Lightweight Cryptography in RFID technology

Quin Diack

E-Security: Coursework

Matriculation #: 40429226

5/4/2020

Introduction

As the years go by, humans constantly are trying to reinvent the tools we use every day to be more efficient and versatile for the everyday person. Of those many advances, physical metal keys have evolved and we are starting use chipset technology such as radio frequency identification or RFID. These pieces of technology consist of small radio transponders and receivers that push and confirm digital key information. As these devices become more optimized and widely used, security behind them has become a large concern. With the small amount of computing power held inside a RFID chip, it is clear that running high power and complex forms of cryptography is not an option. Yet without encryption to some degree users run the risk of their data, physical access, or authorization being compromised. Through lightweight cryptography using symmetric encryption methods such as AES and data stream ciphers, this report will explain how less power doesn't necessarily mean less secure when comparing IoT devices.

RFID tags are typically comprised of two main components, first is an antenna for sending and receiving of the radio signals. While second, there is a modulator/demodulator for the processing of the signal and other small tasks. There are three forms of RFID tags used currently today, passive, active, and semi-passive. The most common are tags without batteries known as passive tags, these are charged by the induction of the sensor that it is scanned against. The more expensive option are RFID tags contain batteries for computing without an additional source of power and used for transmitting other data wirelessly. These devices are used by more secure organizations such as the NSA or FBI for tracking inventory products. While the third option is a hybrid of these two, containing a battery that is only used for the computation on the device internally. (Knospe 2004)

As these chips are designed with a very small form factor, therefore, they only can hold a certain amount of data. This amount of data is measured in what are called GE's or gate equivalents, where the

typical passive tag currently holds about 10,000 GE's. With this limited space, dividing the GE's into a secure but sustainable design is one of the first main approaches that cryptographers and engineers have to agree upon. Currently the adopted structure uses about 20% of the gate equivalents for storing the security and crypto algorithms while the other 80% is used to process radio signals and store memory.

More specifically, when analyzing the demand of running cryptography algorithms on the system, developers tested the two common asymmetric encryption algorithms AES (advanced encryption standard) and DES (Data Encryption Standard). It was found that DES used about 2,310 GE's while encrypting the plaintext in 144 clock cycles, meanwhile AES used an encryption only architecture that spent 3,100 GE's. This report will focus on lightweight AES or LED, and a third option known as the rabbit stream cipher. (Maimut, 2012)

Design approach and specifications

The structure of the LED algorithm that will be used similarly relates to the structure of how original AES encryption is written. Just as in AES the design features Sboxes, shifting rows, and mixing columns. This lightweight algorithm contains a 64-bit block cipher and takes in two instances of 64bit and 128 bit keys. At the start the cipher uses an Sbox that is commonly used in a variety of lightweight cryptographic algorithms to change the original values based off a hexadecimal pairing.(fig.1) (Guo,2011)

x	0	1	2	3	4	5	6	7	9	A	B	C	D	E	F
S[x]	6	C	8	0	E	A	7	2	D	1	A	3	4	F	9

(Fig. 1)

The next step in the algorithm is used to mix the columns after the Sbox into a max distance separable matrix. It is calculated by exponential multiplying the value of columns A to the forth power,

establishing four identical rounds of rotation without the addition of other key input values. The ending value of M will be equal to four applications of the input value of A.(fig.2)

$$(A)^4 = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 4\,1\,2\,2 \end{pmatrix}^4 = \begin{pmatrix} 4\,1\,2\,2 \\ 8\,6\,5\,6 \\ \mathrm{B}\,\mathrm{E}\,\mathrm{A}\,9 \\ 2\,2\,\mathrm{F}\,\mathrm{B} \end{pmatrix} = M.$$

(fig.2)

For the 64 bit key in plain text we break into 16 four bit parts and arrange them into a four by four array. These will be identified by using 16 'm' values and loaded in a row based fashion, unlike the column based function we see in common (not-lightweight) AES encryption. The key is then viewed in a similar four by four context where we compute an S of K value that will be alternatively equal for both the 64 bit key and the 128 bit key. This is done using the equation ski j = k(j+i∗16 mod l) for each of the values inside the box as shown below. (fig.3) (Guo, 2011)

$$\begin{bmatrix} sk_0^i & sk_1^i & sk_2^i & sk_3^i \\ sk_4^i & sk_5^i & sk_6^i & sk_7^i \\ sk_8^i & sk_9^i & sk_{10}^i & sk_{11}^i \\ sk_{12}^i & sk_{13}^i & sk_{14}^i & sk_{15}^i \end{bmatrix} \qquad \begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix} \quad \text{all subkeys for the 64-bit key case}$$

