypto class**
DES	DES [DES_77]	*SYM_ENC*
IDEA	IDEA [LaMa_90]	*SYM_ENC*
RSA	RSA [RSA_78]	*ASYM_ENC*
GMR	GMR [GoMR_88]	*SIGN*
DAMG_HASH	Damgård hash function [Damg_88]	*ASYM_HASH*
RIPE_MD	RIPE-MD hash function [RIPE_93]	*HASH*
MD5	MD5 hash function [Rive_91]	*HASH*
SHS	Secure Hash Standard [FIPS_93]	*HASH*

Table 1: Currently implemented elementary mechanism classes

The mechanism class of a hybrid mechanism is realized as a template class in order to allow different other mechanism classes as servers. Thus, only the respective purpose or crypto classes of the servers are determined. Paying attention to this restriction, the particular mechanism classes of the servers can be chosen arbitrarily when using the constructor of a hybrid mechanism class to generate a new hybrid object (e.g. in case of hybrid encryption, *RSA* and *IDEA* can be chosen as servers for one object and *RSA* and *DES* for another). Corresponding server objects will be generated automatically and become an internal part of the hybrid object. Their key and purpose services are used internally to implement the key and purpose services of the hybrid mechanism class. Thus, the inherited methods of its crypto class are overloaded appropriately. Notice that the cooperation of several cryptographic mechanisms is

[2]The development of the CryptoManager++ cannot be regarded as finished yet. It is planned to implement several more elementary mechanisms, especially the RSA and DSS signature schemes.

not visible at the API. A hybrid mechanism looks to the application programmer just like an elementary mechanism.

In detail, the CryptoManager++ offers the following hybrid mechanism classes:

(1) class ***HYBRID_ENC*** (hybrid encipherment mechanism)
Uses an (arbitrary) symmetric encipherment mechanism to encrypt a plain text and an (arbitrary) asymmetric encipherment mechanism to encrypt the actual symmetric session key which is added to the cipher text. Therefore, it combines the advantages of symmetric (speed) and asymmetric (easy key distribution) encryption.

(2) class ***HYBRID_SIGN*** (hybrid signature scheme)
Uses an (arbitrary) hash mechanism to calculate a message specific hash value of fixed length and an (arbitrary) signature scheme to sign the hash value afterwards. Therefore, it allows the efficient generation and verification of signatures for large messages.

(3) class ***ENC_RedPred*** (asymmetric encipherment mechanism with redundancy predicate)
Uses an (arbitrary) hash mechanism to generate a redundancy predicate which is added to the plain text and an (arbitrary) asymmetric encipherment mechanism to encrypt the plain text afterwards. The redundancy predicate is tested after the decryption in order to prevent chosen cipher text attacks (especially in case of RSA).

(4) classes ***SYM_ENC_AUTH*** and ***ASYM_ENC_AUTH***
Use an (arbitrary) symmetric / asymmetric encipherment mechanism in the PCBC mode. Therefore, it provides encryption and authentication in one pass.

Table 2 gives a detailed overview of the hybrid mechanism classes provided by the CryptoManager++.

hybrid mechanism class	**inherited from crypto class**	**number of servers**	**crypto class / purpose class of servers**
HYBRID_ENC	*ASYM_ENC*	2	*ASYM_ENC*, *SYM_ENC*
HYBRID_SIGN	*ASYM_SIGN*	2	*ASYM_SIGN*, *HASH*
ENC_RedPred	*ASYM_ENC*	2	*ASYM_ENC*, *HASH*
SYM_ENC_AUTH	*SYM_ENC*	1	*SYM_ENC*
ASYM_ENC_AUTH	*ASYM_ENC*	1	*ASYM_ENC*

Table 2: Hybrid mechanism classes of the CryptoManager++

The complete class hierarchy of the CryptoManager++ is shown in figure 4.

