Lecture Notes for Advanced Web Security 2016

Part 6 — Data Representation and PKI

Martin Hell

1 ASN.1

ASN.1 (Abstract Syntax Notation One) is a standardized way of representing data. The purpose is to define a way to represent objects such that they can be created, transferred, and read by different applications on different machines and platforms. ASN.1 dates back to 1984 when it was standardized as X.409. In 1988, the encoding rules were separated into its own standard and ASN.1 moved to the new standard X.208. Several changes were later made and ASN.1 again moved to a new standard, X.680, which was approved 1995. This is still the current ASN.1 standard. The most recent edition of X.680 was published in 2008 [9]. ASN.1 with its encodings is actually described by several documents allocating the space X.680-X.699. There are (at least) two good books on ASN.1 freely available [14, 6]. The interested reader is referred to these, as well as the standards documents, for a more in-depth treatment.

ASN.1 defines an abstract way of describing the objects, but does not define how these are actually encoded. Depending on the situation, a suitable encoding rule can be applied. Encoding rules that can be used include e.g., BER, DER, and XER. These will be described later. ASN.1 can be compared to ABNF and XML Schema, which are also used to describe how data should be described. ABNF is typically encoded using ASCII text while XML Schema is encoded using XML.

It is convenient to define standards using ASN.1 as it allows for precise specifications, but using a very compact format. Being able to interpret ASN.1 can allow an engineer to quickly understand the details of a specification. The LDAP standard is an example of a standard written in ASN.1. LDAP typically uses BER as the transfer encoding. Digital certificates, and CRLs in X.509 are described using ASN.1 and encoded with DER. The Cryptographic Message Syntax (CMS), used in e.g., S/MIME is also described using ASN.1.

Formally, ASN.1 defines data types and values. A data type is a category of information, e.g., a number, a string, a picture or a video, while a data value is an instance of a type. ASN.1 defines basic types and values and rules for combining them, producing more complex types and values.

The example built in this section will show how to represent data for a stu-

Table 1: Built-in types and their universal tag in ASN.1

Type	Description	Tag
`BOOLEAN`	A Boolean which can only take one of the values 0 or 1.	1
`INTEGER`	An integer of any size.	2
`DATE`	A date in the format YYYY-MM-DD.	
`BIT STRING`	Binary data that does not have to be a multiple of 8 bits.	3
`OCTET STRING`	Binary data that is a multiple of 8 bytes.	4
`UTF8String`	A string encoded with UTF-8, allowing the use of Unicode.	12
`NumericString`	A string of numbers 0-9 and the "space" character.	18
`VisibleString`	ASCII characters other than control characters.	26

dent at LTH. Note that this will not necessarily be the best way of representing this data for a student. The purpose is instead to illustrate some different features of ASN.1. As a starting point, below the type `PhoneNumber` is defined.

```
PhoneNumber ::= SEQUENCE {
  phoneID          VisibleString,
  number           NumericString
}
```

The type `PhoneNumber` consists of one `VisibleString` type named "phoneID" and a `NumericString` type named "number". The names in this case are just names for the different elements in the sequence and are chosen by the designer in order to give a clear description of the protocol for the readers. However, the computer does not care about these names at all.

The phoneID can be used to define what type of phone number it is (mobile, home, work etc) and the number is the actual number to dial. The types `VisibleString` and `NumericString` are types built into ASN.1 and pre-defined. There are several pre-defined types and some common examples are given in Table 1. The tag associated with each type will be discussed later.

The `SEQUENCE` is a type defining a list of types. The types defined by `SEQUENCE` in the example above are `VisibleString` and `NumericString`, which in this case are called *component types*. There are five types that can be used to define component types. These are given in Table 2.

It follows from the definitions that `SET` and `SET OF` can always be replaced by `SEQUENCE` and `SEQUENCE OF` respectively, removing some freedom. Still, if there is no important advantage in using `SET` and `SET OF`, it is often better to use `SEQUENCE` and `SEQUENCE OF` instead since it can make the decoding faster. As one person can have many phone numbers, the previous example can be

Table 2: Types used to define component types.

Type	Description	Tag
`CHOICE`	A list of types and the value is one of the listed component types.	*
`SEQUENCE`	An ordered list of component types. The values must be given in the specified order.	16
`SEQUENCE OF`	Same as `SEQUENCE` but all components types have to be of the same type.	16
`SET`	An un-ordered list of distinct component types, i.e., the values can appear in any order.	17
`SET OF`	An un-ordered list of a single component type, i.e., values can be in any order but they are always of the same type.	17

extended to define a type called `Student` which has a name and has a list of phone numbers.

```
Student ::= SEQUENCE {
  studentName      UTF8String,
  phone            SEQUENCE OF PhoneNumber
}

PhoneNumber ::= SEQUENCE {
  phoneID          VisibleString,
  number           NumericString
}
```

Now, we have a type `Student` consisting of a UTF8String with name studentName and a sequence of the type `PhoneNumber`, named phone. Since the `PhoneNumber` type is not built into ASN.1, it must be defined as well. This definition is given separately. Inside the `Student` type definition, the string "PhoneNumber" is just a name referring to a type, and the computer does not care about this name. It can be arbitrarily chosen by the designer, similar to the other names. If the type `PhoneNumber` is used in several places, defining `PhoneNumber` separately is convenient since the type is defined once and can be reused wherever needed. However, if it is just used once, then it is often more convenient to just define it inside the `Student` definition.

Furthermore, it is possible to put constraints on the types. This can be used to put limitations on the values for the types and it can also give a more compact encoding. Below, the explicit definition of the `PhoneNumber` type has been removed and the studentName can be of size at most 40.

```
Student ::= SEQUENCE {
  studentName      UTF8String (SIZE(1..40)),
  phone            SEQUENCE OF SEQUENCE {
                     phoneID     VisibleString,
                     number      NumericString
                   }
}
```

Another type that is common is the `ENUMERATED` type. The value of this type is a named integer. As usual the name is not important to the computer, only the integer value. On the other hand, the integer is not assumed to carry any semantics in the protocol. This is what the name is for, similar to the `enum` types in Java and C. It is possible to include comments by using two hyphens (`--`). The comment ends at the end of the line, or when a new set of two hyphens occur. The perhaps familiar /* and */ can also be used for comments and these can span several lines.