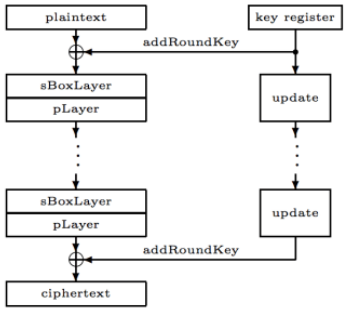
$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix} \begin{bmatrix} k_{16} & k_{17} & k_{18} & k_{19} \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \end{bmatrix} \quad \text{the two first subkeys for the 80-bit key case}$$

$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix} \begin{bmatrix} k_{16} & k_{17} & k_{18} & k_{19} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{24} & k_{25} & k_{26} & k_{27} \\ k_{28} & k_{29} & k_{30} & k_{31} \end{bmatrix} \quad \text{all alternating subkeys for the 128-bit key case}$$

(fig.3)

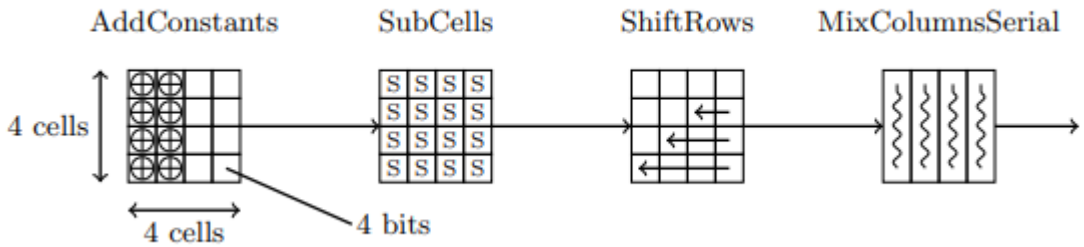
During the next step we are going to add a round key through a series of XOR's with the current state and the subkey or SK^i. The number of rounds will depend of the length of the key size such as 64bit or 128 bit keys. During the 64bit key we will be setting the S value to 8, whereas during the 128 bit key we will be setting the S value to 12. (fig.4)

```
generateRoundKeys()
for i = 1 to 31 do
    addRoundKey(STATE, K_i)
    sBoxLayer(STATE)
    pLayer(STATE)
end for
addRoundKey(STATE, K_32)
```



(fig.4)

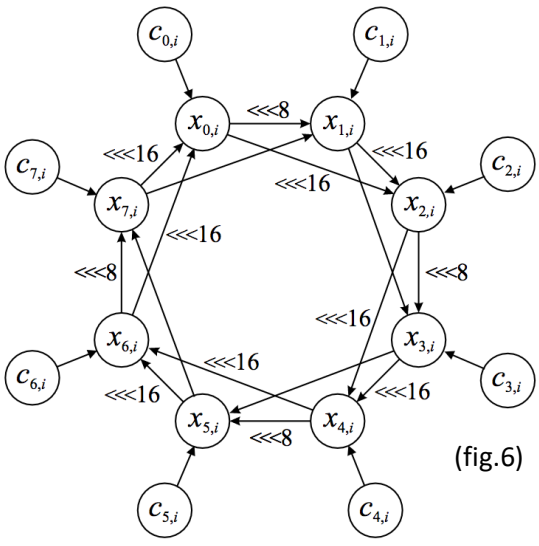
The STATE step shown in the diagram above will be comprised of four rounds, these will be adding constants, substitute cells, shift rows, and mix columns serial. For the add constants round, six bits are initialized to 0 and will be placed in the possition to the left side of the table while XORed with the previous value in the cell. The substitute cells round will replace the nibble with the generated nibble from using the Sbox. Shift rows will simply shift the rows cell position to the left. For instance, 0,1,2,3 becomes 3,0,1,2. The final round to the state value will be a column mix, done by post multiplying the vector and the M matrix. The full state round is demonstrated in the diagram shown below. (fig.5) (Guo, 2011)



The rabbit stream cipher

By using a block cipher like the LED model we just discussed, Each time we add a round to the algorithm we are delaying response time. The previous suggested model is efficient as it uses only four rounds apposed to real AES encryption, however using a stream cipher can provide us with a much faster response and real time encryption/decryption. To test the speed comparison of block ciphers and stream ciphers on low power IoT devices, we will be reviewing the rabbit stream cipher and graphing their relational speed.

