THE OLIVETTI & ORACLE RESEARCH LABORATORY

Nothing better than a Python to write a Serpent

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

http://www.cl.cam.ac.uk/~fms27/serpent/

Frank Stajano 1998

Serpent is a new block cipher created by Ross Anderson, Eli Biham and Lars Knudsen as an AES candidate. It was designed to be faster than DES and safer than triple-DES. After the authors wrote their own preliminary C version, an independent control experiment reimplemented the cipher from scratch in Python. This was by far the slowest ever implementation — but, favouring clarity and readability over efficiency, it was also the first one to produce the correct results.

Writing a cipher is an open-loop process: the output must look like random garbage anyway, so how do you know when it's right? (Optimisation comes later!)
Python lets you express an algorithm at a suitably high level of abstraction, essentially isomorphic to the equations of the original paper, without low-level machine-oriented distractions.
Lesson learned: "ideas in executable form" make a good complement to the formal specification of an algorithm.

The AES challenge

The widely used DES (Data Encryption Standard) block cipher, fielded as a US standard in 1977 and possibly the best known symmetric-key encryption algorithm, is now reaching retirement age: its 56 bit key is too short by modern standards, since a parallel keysearch machine that can exhaust the keyspace in a few hours can be built for about a million dollars.
Hence the American NIST (National Institute for Standards and Technology) issued a call for algorithms for the definition of AES, the Advanced Encryption Standard. The successor to DES shall have a block size of 128 bits and shall accept key sizes between 128 and 256 bits.

Serpent in Python

The Python reference implementation represents bit blocks simply as little-endian strings of the ASCII characters "0" and "1". This scheme yields a "virtual CPU" with arbitrarily long words and it becomes a simple matter to perform operations such as XOR, rotate, extract a particular bit and so on, as well as assigning a block or a key to a variable or to the return value of a function, and inspecting it (even interactively) just by printing it out.

Formal description of the cipher

$$\hat{B}_0 := IP(P)$$
$$\hat{B}_{i+1} := R_i(\hat{B}_i)$$
$$C := FP(\hat{B}_{32})$$

```
def encrypt(plainText, userKey):
    """Encrypt the 128-bit bitstring 'plainText'
    with the 256-bit bitstring 'userKey'
    and return a 128-bit ciphertext bitstring."""

    K, KHat = makeSubkeys(userKey)
    BHat = IP(plainText) # BHat_0 at this stage
    for i in range(r):
        BHat = R(i, BHat, KHat) # BHat_i -> BHat_i+1
    # BHat is now _32 i.e. _r
    C = FP(BHat)
    return C
```

Round function

$$R_i(X) = L(\hat{S}_i(X \oplus \hat{K}_i)) \qquad i = 0, \ldots, 30$$
$$R_i(X) = \hat{S}_i(X \oplus \hat{K}_i) \oplus \hat{K}_{32} \qquad i = 31$$

```
def R(i, BHati, KHat):
    """Apply round 'i' to the 128-bit bitstring
    'BHati', returning another 128-bit bitstring
    (conceptually BHatiPlus1). Do this using the
    appropriately numbered subkey(s) from the
    'KHat' list of 33 128-bit bitstrings."""

    assert 0 <= i <= r-1 # Valid round number
    xored = xor(BHati, KHat[i])
    SHati = SHat(i, xored)
    if 0 <= i <= r-2:
        BHatiPlus1 = LT(SHati)
    else:
        assert i == r-1:
        BHatiPlus1 = xor(SHati, KHat[r])
    return BHatiPlus1
```

Functions used in the round

```
def S(box, input):
    """Apply S-box number 'box' to 4-bit bitstring 'input' and return a
    4-bit bitstring as the result."""

    return SBoxBitstring[box%8][input]
    # There used to be 32 different S-boxes in serpent-0. Now there are
    # only 8, each of which is used 4 times (Sboxes 8, 16, 24 are all
    # identical to Sbox 0, etc). Hence the %8.

def SHat(box, input):
    """Apply a parallel array of 32 copies of S-box number 'box' to the
    128-bit bitstring 'input' and return a 128-bit bitstring as the
    result."""

    result = ""
    for i in range(32):
        result = result + S(box, input[4*i:4*(i+1)])
    return result

def LT(input):
    """Apply the table-based version of the linear transformation to the
    128-bit string 'input' and return a 128-bit string as the result."""

    assert len(input) == 128:
    result = ""
    for i in range(len(LTTable)):
        outputBit = "0"
        for j in LTTable[i]:
            outputBit = xor(outputBit, input[j])
        result = result + outputBit
    return result
```

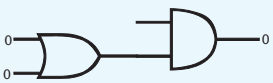
S for the bitslice variant

```
def SBitslice(box, words):
    """Take 'words', a list of 4 32-bit bitstrings, least significant word
    first. Return a similar list of 4 32-bit bitstrings obtained as
    follows. For each bit position from 0 to 31, apply S-box number 'box'
    to the 4 input bits coming from the current position in each of the
    items in 'words'; and put the 4 output bits in the corresponding
    positions in the output words."""

    result = ["", "", "", ""]
    for i in range(32): # ideally in parallel
        quad = S(box, words[0][i] + words[1][i] + words[2][i] + words[3][i])
        for j in range(4):
            result[j] = result[j] + quad[j]
    return result
```

The bitslice technique

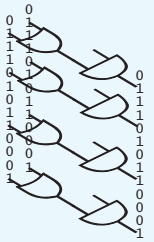
Premature optimisation is bad, but it would be foolish to dismiss efficiency altogether. Once the program is correct, it is often important for it to be fast, especially for a cipher.
However the best gains are achieved not via a posteriori optimisations but by designing the efficiency in the algorithm. Serpent shines in this department, with its patented bitslice technique.



The S-boxes in DES were originally designed for convenient implementation on 1970s hardware. Take bit 4, OR it with bit 2, AND it with bit 5 and so on. When implemented on a modern CPU, these bit operations are cumbersome and wasteful: you have to mask out the relevant bit, shift it into position and then perform a full-width ALU OR to yield just one bit of result.

Serpent was designed from the start around bitslice operation: the data is cleverly arranged so that a 32 bit wide OR operation now performs 32 independent bitwise ORs in one go.

Note that the code shown above is actually just a "simulation" of bitslice operation: it produces the same intermediate results (useful for debugging a real bitslice implementation), but it does not actually obtain them using the parallel bitslice technique. To do so, one must rewrite the S-boxes in terms of boolean operations instead of as a lookup table.



In Serpent's bitslice operation, the bits in the block are "stacked vertically" to a height equal to the width of the ALU. And the move to 64 bit processors will gain us another factor of 2.

It is difficult to meaningfully compare the relative speeds of the AES candidates because the different teams have chosen different trade-offs between security and speed. Serpent has a very conservative design and uses twice as many rounds as would be needed to guard against all the known attacks; hence it is only marginally faster than DES, while others are substantially faster. However its highly optimised bitslice design is indeed very efficient: when the number of rounds of all the AES candidates is normalised to a common strength, Serpent tops the charts for speed.