An `OBJECT IDENTIFIER` is a very special type as its value must be globally unique in the sense that it always refers to the exact same thing. This can be a shop item, a value representing a protocol or an algorithm. This is accomplished by defining a tree structure, in many ways similar to the DNS structure, where each node has one value and is administered by some authority. By contacting an authority it is possible to get a child of its node and create your own subtree, usually called arc in ASN.1. Most nodes also has a corresponding name in order to clarify what they represent or who they belong to. As an example, the SHA-256 hash algorithm object identifier (OID) is defined as below.

```
id-sha256  OBJECT IDENTIFIER  ::=  { joint-iso-itu-t(2)
        country(16) us(840) organization(1) gov(101) csor(3)
        nistalgorithm(4) hashalgs(2) 1 }
```

Giving types a default value can be done using the `DEFAULT` keyword. This will be the assumed value if it is not included. Another keyword is `OPTIONAL`, which can be used to define that a value is optional and does not have to exist. Below, the example has been expanded to include an `OBJECT IDENTIFIER` named studentID and an `ENUMERATED` type named program. In addition, more information about the student has been added as a `SEQUENCE` type named status. This includes four component types, an `INTEGER` for the amount of hp-points, a `REAL` for the average grade, a `BOOLEAN` to define if the student is currently active or not, defaulting to true, and an optional `UTF8String` used to hold additional comments about the student. The StudentStatus is defined separately just for the sake of the example. Information about courses that the student has finished is also added, consisting of a reference to the type `Course` and the type `VisibleString` that can take one out of five specified values.

```
Student ::= SEQUENCE {
  studentName      UTF8String (SIZE(1..40)),
```

```
  studentID        OBJECT IDENTIFIER,
  program          ENUMERATED {
                     C(0), D(1), E(2), F(3), Pi(4)
                   },
  phone            SEQUENCE OF SEQUENCE {
                     phoneID     VisibleString,
                     number      NumericString
                   },
  status           StudentStatus,
  finishedCourses  SEQUENCE OF SEQUENCE {
                     courseInfo  Course,
                     grade       VisibleString
                                 ("G", "VG", "3", "4", "5")
}

StudentStatus ::= SEQUENCE {
  hp               INTEGER,
  avgGrade         REAL (3.00..5.00),
  active           BOOLEAN DEFAULT TRUE,
  comment          UTF8String OPTIONAL
}
```

Note that the definition of the type `Course` is not provided. This will be fixed as part of the next example. The numbers in parentheses in the `EMUMERATED` gives the number that each name should be associated with. These numbers are optional, but were required in older version of ASN.1 (before 1995). If no numbers are given, the automatically assigned numbers will start from zero and increment for each additional name in the enumeration. It is also possible to assign some numbers and let the rest be automatically assigned. The rules for this can be found in the standard [9].

Type definitions in ASN.1 are grouped in modules. Complex protocols or specifications can consist of several modules. It is possible to import type definitions from other modules. By default any definition can be imported, but it is possible to restrict the types that can be imported by using the `EXPORT` keyword.

```
Student-module
 { ... }
DEFINITIONS
         AUTOMATIC-TAGS ::=
BEGIN

EXPORTS Student;

IMPORTS Course FROM LTH-module { ... };
```

```
Student ::= SEQUENCE {
        .
        .
        .
}

END
```

The first line gives the name of the module. This is followed by the *module identifier* in curly brackets. This is an OID which identifies the module and it has to be globally unique just as the `OBJECT IDENTIFIER` type. However, the module identifier is optional in order to allow anyone to use ASN.1 without registering a module identifier. The keywords DEFINITIONS, BEGIN and END are just used to separate the different parts of the module. AUTOMATIC-TAGS defines how tagging will be done and automatic tagging is the usual and recommended way of tagging. The alternative options here are IMPLICIT and EXPLICIT. In the example, the definition of the type `Course` is imported from another module called LTH-module. Note the semi-colon concluding the `IMPORT` statement. This is also used after an `EXPORT` statement.

It is of course possible to build much more complicated ASN.1 specifications than this, but the example above should give the reader enough introduction to understand the basics.

1.1 Tagging

Every ASN.1 type has a tag associated with it. The tags given in Tables 1 and 2 are of the class UNIVERSAL. There are in total four classes, the other being APPLICATION, PRIVATE and context-specific. Apart from UNIVERSAL, the context-specific class is the most common and important. The tagging is important for some encoding rules, e.g., BER, DER and CER, since the data is encoded by giving the tag. Thus, when an INTEGER is encoded, this is specified using the UNIVERSAL tag number 2. When writing the specification, it is possible to change the tag used. Consider the following example.

```
studentName ::=      INTEGER

studentName ::=  [1] IMPLICIT INTEGER

studentName ::=  [1] EXPLICIT INTEGER
```

In the first case the tag would be 2 with class UNVERSAL. Using the square brackets, this can be overridden. The second line changes the tag to 1. The class in this case is context-specific. If another class is desired, this must be explicitly written. IMPLICIT here means that the universal tag is replaced by the context-specific tag. If EXPLICIT is used instead, an outer tag environment is added in addition to the original tag.

1.2 Encoding Rules

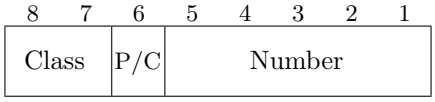
When the abstract values are to be sent over a network or a channel, they need to be represented by bits. Exactly how to represent the values is defined by the encoding rules. There are several different rules defined and depending on the situation, a particular encoding rule might be the best choice. Deciding which encoding rules to use might be something that is negotiated before sending the data, or it might be fixed in advance for the protocol. At the same time, new encoding rules, optimized for specific situations, can be adopted by a protocol or application without making any changes to the ASN.1 specification. To make things even more flexible, one encoding of data can be used in one situation and then the data can be re-encoded to make it suitable for another situation.

This section will give an overview of the different encoding rules that are possible and how they differ.

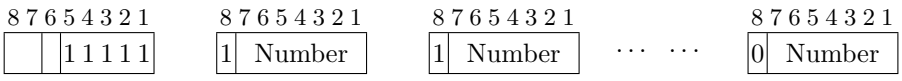
1.2.1 BER

The Basic Encoding Rules (BER) are the original encoding rules that were defined for ASN.1. They were at first described in the same standards document as ASN.1 (X.409, 1988) but were later moved to its own standard (X.209, 1988), when ASN.1 became the X.208 standard. The main characteristic of the BER encoding is that it is simple, but it is quite inefficient, at least compared to e.g., PER. Still, it is much more efficient than text based encodings, which were otherwise common. The most up-to-date specification is given in [10].

BER uses a Type-Length-Value (TLV) approach in the encoding. The *type* defines what is being encoded (SEQUENCE, INTEGER UTF8String, etc), the *length* defines the length of the value field, and the *value* is the actual value that is being encoded. Each type is associated with a tag. Sometimes a fourth component, representing the end of contents, is used also. This depends on how the length is encoded. The tags for some types are given in Tables 1 and 2 (for `CHOICE`, the tag is that of the choice made). The tag is given in an identifier byte of the form



If the number is larger than 30, the identifier is expanded to several octets. The first octet specifies class and P/C and the last 5 bits are set to ones to indicate that the identifier contain several octets.