One main result of the chosen object oriented design is that the integration of new cryptographic mechanisms is very simple. In order to integrate a new cryptographic mechanism, merely one new mechanism class has to be inherited from a corresponding crypto class S. Since those parts of the key and purpose services which are identical for all specific cryptographic mechanisms inherited from S are implemented at the layer of purpose classes and key classes, only those parts which are specific for the new cryptographic mechanism have to be implemented by the new mechanism class. Existing applications can take advantage from new mechanism classes straight away because the API of the CryptoManager++ does not change by integrating new mechanism classes.

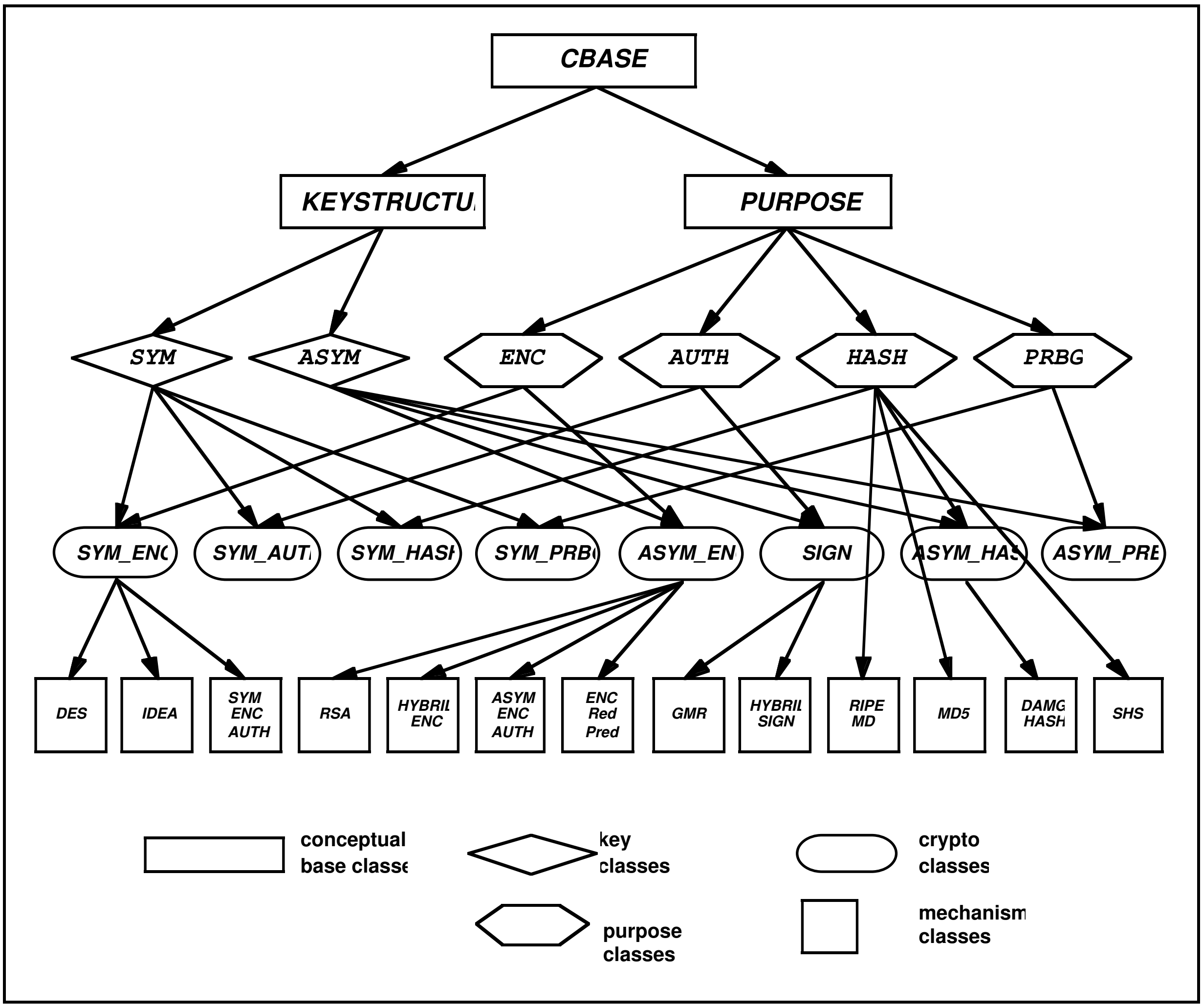

Figure 4: Complete class hierarchy of the CryptoManager++

4. Individually Adjustable Security

The CryptoManager++ allows to flexibly adjust the security of its cryptographic services to the risks involved by an application. The application programmer can choose from a great variety of cryptographic mechanisms which differ with respect to security, efficiency and patent related or legal circumstances. Moreover, each object of a mechanism class can be configurated individually by fixing certain security parameters such as the key length. For example, the following security parameters are provided:

- key length for an object of an asymmetric mechanism class (inherited from key class *ASYM*)
- mechanism classes of the servers for an object of a hybrid mechanism class
- mode of operation (e.g., ECB, CBC, or PCBC) for an object of an encipherment mechanism class (inherited from purpose class *ENC*)

These parameters can be fixed individually when using the constructor of a mechanism class to generate a new object. Default parameters will be used if the constructor is applied without explicit parameters.

5. Bit Stream Operands

Almost all data operands of the CryptoManager++ (externally stored cryptographic keys, messages, plain texts, cipher texts, pseudo random bitstrings, hash values, digital signatures) are dynamic operands (i.e. bit streams which lengths are not fixed at the compile time). A dynamic operand is represented by a tuple (a, l) with a denoting its start address and l denoting its length.

5.1 Cluster Operands