The rabbit stream cipher uses a key stream it creates from a 128-bit key and a 64-bit IV. The IV or initial vector, is used to make sure that ciphertext will be different when sending the same message over again. The encryption of the stream uses 8 states and 8 counters where the key is loaded into c_i and x_i registers as seen in the figure below (fig.6) Once loaded the Initial vector is XOR's with the counters state register.



(fig.6)

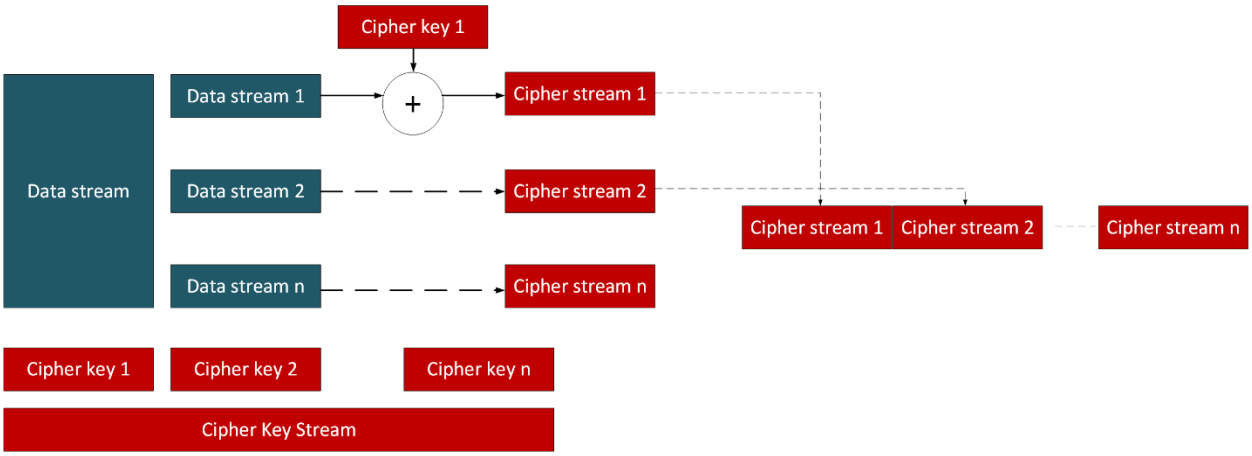
Counters are defined as: c0- c7

States are defined as: x0-x7

State number is defined by: i

After the IV, the algorithm rotates the the state value to the left. After the rotation, the states (x) are separated into eight state values. This creates a 128bit key stream block for each repetition of i. Finally, the data stream will be XOR'd with the 128bit key stream that was just created.

The rabbit stream cipher also will not add any padding or extra blank data when sending the message. This is shown as the size of the input in bytes will be the same as the ciphertext's bytes, thus providing smooth and quick processing. (Buchanan, 2018)



Cryptographic Prototype

To compare the speeds that each of these programs run on a RFID chip will be difficult inside our test environment, as we don't have a sensor or RFID cards to mess around with. However, we can still compare their speed by comparing processing speed on a computer. We discovered source code for the rabbit stream cipher and the LED block cipher and altered it fit our needs for testing. The source code will be available for these programs at the end of the report (Page 14-23).

Through testing the block cipher, we can demonstrate that both the C and D values shown below are equal. This proves that this is a successful cipher and encryption and decryption both work fine. The time we received for executing on our test computer will be faster than an RFID card due to processing power, but the speed relationship will be accurate. (fig. 7)

```
kali@kali:~$ python lighweighttest.py
Test code showing RFID and Reader keys match
('RFID private key: ', 'efbb30710f6958eaa456b9069c0d6db92ce76ae1dc8ae17ae6ed406f7f223692')
('Reader private key: ', '5b15cf3c1b922ec1593bd994294ea7655c18880208ae082f602aec02940c7133')
()
('value of A ', '8d7ac98cf67bc510b3ea9886afbfb0489be3ac9ed77f6429b18d8b3d1ea4ff70')
('value of B ', '8068dc8527bccb3d7df95682a8c4d3ecaf648a42be0b5936f4fc5da137d74b65')
('value of C ', 'c625d4ef2e83c6819fb13587652649b37dacf5dbf5c8c9e1a4ce63e623b92c26')
('value of D ', 'c625d4ef2e83c6819fb13587652649b37dacf5dbf5c8c9e1a4ce63e623b92c26')
()
Check that C is equal to D
```