The minimum number of octets must be used here. The class is the class that the type belongs to. In the examples given here, only universal class is

used. This is given by `00` in these two bytes. The bit P/C defines if the type is primitive or constructed. Primitive means that the value is the actual value of the type, such as in the case with `BOOLEAN, INTEGER` etc, while constructed means that the value is itself a series of TLV encodings. As two examples, the identifier for the type `SEQUENCE` is 0x30 and for `BOOLEAN` it is 0x01. In the following, all encodings will be given in hexadecimal numbers.

The length encoding can be done in one of three possible ways.

1. **Short definite form.** The length is encoded using one octet with the MSB set to zero and the remaining 7 bits represent the length of the content (in octets). As an example, length 20 can be encoded as 00010100. If the length is larger than 127, the short definite form can not be used.

2. **Long definite form.** To encode longer lengths than 127, several octets are used. The first octet has MSB set to one and the remaining 7 bits represent the number of octets that follows and that will give the length. As an example, length 131 would be encoded as 10000001 10000011. However, it can also be encoded as 10000010 00000000 10000011. Similarly, length 20 can be encoded as 10000001 00010100. Which way to encode is chosen by the sender.

3. **Indefinite form.** This form uses two terminating all-zero octets to indicate end-of-contents. The use of the indefinite form is determined by letting the length octet have the value 10000000.

For constructed encodings, either form can be used but for primitive encodings short or long *definite* form must be used. This removes the problem that the content may happen to consist of two consecutive all-zero octets.

How the values are encoded will depend on the actual type of the value. For details, refer to the X.690 standard [10], but a few simple examples are given here. These examples will also be used to illustrate the difference between BER, DER and CER.

A `BOOLEAN` is encoded with an all-zero octet for the value FALSE, while the value TRUE is encoded with any other octet (01-FF). Which way to encode TRUE is chosen by the sender. Thus, the full TLV encoding of TRUE can be 01 01 FF, but it can also be 01 81 01 01 (using long definite form and 01 to represent true).

An `INTEGER` is encoded as the two's complement of the integer value, using as few octets as possible. The full TLV encoding of -2 can be 02 01 FE.

A `VisibleString` is encoded as the ASCII representation of the characters. The string "string" has ACSII representation 73 74 72 69 6e 67 and can be encoded as 1A 06 73 74 72 69 6e 67. A `VisibleString`, and also other types that are strings, can optionally be fragmented into smaller strings, each fragment sent as its own TLV. Then, the outer TLV is constructed (P/C=1) and the inner are primitive. Splitting "string" in two equal parts, and using indefinite form for the outer TLV, will give the encoding 3A 80 1A 03 73 74 72 1A 03 69 6e 67 00 00. The fragmentation can be done at any point and can also be nested, making very many encodings for the same string possible.

1.2.2 DER and CER

The BER encoding rules has in many cases several different possible encodings for the same thing. The length could be encoded in many ways, `BOOLEAN` has many alternatives for TRUE and strings can be fragmented in any way.

The Distinguished Encoding Rules (DER) and the Canonical Encoding Rules (CER) are a subset of BER. They are both canonical, meaning that they describe unique encodings for the data. DER and CER are both described in the same standard as BER, X.690. [10].

To provide unique encodings, several decisions have been made to remove the alternatives in BER.

- For the type `BOOLEAN`, TRUE is always encoded as `FF`.
- In a `SET` the values are sorted in order of the tag number.
- In DER, the shortest possible definite form is always used. In CER, if encoding is constructed, indefinite length form is always used and for primitive encodings the shortest possible definite form is used.
- In DER, strings are only given in primitive form, i.e., they are not fragmented. In CER, strings are always split into fragments of size 1000 (except the last if the string is not a multiple of 1000 octets).

1.2.3 PER

BER, DER, and CER gives a quite compact representation of the ASN.1 data, especially compared to a text-based approach. However, it still sends quite much data that is unnecessary and should not really be needed. There is e.g., no point in encoding the type since this is clear from the ASN.1 specification anyway. There is also no point in wasting 8 bits to transmit a Boolean value. One bit should be enough. If we encode an `INTEGER` that can only take eight different values, it should be enough with three bits. There are several more examples, and the Packed Encoding Rules (PER) has been standardized in order to provide an encoding that requires as few bits as possible. The standard is given in X.691 [11] which first appeared in 1995. In the case with `INTEGER`, if it can take one of the eight values 27-34, it can still be encoded using only three bits. In this case the bits will give the offset from 27.

PER are suitable in an environment where space is very limited, e.g., smart cards. The size of a typical PER encoding is roughly half of the size of the corresponding BER, DER or CER encoding, but the exact values are of course dependent on the situation. Without going into any details here, consider the following (extreme) example.

```
Example ::= SEQUENCE {
  a             BOOLEAN,
  b             INTEGER (27..34),
  c             SEQUENCE {
                      c1     BOOLEAN,
                      c2     BOOLEAN,
                      c3     INTEGER (50..53)
                }
}
```

Using DER, the encoding would require several octets while when using PER only one octet is needed (we do not have to specify type or length anywhere). However, if there would be no constraints on the integers there would have to be some length encoding of the `INTEGER` type, requiring a few more bytes. This shows that the constraints are important in this case and aids in optimizing the PER encoding. PER itself comes in a few different flavors. A canonical variant is called Canonical PER.

1.2.4 XER

It is clear that ASN.1 is much more compact than text-based protocols. Still, in many situations compactness of encoding is not of primary concern. XML is a very widespread markup language with many supporting tools. Being able to express ASN.1 data in XML can be very useful when moving the data between systems and e.g., databases. The XML Encoding Rules (XER), standardized in 2001 [12], is a standard for just that. It typically requires many more octets than other encoding rules since it is completely text-based. There are three main variants of XER, namely basic, canonical and extended XER.

2 Representing protected data using CMS

The Cryptographic Message Syntax (CMS) is an IETF standard defining how to represent protected data. It supports several different types of protection, such as signatures, MACs and encryption. The representation of the data is described using ASN.1. The current version of CMS is based on PKCS #7, which was a standard promoted by RSA Security Inc. The PKCS #7 description can be found in RFC 2315, which has informational status. PKCS #7 was picked up by IETF, who continued to develop it and released it as an IETF standard. The most recent specification of CMS can be found in RFC 5652 [8].

There is one protection content, called ContentInfo, which encapsulates a type. The ContentInfo is described as follows.

```
ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content [0] EXPLICIT ANY DEFINED BY contentType
}

ContentType ::= OBJECT IDENTIFIER
```

It first defines the type used and then the rest of the content is defined by the type. Being an OID, the type is identified by a globally unique string. There are six different types described in RFC 5652.

- Data Type. This is just a string of bytes.
- Signed-Data Type. This is data with an additional digital signature. More than one signature can be present.
- Enveloped-Data Type. This is encrypted data together with the encrypted encryption key. There can be several recipients and then the encryption key is encrypted for each recipient. Each recipient then uses its own key to decrypt the encryption key.
- Digested-Data Type. This is data together with a message digest of the data. It adds integrity to the data but not authentication or encryption.
- Encrypted-Data Type. This is encrypted data. It does not contain an encrypted encryption key, just the algorithm used to encrypt the data.
- Authenticated-Data Type. This is data together with a MAC of the data. Similar to the enveloped-data type, it also contains an encrypted authentication key, encrypted with a key shared with the recipients.