For some services of the CryptoManager++, inputs should be allowed which lengths potentially exceed the amount of available main memory (e.g. a large plain text stored on an external device). In this case, the CryptoManager++ offers the possibility to partition a complete input (bit stream) $in = (a, l)$ into several *cluster operands* $c_1 = (a_1, l_1), \ldots, c_n = (a_n, l_n)$ and to process them sequentially afterwards[3]. The same concept is used for outputs which lengths potentially exceed the available main memory. Partition into cluster operands and corresponding services allowing to process them are provided for messages, plain texts and cipher texts. Externally stored cryptographic keys, pseudo random bitstrings, hash values and digital signatures are regarded as *complete operands* meaning that they are always processed by a single service call. A partition into clusters is not provided for them.

5.2 Processing Clusters

An input which length exceeds the amount of available main memory is processed in 4 steps (as illustrated in figure 5):

(1) allocate memory for an input and an output buffer of equal size

(2) select the cryptographic service of the CryptoManager++ to be applied

(3) determine a cluster partition of the input (i.e. cluster lengths and number)

(4) for all resulting input clusters do

  (4.1) load cluster into input buffer

  (4.2) apply cryptographic service

  (4.3) store content of output buffer (resulting output cluster)

The arbitrarily fixed size of the input and output buffer (which is only limited by the amount of memory) determines the maximum length of each cluster (*maximum cluster length*).

The length of a single cluster c_i belonging to a cluster partition $c_1, \ldots, c_n$ of an input must not always correspond to the fixed maximum cluster length. For example, its length must be chosen so small that the resulting output cluster will still fit into the output buffer. In detail, the length of each single input cluster c_i depends on the service to be applied, on the particular mechanism class, on the actual key length and on the cluster position i. For the latter, a distinction between "first", "middle" and "last" clusters turned out to be sufficient. For example, using RSA encryption, it has to be taken into consideration that the resulting cipher text clusters will be larger than the plain text clusters since the plain block length is chosen smaller than the cipher block length (which itself depends on the key length). Furthermore, the lengths of all plain text clusters except of the last one have to be a multiple of the plain block length and the last plain text cluster will be expanded because it is padded. In order to release the application programmer from such complicated considerations, the CryptoManager++ provides special services which precalculate the cluster lengths and the resulting number of clusters for any specified input length.

[3]The application programmer must be able to partition an input into cluster operands explicitly because the CryptoManager++ does not directly support operands outside the main memory.

