MÁSTER INTERUNIVERSITARIO EN CIBERSEGURIDAD

Information Security Lab assignment III

1 Problem Description: broadcast encryption with revocation

In this exercise you will implement a simplified form of the AACS (Advanced Access Content System), which is the industry standard adopted for encrypting the contents in DVDs and Blu-ray discs. There have been multiple claims that this system has been broken, but it is instructive to learn how it was constructed and how it operates.

DVDs and Blu-ray discs contain copyrighted material that should be only reproducible by authorized player devices. Suppose there are a total of n devices, and without loss of generality assume that n is a power of two. We view these n devices as the leaves of a complete binary tree with $t = \log_2 n$ levels, plus the root node. Suppose also that every node in the tree is assigned a random key taken from the key space $\mathcal{K}$. The set of keys embedded in device number $i \in \{1, \ldots, n\}$ is the set of keys on the path from leaf number i to the root. Thus, every device is given exactly t keys.

Initially, all contents in the disc is encrypted with the root key, in the following way: for encrypting a movie m in a disc

1. A random key k is created.
2. The key k is encrypted with a secure cipher $c_1 \leftarrow E(k_{\text{root}}, k)$
3. The content is encrypted with a semantically secure cipher as $c \leftarrow E'(k, m)$
4. The output is (c_1, c)

For playing m, the device uses k_{root} to decrypt c_1 and recover k; then, it uses k to decrypt the movie m.

1.1 Why a tree of keys?

Using a tree of keys provides security just in case some of the device has been compromised, i.e., when all the keys contained in it have been published to the world. Upon this event, all the future content is encrypted using the keys from the siblings of the t nodes on the path from leaf i to the root (see the Figure). For example, if device number 3 had been attacked and compromised, then the keys k_3, k_{10}, k_{13} and k_{15} would have been announced. To encrypt new content, the creator would use now the keys k_4, k_9 and k_{14} as in:

1. Create a random key k for the disc.
2. Compute $c_1 \leftarrow E(k_4, k)$, $c_2 \leftarrow E(k_9, k)$, $c_3 \leftarrow E(k_{14}, k)$. That is, encrypt the key k under k_4, k_9 and k_{14}.
3. Encrypt the content $c \leftarrow E'(k, m)$
4. Output (c_1, c_2, c_3, c)

Now, observe that device number 3 **cannot decrypt the content**, because it cannot decrypt any of the headers in the disc and, in turn, decrypt k. Nevertheless, **any other device has one key to decrypt the header**, so it can recover the contents in the disc.

Therefore, arranging the device keys in a tree allows to revoke the key of a single device when it has been compromised, while leaving all the rest of the devices unchanged. The system supports revocation of any subset of devices.

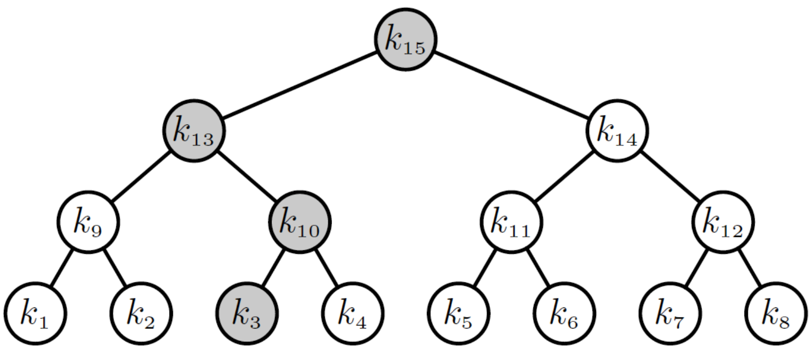

Figure 1: An example tree of keys.

1.2 The general encryption procedure

According to the description above, the general procedure for encrypting copyrighted material under potential revocation is this: assume the set of devices not yet compromised is S

1. Generate a random key k.
2. For every node u in a cover of S, compute $c_u \leftarrow E(k_u, k)$.
3. Encrypt the content as $c \leftarrow E'(k, m)$.
4. Output $(\{c_u\}_{u \in \text{cover}(S)}, c)$.

A *cover of* S is a minimal set of nodes for which every leaf in S is a descendent of some node in the cover, but leaves outside S are not. For example, $\{k_4, k_9, k_{14}\}$ are a cover of $\{k_1, k_2, k_4, k_5, k_6, k_7, k_8\}$.

The only nontrivial part in this procedure —aside from applying the encryption functions— is how to determine the cover of a given subset $S \subseteq \{1, \ldots, n\}$ in the binary tree of keys. To solve this, the easiest way is to relabel all the nodes in the tree starting at the root (node 1), its descendants (nodes 2 and 3, from left to right), the second level nodes (4, 5, 6, 7, from left to right) nd son on. This way, the leaves in the tree are nodes with numbers $2^t, \ldots, 2^{t+1} - 1$. With this labeling, the sibling of node j is

$$\text{sibling}(j) = \begin{cases} j+1 & \text{if } j \text{ is even} \\ j-1 & \text{if } j \text{ is odd.} \end{cases} \tag{1}$$

And the parent of a node has index

$$\text{parent}(j) = \left\lfloor \frac{j}{2} \right\rfloor. \tag{2}$$

A cover set for S is simply the set of siblings of nodes in the paths from the root to any node not in S. Namely, if $u \notin S$ its path from the root is recursively computed as $\mathcal{P}_u = \{u\}$, and appending to $\mathcal{P}_u$ the parent of the last element appended to $\mathcal{P}_u$.

2 Task

Implement this simplified version of AACS. As encryption functions E and E' use AES-128 in counter mode or in CBC mode.[1] Your program must receive two inputs: the message to encode —an image or a short movie would be great examples— and a set $T \subset \{1, \ldots, n\}$ of revoked devices (T could be empty). And the program must produce the encoding of m so that the devices in T are unable to reproduce the content whereas the devices in $\{1, \ldots, n\} \setminus T$ can reproduce it. Notice that it is not strictly necessary that n is a power of two.

[1] You do not need to know at this moment what CBC (Cipher Block Chaining) is. Just invoke AES in CBC mode using your favorite crypto library.

3 Code templates

```python
import os

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import (
    Cipher, algorithms, modes
)

def encrypt(key, plaintext):
    # Generate a random 128-bit IV.
    iv = os.urandom(16)

    # Construct an AES-128-CBC Cipher object with the given key and a
    # randomly generated IV.
    encryptor = Cipher(
        algorithms.AES(key),
        modes.CBC(iv),
        backend=default_backend()
    ).encryptor()

    # Encrypt the plaintext and get the associated ciphertext.
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()

    return (iv, ciphertext)

def decrypt(key, iv, ciphertext):
    # Construct a Cipher object, with the key, iv
    decryptor = Cipher(
        algorithms.AES(key),
        modes.CBC(iv),
        backend=default_backend()
    ).decryptor()

    # Decryption gets us the plaintext.
    return decryptor.update(ciphertext) + decryptor.finalize()
```