Refer to the specification for the exact identifier for each type. While the ContentInfo can only hold one type, the different types can themselves be nested. Typically, the data type is nested inside one of the other five types.

2.1 The Signed-Data Type

The exact content of the resulting data will depend on which type is used, but there are many similarities between the different types. This section will use the signed-data type as an example for looking at further details on how the data is constructed. A similar approach is used for the other types. The Signed-Data type can be used to represent digitally signed data. One feature is that it supports several signers for the same data. The PKI used for public keys in CMS is PKIX, i.e., the public keys are given in X.509 certificates. The ASN.1 syntax given in this section is slightly more compact than the one found in RFC 5652, and some types are left undefined. Consult the specification for the complete ASN.1 description.

The OID for the content type SignedData is given by

```
id-signedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
  us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2 }
```

and the type SignedData has ASN.1 definition

```
SignedData ::= SEQUENCE {
  version            CMSVersion,
  digestAlgorithms   SET OF DigestAlgorithmIdentifier,
  encapContentInfo   SEQUENCE {
                        eContentType ContentType,
                        eContent [0] EXPLICIT OCTET STRING OPTIONAL },
  certificates [0]   IMPLICIT CertificateSet OPTIONAL,
  crls [1]           IMPLICIT RevocationInfoChoices OPTIONAL,
  signerInfos        SET OF SignerInfo
}
```

The version is used to give the version of the syntax. The digestAlgorithms is a set of hash algorithms that have been used by the different signers. The algorithm used by each signer will later be given in the SignerInfo, but to allow a receiver to make only one pass through the complete data, the union of all algorithms used can be given here. The encapContentInfo is the data that is signed. In the simplest case, this is just a data type, but it can be any other content type as well, specified by eContentType. The inclusion of certificates and CRL info here is optional but can be used if wanted. The set of certificates can be used to provide the receivers with trusted root certificates and certificate chains that might be needed to verify the signers' certificates. The CRLs contain information about revocation of certificates. This can be used to convince the receivers that the signers' certificates are valid. The SignerInfo gives information about the different signers. The type SignerInfo is defined as follows.

```
SignerInfo ::= SEQUENCE {
  version              CMSVersion,
  sid                  SignerIdentifier,
  digestAlgorithm      DigestAlgorithmIdentifier,
  signedAttrs [0]      IMPLICIT SignedAttributes OPTIONAL,
  signatureAlgorithm   SignatureAlgorithmIdentifier,
  signature            SignatureValue,
  unsignedAttrs [1]    IMPLICIT UnsignedAttributes OPTIONAL }


SignerIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier
}
```

The version will depend on which type of SignerIdentifier is used. The SignerIdentifier is used to specify which certificate (and thus public key) should be used

to verify the signature. Either the issuer name and the certificate serial number is used to identify the certificate, or a key identifier is used. The subject key identifier is an extension in X.509. The digestAlgorithm is the hash algorithm used by this signer, when computing the value to sign. SignedAttributes are the attributes that are signed if the data to sign is more than just the data type. The signatureAlgorithm is the OID of the algorithm that this signer has used. The signature is the actual digital signature and the UnsignedAttributes are additional attributes that are not covered by the signature.

The encoding rules used by CMS are BER with indefinite length encoding. However, signed attributes must be DER encoded so that the recipient can easily handle attributes that are unknown. The DER encoding offers a one-to-one mapping between the abstract values and the encoding so the recipient can just see an unrecognized value as a fixed string (the DER encoding).

3 Representing Personal Information (PKCS #12)

PKCS #12 is a standard to represent personal information. The standard has been published by IETF (though not as an IETF standard) as RFC 7292 [**?**]. It uses parts of the CMS (PKCS #7) standard in order to represent data, but focuses on holding in particular personal data, such as private keys or certificates. It also builds on PKCS #8, which defines how to represent private key information. PKCS #8 has later been extended to its own content type in RFC 5208 so that it can be used directly in CMS.

PKCS #12 supports two integrity and two privacy modes.

- Public-key privacy mode. The personal information is protected using the recipients public key.
- Password privacy mode. The personal information is protected using a symmetric key derived from a password.
- Public-key integrity mode. The personal information is protected with a signature using the private key of the source.
- Password integrity mode. The personal information is protected with a MAC with a key derived from a password.

These possibilities allows four combinations of integrity and privacy. All modes are allowed. The top-level type, which is the one that is ASN.1 exported is the PFX (Personal Information Exchange).

```
PFX ::= SEQUENCE {
  version     INTEGER,
  authSafe    ContentInfo,
  macData     MacData OPTIONAL
}
```

The macData is optional and is only used when password integrity mode is used. The MacData type is defined as follows.

```
MacData ::= SEQUENCE {
  mac         DigestInfo,
  macSalt     OCTET STRING,
  iterations  INTEGER DEFAULT 1
}
```

In this type, the mac is just the MAC data, while macSalt is a salt used to derive the MAC key and iterations is used to define how many times the key generation algorithm is repeated when deriving the MAC key. This is used as input to the PBKDF2 key derivation function. Returning to the PFX type, the authSafe is a ContentInfo type as defined by CMS. This can be either of type SignedData if public-key integrity mode is used or it is of type Data if password integrity mode is used. The content of the authSafe ContentInfo type shall be the BER encoded value of type AuthenticatedSafe, which is a sequence of ContentInfo types.

```
AuthenticatedSafe ::= SEQUENCE OF ContentInfo
```

Which ContentInfo types are used here will depend on which privacy mode is used. If data should not be encrypted at all, e.g., due to restrictions in cryptography used, the type Data is used. For public-key privacy mode the EnvelopedData type is used and for password privacy mode the EncryptedData type is used. These types themselves have content. For plaintext data this is just the data, but for privacy protected data this is a BER-encoded SafeContents type.

```
SafeContents ::= SEQUENCE OF SafeBag
```

The SafeBag type then holds the actual information. There are six types of SafeBags defined.

- KeyBag. A PKCS #8 private key.
- PKCS8ShroudedKeyBag. A PKCS #8 encrypted private key.
- CertBag. A certificate.
- CRLBag. A Certificate Revocation List.
- SecretBag. A personal secret.
- SafeContents. Allows for nesting the other types instead of just putting them in a sequence.

Note that all of these have their own OID and can also have attributes.