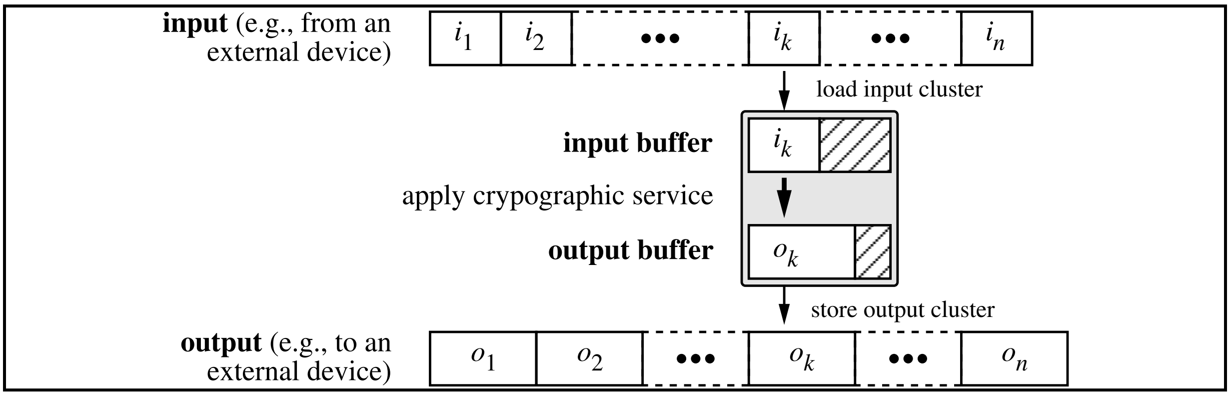

Figure 5: Processing clusters sequentially

6. Portability and Efficiency

Our intention was to implement the CryptoManager++ in a way that allows to adapt it easily to different computer platforms such as Sun, Apple Macintosh or PC. However, this objective is in conflict with the objective of efficiency: to achieve maximum portability, implementation has to be done completely in a standardized high level programming language whereas a more efficient implementation can be achieved by using assembler language cleverly. Our compromise was to do the implementation almost completely in C++, using a minimal assembler kernel (currently for Sun SPARCStations).

Performance tests executed on a Sun SPARCStation 5 resulted, for example, in a hash performance of about 28 MBit/s for MD5, 25 MBit/s for RIPE-MD and 11 MBit/s for SHS. Concerning encipherment mechanisms, an encryption performance of about 1,6 MBit/s for IDEA, 1 MBit/s for DES and 25 KBit/s (decryption: 1 KBit/s) for RSA[4] with a key length of 1024 bit were achieved (all in CBC mode). Further tests showed that the overhead caused by inheritance and overloading is less than 4%.

7. Conclusion

The most important characteristic of the CryptoManager++ is its object oriented design which results in the following main advantages:

- **uniform and convenient use of cryptographic services**
  Though the API of the CryptoManager++ provides services for a great variety of cryptographic mechanisms, all cryptographic services offered can be used uniformly. Especially, this applies to hybrid cryptographic mechanisms .
- **easy integration of new cryptographic mechanisms**
  In order to integrate a new cryptographic mechanism, merely one new mechanism class is inherited from a corresponding crypto class. Only those parts which are specific for the new cryptographic mechanism have to be implemented by the new mechanism class. The integration of a new cryptographic mechanism does not affect the API.
- **lean implementation**
  Thanks to the concept of inheritance, identical parts of cryptographic mechanisms' key and purpose services had only to be implemented once at the layer of key and purpose classes.

Concerning the further development in the near future, it is planned to implement several more cryptographic mechanisms, especially the signature schemes RSA [RSA_78] and DSS [DSS_92] as well

[4]Our multi precision arithmetic library used to implement RSA is not completely optimized yet.

as pseudo random bitstring generators like BBS (x^2-mod-n generator [BlBS_86]). Furthermore, we start working on providing several import and export filters which will allow the CryptoManager++ to support different protocol formats like X.500 [ISO_90], PKCS [Kali_91], PGP [Garf_95] or GSS [XOpe_94].