(fig. 7)

When executing the python code with the time command we receive three sets of time for each run. We receive, real time, user time, and sys time which are used to describe three different time measurements. Real time is the time it takes for the whole process to elapse from the beginning to the end of the call. Meanwhile User and Sys time, is associated with the amount of time the CPU uses inside (sys) or outside (user) the kernel. For our analysis we will be referring to the real time attribute, which tells us that the block cipher takes about .027 seconds to execute. As mentioned before we can expect that this will be faster than when executed on a RFID sensor, due to the extra processing power on our test machine.

When evaluating the rabbit stream cipher, we expect to have faster results. This is mainly because there is no need for the multiple rounds of the blocks that the LED cipher is using. Instead with a stream cipher we have a more direct encryption/decryption path despite a longer code to execute. As mentioned earlier, the size of the ciphertext will be equal to the bytes of the plaintext. This can be seen below as we execute the rabbit stream cipher. (fig. 8)

```
Message:         Hello
IV:  0
Encryption password:     qwerty
Encryption key:      d8578edf8458ce06fbc5bb76a58c5ca4

======Rabbit encryption========
Encrypted:   b'184d78c3a6c2aa'
Decrypted:   Hello
```

After executing we confirmed our hypothesis by seeing a .018 second real time execution. This shows us that rabbit streamline cipher is a faster option for lightweight cryptography of RFID cards. As speed is one of the top difficulties faced with lightweight encryption, we can consider this a big win for the rabbit cipher.

Security Analysis

Although we now know that rabbit cipher is faster than the LED block cipher, before adapting it to our devices it is important to analyze the security infrastructure behind it. In security, there is a constant tradeoff between speed or efficiency and security. First, we are going to discuss our finding on the security that the block cipher provides us with. As this cipher is very closely related to AES a lot of the research on its security has already been completed for us. For 64bit keys LED will XOR the state every four rounds, with the total of 8 steps leaving us with 32 rounds. With 128-bit keys we split the key into two equal parts of k1 and k2. Again, each have four rounds, but we apply 12 steps for the two keys leaving us with 48 rounds. For the single key model, we can easily produce proofs of the minimal number of Sboxes currently active for the permutation with the equation: $(d + 1)^{2.}$. We can use the same equation for the related key model and arrive at 25 active Sbox's for each of them. That being said, single key will have 25 Sboxs for 4 rounds apposed to related key model which will have 8 rounds. (fig.9)

	LED-64 SK	LED-64 RK	LED-128 SK	LED-128 RK
minimal no. of active Sboxes	200	100	300	150
differential path probability	2^{-400}	2^{-200}	2^{-600}	2^{-300}
linear approx. probability	2^{-400}	2^{-200}	2^{-600}	2^{-300}

(Fig. 9)

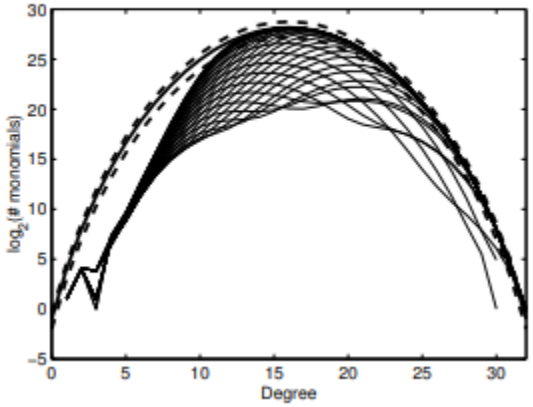
No key is used for the 4 rounds during the permutation which limits the amount of freedom an attacker has for manipulation. A few other cryptoanalysis techniques we could use rational cryptoanalysis, algebraic attacks, and slide attacks. With rotation analysis, LED removes any rotation property by using the Sbox layer. For algebraic attacks, using a 64bit key will consist of 10752 quadratic equations during the permutation for defense. Lastly slide attacks will be prevented by making all rounds different due to the constants added as a round dependent. (Bonnetain, 2019)

Overall the LED block cipher is secure against the classic linear or differential cryptanalysis strategies for both single key (64-bit) and related key (128-bit) models. As this encryption format appears effective and secure, it is still a viable lightweight cryptographic algorithm. Despite being slower than the rabbit stream cipher, its speeds are still fast enough to use in current RFID chips. That being said, evaluation of the rabbit stream cipher could show security vulnerabilities.

