CS 435/890BN Project (Spring/Summer 2017)

Due: 9:00am, August 18, 2017
(This project work should complete individually. It is NOT a team project)

Note: There are two portions for the project. The programming implementation is the first portion where source code with internal comments are required, and a report is the second portion. Both portions are due on August 18, 2017, at 9:00am. Please compress them into one zip file with name P-YourName-StudentID.zip, and submit through UR Courses. Late submission without sufficient reason and/or permission from instructor will not be accepted.

The first portion is programming implementation:

An RC4 state based secure unicast protocol is described in the following: Suppose A (sender) and B (receiver) have the same secure key to initialize RC4 state $(S, i, j)_A$ for A and $(S, i, j)_B$ for B and initially $(S, i, j)_A = (S, i, j)_B = (S, i, j)_o$. Initially A and B also set their sequence counters to zero. Each data packet has 272 bytes (4 bytes for sequence counter value, 252 bytes for data segment and 16 bytes for hash value):

SC (4 bytes)	Data Segment (252 bytes)	HV (16 bytes)

For Sender:

1. The sender divides the input plaintext message into contiguous 252-byte data segments and assigns SC to each of them. The sequence counter (SC) value is increased by 1 in increased order (initially $SC_A = 0$). If there are not enough data in the data segment of the last data packet, pad a 1 followed by as many 0 as necessary.
2. The sender calculates the hash value for that data packet by inputting SC and the unencrypted data segment, and then places the 128-bit hash value into the data packet.
3. The sender produces the encrypted data packets by only encrypting data segment and hash value (do not encrypt SC value). The sender updates its SC_A and $(S, i, j)_A$ after the encryption.

For Receiver:

Initially $(S, i, j)_B = (S, i, j)_0$ and $SC_B = 0$. When receiving a new packet, B compares its own SC value (SC_B) with the SC value of the received packet. If the difference of the SC

value of the packet and its own SC value (SC_{packet} - SC_B) is 0, then $(S, i, j)_B$ is used as the RC4 state to decrypt the data segment and hash value of that received packet and then increase the sequence counter by 1. Otherwise, calculate the right RC4 state from current $(S, i, j)_B$ by applying certain rounds of PRGA or IPRGA, and then use the right RC4 state to decrypt the data segment and hash value of that received packet and set the sequence counter value of receiver by the SC value of the packet plus 1. B also needs to calculate the hash value according to the decrypted data (SC and data segment) and then compare it with the one directly get from decrypted packet. If they are not the same, B requests A to resend the packet.

Write a program with two interfaces (one for sender and another for receiver) to implement the above secure unicast protocol.

Note:

1. You can use any common programing language to implement.
2. The plaintext, the key and the offset value are input to your program. You should not hard code them or put them into a file, as they should be inputted through the software interfaces. Before run your program, the same key and offset value should be inputted through interfaces for both sender and receiver, but plaintext should only be inputted through the interface for the sender.
3. Your program should be able to handle any key with key size from 0 to 256 bytes, any plaintext, and any offset value, but (4) and (5) list the requirement for what key, offset value, and plaintext you should use for the screenshot with your report.
4. In the screenshot for your report, you should use ABCDEF0123456789ABC2017||your ID number (9 hexadecimal digits) as the secure key (128 bits), and "the last two number of your ID" mod 16 as the offset value for hash value calculation (for example your ID number is 123 456 789, then 89 mod 16 = 9 is the offset value you should use). There is no requirement for your input message, but the input message should be readable.
5. In your report, please test your program by the following cases. Input message is 1000 bytes long (4 packets) for case a, b and c. Please demo with screenshot the entire process with detailed description in your report.
    a. Case 1: the sequence of the packets received is 0, 1, 2 and 3
    b. Case 2: the sequence of the packet received is 1, 0, 3 and 2
    c. Case 3: the sequence of the packet received is 3, 2, 1 and 0
    d. Case 4: input message is 1800 bytes long, sent in increased order
6. **You have to use RC4-BHF (see the appendix) as the hash function to calculate the hash value.**
7. Your source codes should contain internal comments.

The second portion is a report:

The **first part of the report** is required for all students, and the **second part of the report** is only required for CS890BN students (CS435 student does not require to write the second part of the report, but please clearly indicate you are taking CS435 on report).