4 Public Key Infrastructure

When using a public key, either for encryption or for verification of a signature, it is important to be certain that the public key belongs to the claimed owner. There is nothing secret in the public key so it can be safely distributed to anyone, put on websites or sent in emails. While this has many advantages compared to symmetric keys, it comes with an important drawback, namely that anyone can create a key pair and publish the public key claiming that it belongs to someone else. This problem is solved using certificates, which is a signed document where the signer (CA) guarantees the binding between the public key and the subject. A public key infrastructure (PKI) includes all aspects of certificates, how they are signed, who signs them, how they are revoked, how they are stored, and how keys are generated. While the concept of certificates is common to all PKIs, different PKI profiles can define specific rules for the infrastructure, e.g., who is allowed to sign a certificate, how they are stored, what type of information is given in the certificate, and how it should be interpreted. The most common certificate format is X.509. PKIX (Public-Key Infrastructure (X.509)) is the name of the IETF working group that specifies PKI services for the Internet. The term PKIX is also used to refer to the profile called Internet PKI profile. In turn, there are also other profiles that are based on the PKIX specifications. Another profile is FPKI (US federal PKI profile). Many countries have their own profile that is used in certain situations. This section is based on some of the PKIX specifications.

This section will start by defining some of the involved parties in a PKI. Then, certificate requests, certificates and revocation of certificates is discussed by detailing some of their content. These structures and messages are specified using ASN.1, and some of the ASN.1 syntax will be given to show both the content of the structures, and how they are built.

4.1 PKI Participants

4.1.1 End Entity

An end-entity is typically a user, i.e., the subject that the certificate is issued to, but it can also be an application. For an end-entity the corresponding private key must not be used to sign other certificates.

4.1.2 CA

A certification authority (CA) is the issuer of a certificate and the entity that signs a certificate. There are different types of CA and the exact meaning of a type can depend on the PKI architecture. In an hierarchical PKI architecture, the root CA is at the top of a tree. The root CA can issue certificates to other CAs. It is then a superior CA issuing a certificate to a subordinate CA. A subordinate CA is thus a CA that is not a root CA. There can also be intermediate CAs, which are CAs that are subordinate CAs themselves and also has its own subordinate CAs.

4.1.3 RA

A registration authority does not sign certificates or CRLs. Instead, it is a system to which a CA has delegated certain management functions, often verification of the identity information that is to be signed by the CA. It can also be in charge of assigning names, generating keys, storing keys and reporting revocation information about a certificate. The RA is an optional component, and in many cases all management functionality is performed by the CA itself.

4.1.4 CRL issuer

A CRL issuer is in charge of issuing certificate revocation lists. The responsibility for CRLs is usually the CA of the issued certificates, but the CA may delegate this to a CRL issuer.

4.2 Certificate Requests

In order to issue a certificate, a subject prepares a certificate request to be signed by a CA. There are two main variants of certificate requests, namely PKCS #10 format and the Certificate Request Message Format (CRMF). This section will give a brief overview of these two message formats.

4.2.1 PKCS #10

The PKCS #10 certificate request format was defined in 2000 by RSA Security Inc. It is described in RFC 2986. The top level content of the request is given by the ASN.1 type CertificationRequest.

```
CertificationRequest ::= SEQUENCE {
  certificationRequestInfo  CertificationRequestInfo,
  signatureAlgorithm        AlgorithmIdentifier,
  signature                 BIT STRING
}
```

It consists of three main parts. The first part is the certificationRequestInfo which is a value that is signed. The last two parts are the signature algorithm and the actual signature. The CertificationRequestInfo includes information about the subject and the public key that will be tied together in the certificate.

```
CertificationRequestInfo ::= SEQUENCE {
  version       INTEGER,
  subject       Name,
  subjectPKInfo SubjectPublicKeyInfo,
  attributes    [0] Attributes
}

SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm         AlgorithmIdentifier,
  subjectPublicKey  BIT STRING
}
```

It additionally contains a set of attributes. This can be used to define certificate attributes but it can also hold e.g., a challenge password which will be used by the requester if he later wishes to revoke the certificate.

By signing the CertificationRequestInfo using the private key, the requester proves that he is in possession of the private key without having to show the private key to the CA. Without this proof of possession anyone could request a certificate for any public key and could then also claim to have produced any signature using the corresponding private key.

PKCS #10 is a widely used format for certificate requests and is supported by many applications.

4.2.2 Certificate Request Message Format (CRMF)

The CRMF format for a certificate request is specified in RFC 4211 [17] and details can be found in that document. It has some similarities with PKCS #10, but differs in some aspects. The most important difference is the added flexibility in the support for proof of possession mechanisms.

The top level content of the request is given by the following ASN.1 type CertReqMsg.

```
CertReqMsg ::= SEQUENCE {
  certReq   CertRequest,
  popo      ProofOfPossession  OPTIONAL,
  regInfo   SEQUENCE SIZE(1..MAX) of AttributeTypeAndValue OPTIONAL
}
```

The CertRequest type is in turn described as follows.

```
CertRequest ::= SEQUENCE {
  certReqId     INTEGER,
  certTemplate  CertTemplate,
  controls      Controls OPTIONAL
}
```

The first field is an ID that can be used to match a reply with this request.

The CertTemplate contains the information that is to be included in the certificate. Much of this information should actually be filled in by the RA or the CA, like issuer name, serial number and signing algorithm, but the public key is an example of a field that is provided by the requestor unless the key is generated by the CA/RA. Other fields include e.g., validity period, subject name and extensions. The optional Controls type can be used to provide the CA/RA with extra information, such as authentication if the subject already has a relationship with the CA/RA.

Following the CertRequest is the Proof of Possession type. Different from PKCS #10, this proof of possession (POP) can be performed in several different ways and the CA/RA can decide which way is most appropriate for the given situation. Depending on the method, the POP is validated by either the CA, the RA or both. The different methods are based on how the key pair will be used, given by the key usage extension in the certificate. If a key has multiple usages, then any method can be used. For a signature key, a signature can be computed on some part of the data. By verifying this signature, the CA/RA knows that the end-entity is in possession of the private key. For a key that will be used to encrypt other keys, one alternative is for the key to be directly sent to the CA/RA, encrypted with a key known to both the end-entity and the CA/RA. Otherwise, the CA/RA can send an encrypted challenge to the end-entity, or the certificate can be encrypted with the public key when it is returned to the end-entity. One important aspect is that, for signature keys, the private key should never be sent to the CA/RA, because this will limit the non-repudiation property of any digital signature produced by that private key.