When we investigate attacks that could be made on the functionality of key setup of rabbit, we find that guessing part of the key would be very difficult. This is because the cipher relies on the counter and state bits having a strong non-linear relation. Also proving even if an attacker were to guess the counter bits, it would be very hard to figure out the state key.

The guess and verify strategy might not be effective on the rabbit cipher, however there are other crypto cracking attacks that we can evaluate this cipher through. Algebraic attacks are well-known vulnerabilities for stream ciphers that have structured their state to update mostly using linear functions. However, the rabbit cipher uses a completely non-linear structure, which strengthens it against algebraic cracking. When looking at the complexity and relation to random of the g-function, we encounter some inconsistencies. The ideal number of monomials for a random algebraic function is 2^{31}, unfortunately the rabbit stream cipher doesn't always output a monomial this precise and high. Rabbit is known to output monomials in the range between $2^{24.5} – 2^{30.9}$, which is noticeably distinguishable

from random. The distribution that was received through this function over the 32 Boolean functions was calculated by the CRYPTICO organization, and shown graphed below (fig.10) (Boesgaard)



In the graph to the right we can see the average monomials that a random function will have shown as the thick line and the dotted lines around it. The Rabbit cipher produces the results, displayed as narrow lines, that are clearly distinguishable from random.

Although the ciphers g-function are not random, the level of security is still strong due to how complex the algebraic normal form outputs bits. With this cipher we still have 2^{24+} monomials for output each bit, and a complex algebraic degree of more than 30. (Boesgaard)

In the end, all types of RFID chips will become more advanced as time goes on. We can assume that they will eventually be capable of higher-level encryption and data/memory storage. However, after my analysis of these two currently used lightweight ciphers I believe their level of randomness built into their encryption algorithms will be suitable for a long time. That being said the, LED cipher had some advantages when it comes physical space in GE's on the chip. This can be useful for when a RFID chip needs more memory. For instance, if the RFID card was used for a master hotel key and needs to store a lot of keys, more space might be required. The block-based design also makes it a more randomized encryption cipher that its stream-based competitor.

As we discussed earlier, the block design is a slower method than the rabbit stream cipher. Although the algorithm used in rabbit does achieve true randomness of $2^{31,}$ it is secure enough to withstand multiple types of attacks and decryption. The ciphers size is larger than the block ciphers,

however where it loses in size it makes up for in speed. The stream cipher itself is quicker without completing all the rounds in a block cipher, but additionally sends the same size ciphertext as the plaintext in bytes. The data stream also split the cipher streams up into packets that can be intercepted and read by the sensor without waiting for the whole stream to arrive, in real time. Which minimizes the time the sensor takes to receive the message greatly. By using a 128-bit key, it defends against brute force attacks and it currently does not have any known weaknesses.

After my research and comparison of these two commonly used lightweight cryptographic ciphers, a strong recommendation can be made to use the rabbit stream cipher. Its form factor is the perfect size for RFID chips, despite being slightly larger code than its competitors. It makes up for this in response time when being tested and used, which is one of the top goals of along with security. The LED block cipher is still a good choice for many devices and could suit certain users better. In the end our research has led us to believe that RFID cards and other lightweight IoT devices can be the future of how we use keys, tracking devices, and many more items. After learning about their security mechanisms, I feel safer about it carrying information such as bank details. Overall, after understanding this literature, I expect to see RFID used more in the world as time goes on.

References

Bill Buchanan. “As Fast As A Rabbit - Light-Weight Crypto For IoT.” Medium, ASecuritySite: When Bob Met Alice, 28 Aug. 2018, medium.com/asecuritysite-when-bob-met-alice/as-fast-as-a-rabbit-light-weight-crypto-for-iot-d072ae3c31da.

Boesgaard, Martin, et al. The Stream Cipher Rabbit. Cryptico.

Bonnetain, Xavier, et al. “Directory of Open Access Journals.” IACR Transactions on Symmetric Cryptology, Ruhr-Universität Bochum, 1 June 2019, doaj.org/article/55fa9875d1bc4ab989a1fa43dfd55906.