Acknowledgements

We acknowledge the excellent work of Ralf Kanne who designed and implemented the first version of the CryptoManager++. Furthermore, we thank Frank Sudholt for further developing the CryptoManager++ and especially for providing the results of the performance tests. Birgit Pfitzmann contributed valuable ideas during the designing stage (and after it).

References

BlBS_86 L. Blum, M. Blum, M. Shub: A Simple Unpredictable Pseudo-Random Number Generator; SIAM J. Comput. 15/2 (1986) 364-383.

Bras_88 Gilles Brassard: Modern Cryptology - A Tutorial; LNCS 325, Springer-Verlag, Berlin 1988.

Damg_88 Ivan Bjerre Damgård: Collision free hash functions and public key signature schemes; Eurocrypt '87, LNCS 304, Springer-Verlag, Berlin 1988, 203-216.

DES_77 Specification for the Data Encryption Standard; Federal Information Processing Standards Publication 46 (FIPS PUB 46), January 15, 1977.

DiHe_76 Whitfield Diffie, Martin E. Hellman: New Directions in Cryptography; IEEE Transactions on Information Theory 22/6 (1976) 644-654.

DSS_92 CACM (some editing): The Digital Signature Standard Proposed by NIST; Communications of the ACM 35/7 (1992) 36-40.

FIPS_93 Secure Hash Standard; Federal Information Processing Standards Publication 180 (FIPS PUB 180), May 11, 1993.

FIPS_94 Cryptographic Service Calls (DRAFT); Federal Information Processing Standards Publication (FIPS PUB xxx), May 23, 1994.

Garf_95 Simson Garfinkel: PGP Pretty Good Privacy; O'Reilly & Associates, Sebastopol 1995.

GoMR_88 Shafi Goldwasser, Silvio Micali, Ronald L. Rivest: A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks; SIAM J. Comput. 17/2 (1988) 281-308.

ISO_90 ISO/IEC 9594-1: Information technology - Open Systems Interconnection - Specification - The Directory - Part 1: Overview of concepts, models and services; ISO International Standard, First edition 15.12.1990.

Kahn_67 David Kahn: The Codebreakers, The Story of Secret Writing; Macmillan Publishing Co., New York, 1967.

Kali_91 Burton S. Kaliski Jr.: An Overview of the PKCS Standards; RSA Data Security, Inc., 10 Twin Dolphin Drive, Redwood City, CA 94065, USA, June 3, 1991.

LaMa_90 Xuejia Lai, James L. Massey: A Proposal for a New Block Encryption Standard; Eurocrypt '90 – Abstracts, Scanticon, Århus, Denmark, May 1990, 361-365.

MeMa_82 Carl H. Meyer, Stephen M. Matyas: Cryptography - A New Dimension in Computer Data Security; (3rd printing) John Wiley & Sons, 1982.

RIPE_93 RIPE Consortium: RIPE integrity primitives Part 1: Final report of RACE 1040; Centrum voor Wiskunde en Informatica, Computer Science/Departement of Algorithmics and Architecture, Report CS-R9324 , April 1993.

Rive_91 Ronald L. Rivest: MD5 – New Message-Digest Algorithm (Abstract); presented at Crypto '91, Rump Session, 13. 8. 1991.

RSA_78 R. L. Rivest, A. Shamir, L. Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems; Communications of the ACM 21/2 (1978) 120-126, reprinted: 26/1 (1983) 96-99.

Schn_93 Bruce Schneier: Applied Cryptography: Protocols, Algorithms, and Source Code in C; John Wiley & Sons, New York 1994.

XOpe_94 X/Open Company Ltd: Generic Security Service API (GSS-API) Base, Preliminary Specifications; X/Open Document Number: P308, X/Open Company Ltd, Apex Plaza, Forbury Raod Reading Berkshire RG1 1AX, UK, (e-mail: XoSpecs@xopen.co.uk), ISBN: 1-85912-025-3.