4.3 X.509 Certificates

X.509 is the certificate format used in PKIX. The first version of the format was X.509 v1 which was specified in 1988. A second version, v2, was specified in 1993 and the current version which is the most common is version 3, specified in 1996. An important addition in version 3 was the use of extensions, which improves the flexibility and the type of information that can be stored, and signed, in the certificate. The format of X.509 v3 certificates is given in RFC 5280 [5] and the ASN.1 modules given in this section are taken from that document. At the top level, the layout of an X.509 certificate is given by the following ASN.1 structure.

```
Certificate ::= SEQUENCE  {
  tbsCertificate       TBSCertificate,
  signatureAlgorithm   AlgorithmIdentifier,
  signatureValue       BIT STRING
}
```

The tbsCertificate (tbs=to be signed) are the values of the certificate that is signed. The signatureAlgorithm specifies which algorithm to use. It consists of an OID and optionally some extra parameters for the algorithm. The signa-

tureValue is the actual signature. The type TBSCertificate is further defined as

```
TBSCertificate ::= SEQUENCE  {
      version         [0]  EXPLICIT Version DEFAULT v1,
      serialNumber         CertificateSerialNumber,
      signature            AlgorithmIdentifier,
      issuer               Name,
      validity             Validity,
      subject              Name,
      subjectPublicKeyInfo SubjectPublicKeyInfo,
      issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                           -- If present, version MUST be v2 or v3
      subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                           -- If present, version MUST be v2 or v3
      extensions      [3]  EXPLICIT Extensions OPTIONAL
                           -- If present, version MUST be v3
}

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension
```

For an exact description of each of the types, refer to RFC 5280. The version gives the version of the certificate and the serialNumber is an integer of size up to 20 bytes. The certificate serial number must be unique among all certificates issued by one issuer. The AlgorithmIdentifier type named signature is the same as the signatureAlgorithm given in the top level. By including it here as well, also the algorithm used is signed. The name type used to identify both the issuer and the subject itself consists of several types. The validity specifies for how long time a certificate is valid and the subjectPublicKeyInfo gives information about the public key that corresponds to the certificate.

As noted before, version 3 added the possibility for extensions, i.e., to add additional information that is useful in specific situations. Extensions are divided into critical and non-critical. An application must not process a certificate if it has a critical extension that it does not recognize, but unrecognized non-critical extensions can be safely ignored. Each extension has its own OID and ASN.1 structure. The ASN.1 structure is DER encoded to an octet string. The format for extensions is shown below.

```
Extension ::= SEQUENCE  {
  extnID      OBJECT IDENTIFIER,
  critical    BOOLEAN DEFAULT FALSE,
  extnValue   OCTET STRING
}
```

Below are some examples of extensions that can be used in X.509 v3 certificates.

- **Authority Key Identifier** (non-critical). This field can be used to identify the public key that should be used to verify the signature of the certificate. It is in particular needed when an issuer has several different signing keys.

- **Subject Key Identifier** (non-critical). This value can be used to identify the public key in the certificate. In a CA certificate, i.e., a certificate with the ability to sign other certificates, this value must correspond to the information in the authority key identifier extension in the signed certificate.

- **Key Usage** (critical). This can be used to specify the purpose of the public key given in the certificate. The purpose could be e.g., dataEncipherment if it is to be used by other parties to encrypt information to the subject, or digitalSignature if the purpose of the public key is to verify a signature created by the subject. Signatures for certificates and CRLs have their own usage type. It is commonly acknowledged that keys should not in general be used for several purposes. A key pair used for signatures should not also be used for encryption. Using different key pairs will provide some damage control for compromised keys. There is also an extension called **Extended key usage**, which gives organizations even more options to specify the key usage.

- **Subject Alternative Name** (critical or non-critical). This can be used to tie an extra subject name to the certificate. It could e.g., be an email address, a DNS name or some other name that makes sense to some local application. The alternative name can be used instead of the subject name, leaving the subject name empty. In that case this extension is critical. Otherwise, it is non-critical.

- **Issuer Alternative Name** (non-critical). Similar to the subject, the issuer can add an alternative name to the certificate.

- **Basic Constraints** (critical or non-critical). This specifies if the certificate is a CA certificate and the number of certificates in the chain containing this certificate. In particular, for user certificates, the Boolean value for CA is set to false so that users can not use their private key to sign other certificates. This extension is critical in CA certificates that are used to verify signatures on other certificates.

- **Authority Information Access** (non-critical). This is used for supplying information about the issuer of the certificate. It has two main purposes. First, it can point to the entity that issued the issuer's certificate. This can help applications to find certificates needed to build the certificate chain. The second purpose is to inform applications of where the OCSP server is located so that online revocation queries can be made (see Section 4.5).

- **CRL distribution point** (non-critical). This can be used to specify where to find CRL information about this certificate. Typically, the issuer is also responsible for CRLs, but this is not always the case. When someone else distributes the CRLs, this extension is used to identify that issuer.

4.3.1 Representing Public Keys

Public keys and signatures can consist of several values. The public RSA key consists of one public exponent, often called e, and a public modulus, called n. A DSA signature consists of two values, usually referred to as r and s. The keys and the signatures need to be represented in a predefined way so there is no ambiguity in their interpretation. RFC 3279 [2] gives ASN.1 structures for some common algorithm parameters, e.g., RSA and DSA keys, and the OIDs that should be used for some common algorithms. More algorithms have subsequently been supported and details on how to represent and encode these can be found in RFC 4055, RFC 4491, RFC 5480, RFC 5758. It is safe to assume that this list will be expanded in the future.

Consider the type SubjectPublicKeyInfo given in the TBSCertificate type. This type is defined as

```
SubjectPublicKeyInfo ::= SEQUENCE  {
  algorithm            AlgorithmIdentifier,
  subjectPublicKey     BIT STRING
}
```

The OID of the algorithm is followed by a bit string representing the public key. For RSA, the same OID is used regardless if the key is used for signatures or for encryption. Recall that usage restrictions can be defined in an extension. An RSA public key has the ASN.1 structure given below.

```
RSAPublicKey ::= SEQUENCE {
  modulus            INTEGER,    -- n
  publicExponent     INTEGER     -- e
}
```

The DER encoding of the RSAPublicKey type is then used as the bit string in the SubjectPublicKeyInfo. How to represent the integers is defined by the ASN.1 standard. The representation of the private key is outside the scope of this document since it is not included in a certificate, but there are ASN.1 structures for that as well, see e.g., RFC 3447 [13]. How the signature is represented is defined by the signature algorithm and is just given as a bit string in the certificate ASN.1 definition. For RSA, RFC 3447 also specifies how to compute the signature and obtain the resulting bit string.

4.4 CRL

As the structure of X.509 certificates, the structure of a CRL is also given in RFC 5280. The most recent version of X.509 CRLs is version 2. The CRL lists certificates that are no longer valid for some reason. Expired certificates are not found in CRLs since it is easy for implementations to check the validity time of a certificate. Instead, reasons to revoke certificates could e.g., be that the private key has been compromised, that the subject should no longer have access to a resource which it had access to through the certificate or that the key pair has been replaced by a new key pair. The CRL issuer is in general a CA. The CA is responsible for maintaining a CRL for certificates it has issued, but it can also delegate this responsibility to another party.

How often a CRL is updated can vary between hours and weeks, depending on the issuer. At the same time, it is important that CRLs are as up-to-date as possible. If a private key is compromised, revocation information of the corresponding certificate must be available to applications immediately in order to avoid attacks based on the compromised key.