Guo, Jian & Peyrin, Thomas & Poschmann, Axel & Robshaw, Matthew. (2011). The LED Block Cipher.. 326-341.

Light-Weight Crypto: Rabbit, asecuritysite.com/encryption/rabbit2.

Knospe, H., & Pohl, H. (2004). RFID security. Information Security Technical Report, 9(4), 39-50.

Maimut, D., & Ouafi, K. (2012). Lightweight Cryptography for RFID Tags. IEEE Security & Privacy, 10(2), 76-79.

Source code:

Rabbit stream cipher

```
#!/usr/bin/python
# coding=utf-8
# --------------------------------------------------------------------------
#
#   R A B B I T   Stream Cipher
#   by M. Boesgaard, M. Vesterager, E. Zenner (specified in RFC 4503)
#
#
#   Pure Python Implementation by Toni Mattis
#
# --------------------------------------------------------------------------
WORDSIZE = 0x100000000
def rot08(x):
   return ((x << 8) & 0xFFFFFFFF) | (x >> 24)
def rot16(x):
   return ((x << 16) & 0xFFFFFFFF) | (x >> 16)
def _nsf(u, v):
   '''Internal non-linear state transition'''
   s = (u + v) % WORDSIZE
   s = s * s
   return (s ^ (s >> 32)) % WORDSIZE
class Rabbit:
   def __init__(self, key, iv = None):
      if isinstance(key, str):
         if len(key) < 16:
            key = '\x00' * (16 - len(key)) + key
         # if len(key) > 16 bytes only the first 16 will be considered
         k = [ord(key[i + 1]) | (ord(key[i]) << 8)
             for i in range(14, -1, -2)]
      else:
         # k[0] = least significant 16 bits
         # k[7] = most significant 16 bits
         k = [(key >> i) & 0xFFFF for i in range(0, 128, 16)]
```

```
    # State and counter initialization
    x = [(k[(j + 5) % 8] << 16) | k[(j + 4) % 8] if j & 1 else
        (k[(j + 1) % 8] << 16) | k[j] for j in range(8)]
    c = [(k[j] << 16) | k[(j + 1) % 8] if j & 1 else
        (k[(j + 4) % 8] << 16) | k[(j + 5) % 8] for j in range(8)]

    self.x = x
    self.c = c
    self.b = 0
    self._buf = 0          # output buffer
    self._buf_bytes = 0     # fill level of buffer
    next(self)
    next(self)
    next(self)
    next(self)
    for j in range(8):
      c[j] ^= x[(j + 4) % 8]

    self.start_x = self.x[:]    # backup initial key for IV/reset
    self.start_c = self.c[:]
    self.start_b = self.b

    if iv != None:
      self.set_iv(iv)

def reset(self, iv = None):
    self.c = self.start_c[:]
    self.x = self.start_x[:]
    self.b = self.start_b
    self._buf = 0
    self._buf_bytes = 0
    if iv != None:
      self.set_iv(iv)
def set_iv(self, iv):
    '''Set a new IV (64 bit integer / bytestring).'''
```

```
    if isinstance(iv, str):
        i = 0
        for c in iv:
            i = (i << 8) | ord(c)
        iv = i
    c = self.c
    i0 = iv & 0xFFFFFFFF
    i2 = iv >> 32
    i1 = ((i0 >> 16) | (i2 & 0xFFFF0000)) % WORDSIZE
    i3 = ((i2 << 16) | (i0 & 0x0000FFFF)) % WORDSIZE
    c[0] ^= i0
    c[1] ^= i1
    c[2] ^= i2
    c[3] ^= i3
    c[4] ^= i0
    c[5] ^= i1
    c[6] ^= i2
    c[7] ^= i3

    next(self)
    next(self)
    next(self)
    next(self

def __next__(self):
    '''Proceed to the next internal state'''

    c = self.c
    x = self.x
    b = self.b

    t = c[0] + 0x4D34D34D + b
    c[0] = t % WORDSIZE
    t = c[1] + 0xD34D34D3 + t // WORDSIZE
    c[1] = t % WORDSIZE
```

```
    t = c[2] + 0x34D34D34 + t // WORDSIZE
    c[2] = t % WORDSIZE
    t = c[3] + 0x4D34D34D + t // WORDSIZE
    c[3] = t % WORDSIZE
    t = c[4] + 0xD34D34D3 + t // WORDSIZE
    c[4] = t % WORDSIZE
    t = c[5] + 0x34D34D34 + t // WORDSIZE
    c[5] = t % WORDSIZE
    t = c[6] + 0x4D34D34D + t // WORDSIZE
    c[6] = t % WORDSIZE
    t = c[7] + 0xD34D34D3 + t // WORDSIZE
    c[7] = t % WORDSIZE
    b = t // WORDSIZE

    g = [_nsf(x[j], c[j]) for j in range(8)]

    x[0] = (g[0] + rot16(g[7]) + rot16(g[6])) % WORDSIZE
    x[1] = (g[1] + rot08(g[0]) + g[7]) % WORDSIZE
    x[2] = (g[2] + rot16(g[1]) + rot16(g[0])) % WORDSIZE
