Appendix C

SageMath Code for Bitstream Ciphers

The code snippets can be downloaded from the web page https://www.staff.uni-mainz.de/pommeren/Cryptology/Bitstream/ where the are grouped in three modules

- `FSR.sage`
- `bmAlg.sage`
- `Periods.sage`

They are written in pure Python, hence execute also in a Python environment. For use with Sage attach them by the instructions:

```
sage: load_attach_path(path="/PATH_TO/Sage", replace=False)
sage: attach('FSR.sage')
sage: attach('bmAlg.sage')
sage: attach('Periods.sage')
```

Remember that `xor` is in the module `Bitblock.sage` introduced with Part II of these lecture notes.

C.1 Feedback Shift Registers

Sage Example C.1 A general feedback shift register. The Boolean function f must be initialized first, using the module `BoolF.sage`, see Part II

```
def fsr(f,x,n):
  """Generate a feedback shift register sequence.
  Parameters: Boolean function f, start vector x,
  number n of output bits."""
  u = x
  outlist = []
  for i in range (0,n):
    b = f.valueAt(u)     # feedback value
    c = u.pop()        # output rightmost bit
    u.insert(0,b)      # feedback the leftmost bit
                       # and shift register to the right
    outlist.append(c)
  return outlist
```

Note the use of the indices:

- start vector: `[x[0],...,x[l-1]] = [u[l-1],...,u[1],u[0]]`
- feedback value: $u_i = f(u_{i-1}, \ldots, u_{i-l})$
- output sequence: `u[0],u[1],...,u[n-1]`

Due to the prominence of LFSRs we concide them a special class that allows for several parallel instances:

Sage Example C.2 Linear feedback shift register.

```
class LFSR(object):
  """Linear Feedback Shift Register
  Attributes: the length of the register
              a list of bits describing the taps of the register
              the state"""

  __max = 1024                              # max length

  def __init__(self,blist):
    """Initializes a LFSR with a list of taps and the all 0 state."""
    ll = len(blist)
    assert ll <= self.__max, "LFSR_Error: Bitblock too long."
    self.__length = ll
    self.__taplist = blist
    self.__state = [0] * ll

  def __str__(self):
    """Defines a printable string telling the internals of
    the register."""
    outstr = "Length: " + str(self.__length)
    outstr += " | Taps: " + bbl2str(self.__taplist)
    outstr += " | State: " + bbl2str(self.__state)
    return outstr

  def getLength(self):
    """Returns the length of the LFSR."""
    return self.__length

  def setState(self,slist):
    """Sets the state."""
    sl = len(slist)
    assert sl == self.__length, "LFSR_Error: Bitblock has wrong length."
    self.__state = slist

  def nextBits(self,n):
    """Returns the next n bits as a list and updates the state."""
    outlist = []
    a = self.__taplist
    u = self.__state
    for i in range (0,n):
      b = binScPr(a,u)
      c = u.pop()
      u.insert(0,b)
      outlist.append(c)
    self.__state = u
    return outlist
```

C.2 The BM Algorithm

Sage Example C.3 The BM algorithm. It uses `bibScPr` from the module `Bitblock.sage`, see Part II

```
R.<T> = GF(2)[]

def bmAlg(u):
  """Find the shortest linear feedback shift register that generates
  the bit sequence u."""
# Initialization -----------------------------
  lcprof = [0]
  phi = R(1)
  lc = 0                 # linear complexity up to actual index
  r = -1                 # last index
  psi = R(1)             # last feedback polynomial
  nn = len(u)
# End initialization
  for n in range(0,nn):
    b = u[n-lc:n]        # coefficients for feedback
    flist = []           # feedback taps
    for i in range(0,lc):
      if (i < phi.degree()):
        coeff = phi.coefficients(sparse=False)[i+1] # get coefficient of phi at t^(:
      else:
        coeff = 0
      flist.insert(0,coeff)
    d = u[n] - binScPr(flist,b) # discrepancy between predicted bit and true bit
                                #   -- always 0 or 1 in F_2
    if (d == 1):
      eta = phi - T^(n-r) * psi
      if (2*lc <= n):
        m = n+1-lc         # new linear complexity
        t = lc             # linear complexity in last state
        lc = m
        psi = phi
        r = n
      phi = eta
    lcprof.append(lc)
  outlist = [lcprof,phi]
  return outlist
```