The *scope* of a CRL is the category of certificates that are covered by the CRL, e.g., all certificates issued by some particular CA or all certificates with subjects belonging to a particular organization. The reason for revocation can also be included in the scope. A CRL is *complete* if it includes all revoked certificates within its scope. An *indirect* CRL contains revoked certificates that have been issued by an entity other than the CRL issuer. A *delta* CRL is a CRL that only contain certificates that have been revoked after a given complete CRL, then called *base* CRL, has been issued.

The top level ASN.1 structure for a certificate list is given below.

```
CertificateList ::= SEQUENCE  {
  tbsCertList          TBSCertList,
  signatureAlgorithm   AlgorithmIdentifier,
  signatureValue       BIT STRING
}
```

As can be seen, this part is very similar to the structure of a certificate. It includes one type, in this case the TbsCertList, that will be signed, the algorithm used to sign, and the signature. The TBSCertList type has the following definition.

```
TBSCertList ::= SEQUENCE  {
  version                 Version OPTIONAL,
  signature               AlgorithmIdentifier,
  issuer                  Name,
  thisUpdate              Time,
  nextUpdate              Time OPTIONAL,
  revokedCertificates     SEQUENCE OF SEQUENCE  {
       userCertificate         CertificateSerialNumber,
       revocationDate          Time,
       crlEntryExtensions      Extensions OPTIONAL
                          } OPTIONAL,
  crlExtensions      [0]  EXPLICIT Extensions OPTIONAL
}
```

Also here, some similarities to the X.509 certificate can be found. It has a version number, a signature algorithm and an issuer. The type named thisUpdate specifies the time at which the CRL was issued and a new CRL is guaranteed to be issued at latest the time given by nextUpdate. After this, the list of revoked certificates is given. The complete certificates are not included, only the serial number is used to identify the revoked certificate. The date that a particular certificate was revoked is also given, together with zero or more CRL entry extensions. RFC 5280 lists three CRL entry extensions.

- Reason code. This can be used to indicate the reason for the revocation. Reasons include e.g., a compromised private key, a compromised CA, a change of affiliation and that the certificate has been superceded by another certificate.
- Invalidity date. This can be used to indicate an expected time when the certificate became invalid. This time may very well be before the time it was revoked if there e.g., are indications that a private key was compromised some time before the actual compromise was discovered.
- Certificate Issuer. For indirect CRLs, the issuer of the revoked certificate can be given using this extension.

Finally, it is possible to give a set of CRL extensions. Similar to certificate extensions, and with the same meaning, these can be critical or non-critical. The Authority Key Identifier and the Issuer Alternative Name extensions have the same meaning as for certificates. Some extensions that are exclusive for CRLs are:

- CRL Number (non-critical). This is a sequence number used for the CRL.
- Delta CRL Indicator (critical). This specifies that the CRL is a delta CRL. The value used is a reference to the CRL number of the base CRL.

- Freshest CRL (non-critical). This extension can be used for complete CRLs to specify where delta CRLs can be found.

In [16], Rivest discusses problems with CRLs. One problem is that it is the entity that is accepting the certificate that should define how fresh the revocation information should be, since the acceptor takes the risk of using e.g., a compromised key. For CRLs, however, it is the CA that determines how often a CRL should be updated. Moreover, the result provided by the CRL is only negative. The acceptor does not get any information whether the certificate is valid, only that it is invalid.

4.5 OCSP

One drawback with CRLs is that there is always some time between the revocation of a certificate and the inclusion of the certificate into a new CRL (thisUpdate). In some situations, e.g., large money transfers or stock market activity, this time can be very critical. A compromised private key can potentially be used to perform attacks during this time. The Online Certificate Status Protocol (OCSP), specified in RFC 2560 [15], aims to minimize this time by defining a protocol that can be used to query a CA about the revocation status for a particular certificate. In this way, the status information can be very recent. The location (URI) of the OCSP server can be found in the certificate extension Authority Information Access.

The ASN.1 specification of the request and response can be found in RFC 2560, and they are built in a similar way as certificates and CRLs. This section will only discuss the main contents of the messages.

The request primarily consists of a hash of the issuer and the issuer's public key, together with the serial number of the certificate. This information can be used to uniquely identify a certificate. Several certificates can be identified in the same request. The request can also be optionally signed. Similar to certificates and CRLs, there are a number of extensions. One important extension is a nonce that can be used to protect against replay attacks.

The response contains a status result for each of the certificates in the request. There are three possible statuses for a certificate, namely good, revoked and unknown. The "unknown" status means that the responder does not know anything about the certificate that is in the request. It can also return an error code if there is something wrong with the request, e.g., that the responder required that the request is signed.

There are several timestamps in the response. The time given by thisUpdate indicates the last time the status sent was known to be correct and nextUpdate indicates the latest time when new information about the status will be available. The response also includes a timestamp named producedAt, which specifies the time at which the response was signed. It is possible to create signed responses in advance and use them immediately when receiving a request. The response must be digitally signed either by the CA that issued the certificate, some

entity that the CA has delegated OCSP functionality to, or someone else that the requester has decided to trust.

5 Avoiding PKI

There has been a large amount of work trying to find alternatives to PKIs. This section will give a brief overview of some techniques that can allow public key cryptography without having an infrastructure with CAs and CRLs.

5.1 Identity-Based Cryptography

When sending an encrypted message, the sender needs to locate the public key of the receiver. This needs to be done by retrieving a certificate, signed by possibly a chain of CAs, that binds the receiver's identity to his or her public key. Then the certificate chain needs to be verified. Identity based cryptography (IBC) was proposed by Shamir in [18]. The idea is that the identity itself *is* the public key. If Alice wants to send an encrypted message to Bob, she can encrypt it with e.g., the email address of Bob. Then there is no need for a certificate. Only the owner of the email address can decrypt the message. It is implicitly assumed here that we know that the email address actually belongs to Bob, otherwise we would be back in the same situation as before. In general the identity can be anything, e.g., an IP address, the social security number (equivalence of "personnummer"), or something else that identifies the receiver. Identity-based cryptography reduces the complexity of the system and simplifies the use of public key cryptography.

To realize IBC, a trusted third party called Private Key Generator (PKG) is used. The PKG has its own asymmetric key pair, called master public key and master private key. The public key is denoted pk_{PKG} and the private key is denoted sk_{PKG}. Users have their identity ID as the public key and the corresponding private key is created by the PKG, denoted sk_{ID}. Exactly how the PKG verifies the identity of the user so that sk_{ID} is communicated correctly is up to the system, but it can be seen as the same step that is performed by a CA before signing a certificate.