    x[3] = (g[3] + rot08(g[2]) + g[1]) % WORDSIZE
    x[4] = (g[4] + rot16(g[3]) + rot16(g[2])) % WORDSIZE
    x[5] = (g[5] + rot08(g[4]) + g[3]) % WORDSIZE
    x[6] = (g[6] + rot16(g[5]) + rot16(g[4])) % WORDSIZE
    x[7] = (g[7] + rot08(g[6]) + g[5]) % WORDSIZE

    self.b = b
    return self


def derive(self):
    '''Derive a 128 bit integer from the internal state'''

    x = self.x
    return ((x[0] & 0xFFFF) ^ (x[5] >> 16)) | \
           (((x[0] >> 16) ^ (x[3] & 0xFFFF)) << 16)| \
           (((x[2] & 0xFFFF) ^ (x[7] >> 16)) << 32)| \
```

```
        (((x[2] >> 16) ^ (x[5] & 0xFFFF)) << 48)| \
        (((x[4] & 0xFFFF) ^ (x[1] >> 16)) << 64)| \
        (((x[4] >> 16) ^ (x[7] & 0xFFFF)) << 80)| \
        (((x[6] & 0xFFFF) ^ (x[3] >> 16)) << 96)| \
        (((x[6] >> 16) ^ (x[1] & 0xFFFF)) << 112)


def keystream(self, n):
   '''Generate a keystream of n bytes'''

   res = ""
   b = self._buf
   j = self._buf_bytes
   next = self.__next__
   derive = self.derive

   for i in range(n):
      if not j:
         j = 16
         next()
         b = derive()
      res += chr(b & 0xFF)
      j -= 1
      b >>= 1
   self._buf = b
   self._buf_bytes = j
   return res
def encrypt(self, data):
   '''Encrypt/Decrypt data of arbitrary length.'''

   res = ""
   b = self._buf
   j = self._buf_bytes
   next = self.__next__
   derive = self.derive
```

```
        for c in data:
            if not j:   # empty buffer => fetch next 128 bits
                j = 16
                next()
                b = derive()
            res += chr(ord(c) ^ (b & 0xFF))
            j -= 1
            b >>= 1
        self._buf = b
        self._buf_bytes = j
        return res

    decrypt = encrypt


if __name__ == '__main__':

    import time

    # --- Official Test Vectors ---

    # RFC 4503 Appendix A.1 - Testing without IV Setup

    r = Rabbit(0)
    assert r.next().derive() == 0xB15754F036A5D6ECF56B45261C4AF702
    assert r.next().derive() == 0x88E8D815C59C0C397B696C4789C68AA7
    assert r.next().derive() == 0xF416A1C3700CD451DA68D1881673D696

    r = Rabbit(0x912813292E3D36FE3BFC62F1DC51C3AC)
    assert r.next().derive() == 0x3D2DF3C83EF627A1E97FC38487E2519C
    assert r.next().derive() == 0xF576CD61F4405B8896BF53AA8554FC19
    assert r.next().derive() == 0xE5547473FBDB43508AE53B20204D4C5E

    r = Rabbit(0x8395741587E0C733E9E9AB01C09B0043)
    assert r.next().derive() == 0x0CB10DCDA041CDAC32EB5CFD02D0609B
    assert r.next().derive() == 0x95FC9FCA0F17015A7B7092114CFF3EAD
```

```
assert r.next().derive() == 0x9649E5DE8BFC7F3F924147AD3A947428

# RFC 4503 Appendix A.2 - Testing with IV Setup

r = Rabbit(0, 0)
assert r.next().derive() == 0xC6A7275EF85495D87CCD5D376705B7ED
assert r.next().derive() == 0x5F29A6AC04F5EFD47B8F293270DC4A8D
assert r.next().derive() == 0x2ADE822B29DE6C1EE52BDB8A47BF8F66

r = Rabbit(0, 0xC373F575C1267E59)
assert r.next().derive() == 0x1FCD4EB9580012E2E0DCCC9222017D6D
assert r.next().derive() == 0xA75F4E10D12125017B2499FFED936F2E
assert r.next().derive() == 0xEBC112C393E738392356BDD012029BA7

r = Rabbit(0, 0xA6EB561AD2F41727)
assert r.next().derive() == 0x445AD8C805858DBF70B6AF23A151104D
assert r.next().derive() == 0x96C8F27947F42C5BAEAE67C6ACC35B03
assert r.next().derive() == 0x9FCBFC895FA71C17313DF034F01551CB

# --- Performance Tests ---

def test_gen(n=1048576):
    '''Measure time for generating n bytes => (total, bytes per second)'''

    r = Rabbit(0)
    t = time.time()
    r.keystream(n)
    t = time.time() - t
    return t, n / t

def test_enc(n=1048576):
    '''Measure time for encrypting n bytes => (total, bytes per second)'''

    r = Rabbit(0)
    x = 'x' * n
    t = time.time()
```

```
        r.encrypt(x)
        t = time.time() - t
        return t, n / t

    print(test_gen())
    print(test_enc())
```