5.1.1 Identity Based Signatures

In an identity-based signature (IBS) scheme, the signer uses its own private key, sk_{ID} to sign a document. In the verification process, both the identity, which is the user's public key, and the public key of the PKG is used. In [18], it was shown how RSA could be used to create an IBS scheme. The following description assumes that the reader is familiar with RSA.

In the first phase, the PKG creates an RSA system by computing the public RSA values n and e, which are seen as the master public key, pk_{PKG}. The master private key is given by the private RSA exponent d. The user's public key is given by the identity $i = ID$ and the corresponding private key, sk_{ID} is

given by g such that

$$g^e = i \bmod n.$$

Note that computing g from e, i and n corresponds to decrypting the value i using RSA. Thus, the private key g is difficult to compute without knowing the factorization of n, which only the PKG knows.

When the user, Alice, wants to sign a message m, she picks a random number r and computes

$$t = r^e \bmod n.$$

The signature is computed by hashing the message together with t, $h(t, m)$ and computing

$$s = g \cdot r^{h(t,m)} \bmod n.$$

The signature consists of the pair (s, t).

When verifying the signature the condition

$$s^e = i \cdot t^{h(t,m)} \bmod n$$

is used since

$$s^e = g^e \cdot r^{eh(t,m)} \bmod n = i \cdot t^{h(t,m)} \bmod n.$$

5.1.2 Identity-based Encryption

While the signature scheme given in the previous section was proposed already in 1984, no construction for an identity-based encryption (IBE) scheme was given. In 2001, two IBE schemes were given independently in [3] and [4]. The scheme in [3] uses a bilinear map on an elliptic curve. The details of this scheme will not be covered here, but it should be noted that these bilinear maps, or bilinear pairings, have been used to construct several variants of IBE schemes and also IBS schemes that are more efficient than the one described above.

5.1.3 Advantages and Drawbacks of IBC

The most important advantage of IBC is that there is no need for a PKI. The recipient of a message does not even have to have a key pair in advance. It is possible to send a message encrypted with e.g., the email address of the recipient. Upon receiving the message, the recipient can query the private key from the PKG.

Another advantage is that there is no need for CRLs. This can also be seen as a drawback since there is no possibility of revoking public keys. If the private key is compromised, it is not practical to revoke an email address. Even if this is possible, consider the situation where the identity is a biometric feature, e.g., a fingerprint or an iris scan. These are not as easy to replace. One solution to this problem is to add a time stamp to the ID so the ID is only valid as long as the timestamp is valid.

While it is convenient to be able to use a receiver's identity to encrypt messages, the constructions have some drawbacks. The most serious drawback

is that the private key for each user is always known to the PKG. This property is called key escrow. This means that the PKG must be fully trusted be everyone. This is similar to a CA in a PKI. Even though the CA does not necessarily have access to the private key, it has the power to create certificates binding any subject to any public key, which creates a similar situation. For IBS schemes, the non-repudiation property is in a sense broken since the PKG has access to the private key. At least, the non-repudiation property will depend on the trust users have in the PKG. For encryption, the fact that the PKG has access to the private key can also be seen as a feature. If the user fully trusts the PKG, he/she can let the PKG decrypt all messages and then send them to the user. Then the user does not have to have any cryptographic functionality implemented. A situation where the PKG is part of the same corporation as the users is a possible usage scenario for this.

5.2 Other Related Constructions

A drawback with CRLs is that the sender of an encrypted message needs to verify that the public key of the receiver has not been revoked. In certificate-based encryption (CBE) schemes, the receiver needs to have, in addition to the private key, an up-to-date certificate in order to decrypt the message. Thus, if the certificate has been revoked, it is not possible to decrypt the message. A CBE scheme is similar to IBC in the sense that there is no need to check the authenticity of a public key before it is used. In addition, the user generates the private key so there is no key escrow problem in CBE. CBE was introduced in [7].

Certificateless public key cryptosystems (CL-PKC), introduced in [1] also solves the problem of key escrow. The idea is that the PKG only creates half the private key and the other half is created by the user. Similar to IBC, this solution does not require a certificate.

References

[1] S. S. Al-Riyami and K. G. Paterson. Certificateless public key cryptography. In C-S Laih, editor, *Advances in Cryptology—ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 452–473. Springer, 2003.

[2] L. Bassham, W. Polk, and R. Housley. Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3279 (Proposed Standard), April 2002. Updated by RFCs 4055, 4491, 5480, 5758, Available at: *http://www.ietf.org/rfc/rfc3279.txt*.

[3] D. Boneh and M. K. Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology—CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.

[4] Clifford Cocks. An identity based encryption scheme based on quadratic residues. In B. Honary, editor, *IMA Int. Conf.*, volume 2260 of *Lecture Notes in Computer Science*, pages 360–363. Springer, 2001.

[5] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Available at: *http://www.ietf.org/rfc/rfc5280.txt*.

[6] O. Dubuisson. *ASN.1 - Communication Between Heterogeneous Systems*. Morgan Kaufmann Publishers, 2000. Available at: *http://www.oss.com/asn1/resources/books-whitepapers-pubs/asn1-books.html*.

[7] C. Gentry. Certificate-based encryption and the certificate revocation problem. volume 2656 of *Lecture Notes in Computer Science*, pages 272–293. Springer, 2003.

[8] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (Standard), September 2009. Available at: *http://www.ietf.org/rfc/rfc5652.txt*.

[9] ITU-T Recommendation X.680. Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation, 2008. Available at: *http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf*.

[10] ITU-T Recommendation X.690. Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), 2008. Available at: *http://www.itu.int/itu-t/recommendations/rec.aspx?rec=X.690*.

[11] ITU-T Recommendation X.691. Information technology ASN.1 encoding rules: Specification of Packed Encoding Rules (PER), 2008. Available at: *http://www.itu.int/itu-t/recommendations/rec.aspx?rec=X.691*.

[12] ITU-T Recommendation X.693. Information technology ASN.1 encoding rules: XML Encoding Rules (XER), 2008. Available at: *http://www.itu.int/itu-t/recommendations/rec.aspx?rec=X.693*.

[13] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003. Available at: *http://www.ietf.org/rfc/rfc3447.txt*.

[14] J. Larmouth. *ASN.1 Complete*. Morgan Kaufmann Publishers, 1999. Available at: *http://www.oss.com/asn1/resources/books-whitepapers-pubs/asn1-books.html*.

[15] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560 (Proposed Standard), June 1999. Updated by RFC 6277, Available at: *http://www.ietf.org/rfc/rfc2560.txt*.

[16] Ronald L. Rivest. Can we eliminate certificate revocations lists? In R. Hirschfeld, editor, *Financial Cryptography*, volume 1465 of *Lecture Notes in Computer Science*, pages 178–183. Springer, 1998.

[17] J. Schaad. Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF). RFC 4211 (Proposed Standard), September 2005. Available at: *http://www.ietf.org/rfc/rfc4211.txt*.

[18] A. Shamir. Identity-based cryptosystems and signature schemes. volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer, 1984.