LED lightweight AES cipher

```
import sys
if sys.version_info >= (3,):
    xrange = range
__all__ = ['scalarmult', 'scalarmult_base']
# implementation is a translation of the pseudocode
# specified in RFC7748: https://tools.ietf.org/html/rfc7748
P = 2 ** 255 - 19
A24 = 121665
def cswap(swap, x_2, x_3):
    dummy = swap * ((x_2 - x_3) % P)
    x_2 = x_2 - dummy
    x_2 %= P
    x_3 = x_3 + dummy
    x_3 %= P
    return (x_2, x_3)
def X25519(k, u):
    x_1 = u
    x_2 = 1
    z_2 = 0
    x_3 = u
    z_3 = 1
    swap = 0
    for t in reversed(xrange(255)):
        k_t = (k >> t) & 1
        swap ^= k_t
        x_2, x_3 = cswap(swap, x_2, x_3)
        z_2, z_3 = cswap(swap, z_2, z_3)
        swap = k_t
```

```
        A = x_2 + z_2
        A %= P
        AA = A * A
        AA %= P
        B = x_2 - z_2
        B %= P
        BB = B * B
        BB %= P
        E = AA - BB
        E %= P
        C = x_3 + z_3
        C %= P
        D = x_3 - z_3
        D %= P
        DA = D * A
        DA %= P

        CB = C * B
        CB %= P
        x_3 = ((DA + CB) % P)**2
        x_3 %= P
        z_3 = x_1 * (((DA - CB) % P)**2) % P
        z_3 %= P
        x_2 = AA * BB
        x_2 %= P
        z_2 = E * ((AA + (A24 * E) % P) % P)
        z_2 %= P
    x_2, x_3 = cswap(swap, x_2, x_3)
    z_2, z_3 = cswap(swap, z_2, z_3)

    return (x_2 * pow(z_2, P - 2, P)) % P
# Equivalent to RFC7748 decodeUCoordinate followed by decodeLittleEndian
def unpack(s):
    if len(s) != 32:
        raise ValueError('Invalid Curve25519 scalar (len=%d)' % len(s))
    t = sum(ord(s[i]) << (8 * i) for i in range(31))
```

```
    t += ((ord(s[31]) & 0x7f) << 248)
    return t
def pack(n):
    return ''.join([chr((n >> (8 * i)) & 255) for i in range(32)])
def clamp(n):
    n &= ~7
    n &= ~(128 << 8 * 31)
    n |= 64 << 8 * 31
    return n
def scalarmult(n, p):
    '''
      Expects n and p in the form as 32-byte strings.
      Multiplies group element p by integer n. Returns the resulting group
      element as 32-byte string.
    '''

    n = clamp(unpack(n))
    p = unpack(p)
    return pack(X25519(n, p))
def scalarmult_base(n):
    '''
      Expects n in the form as 32-byte string.
      Computes scalar product of standard group element (9) and n.
      Returns the resulting group element as 32-byte string.
    '''

    n = clamp(unpack(n))
    return pack(X25519(n, 9))
```