The first part of your report is your summary and review of the programming implementation. The report should at least include but not limited to the introduction to your program, external comments, how to run your program, screen shots of running your program (demo) in 4 cases, and your thoughts for the programming implementation, etc.

The second part of your report is open topic, but the topic(s) should relate to RC4, RC4-like stream cipher Spritz, the above implemented RC4 state based secure unicast protocol and hash function RC4-BHF, or other related mechanism(s). Some examples of what you can write for your second part (not limit to) are in the below. There is no page or length limit or requirement for this part, but it must be your own idea/understanding and written by your own words (not copied from others or other articles):

- You can read some papers related to RC4, RC4-like stream cipher Spritz, other related mechanisms, and their applications, and write a summary or survey to summarize your learning.
- You can write down your understanding, thoughts and idea about the above implemented RC4 state based secure unicast protocol and hash function RC4-BHF.
- You can think what you can do to improve the RC4 state based secure unicast protocol and hash function RC4-BHF, how to extend their usages and increase their security, and write them down in detail.

Please list all the references if there is any.

Appendix – RC4-BHF:

This appendix introduces RC4-BHF, an RC4 based hash function. The input of RC4-BHF is a message of arbitrary length up to 65536 bits and a non-negative integer which is called offset number. The maximum length of input message is 65536 bits and the output is a fixed-length hash value to be 128-bit long. The internal state of RC4-BHF is a 256-byte permutation. N is a non-negative integer which can be arbitrary length and we suppose the length of the input of RC4-BHF is N-bit. N can be zero, it can be arbitrarily large, and it does not need to be a multiple of eight.

RC4-BHF performs the following steps to calculate the hash value from an input message: (1) Append Padding Bits and Length, and Divide the Padded Message, (2) Compression, and (3) Output. Algorithm 1 and Algorithm 2 give the pseudo code of KSA^* and $PRGA^*$ respectively.

In KSA^*, we use M_k to produce output $STATE_{Mk}$ from $STATE_k$. This involves starting

with S[0] and going through to S[255], and, for each S[i], swapping S[i] with another byte in S according to a scheme dictated by $M_k[i]$.

For PRGA*, it produces $STATE_k$ from $STATE_{Mk}$. This involves starting with S[0] and going through to S[len], and, for each S[i], swapping S[i] with another byte in S according to a scheme dictated by the current configuration of S.

```
Input: Mk and STATEk
Output: Updated Internal State STATEMk

j := 0
for i from 0 to 255
    j := (j + S[i] + Mk[i mod 64]) mod 256
    swap values of S[i] and S[j]
endfor
```

Algorithm 1: The Pseudo Code of *KSA**

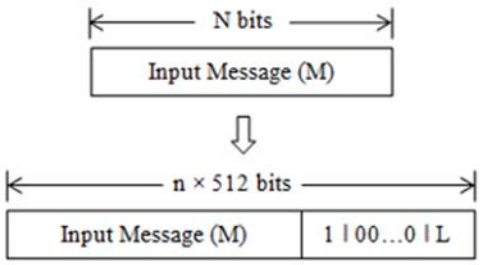
```
Input: Internal State STATEMk and len
Output: Updated Internal State STATEk

for i from 0 to (len – 1)
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
endfor
```

Algorithm 2: The Pseudo Code of *PRGA**

Step 1: Append Padding Bits and Length, and Divide the Padded Message

The input to this step is the input message of arbitrary length, and the output is one or multiple of 512-bit message blocks. The message is appended with padding bits and length so that the length in bits is congruent to 0, modulo 512. That is, the message is extended so that the length is an exact multiple of 512 bits. Padding is always performed, even if the length of the message is already congruent to 0, modulo 512. The appending process is illustrated in Figure 1.

Appending padding bits is performed as follows: a single "1" bit is appended to the input message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 496, modulo 512. In all, at least 1 bit and at most 512 bits are appended. A 16-bit representation of N (the length of the input message, before the padding bits were added) which we denote as L is appended to the result of the previous padding bits process. At this point the resulting message (after appending with padding bits and N) has a length that is an exact multiple of 512 bits.



- M is the N-bit input message
- || is the concatenation
- L is a 16-bit or 64-bit data segment to indicate the length of the input message M
- The number of 0 v is the least non-negative integer that satisfy $N + 16 \text{ (or } 64) + 1 + v \equiv 0 \bmod 512$

Figure 1: The Appending Process

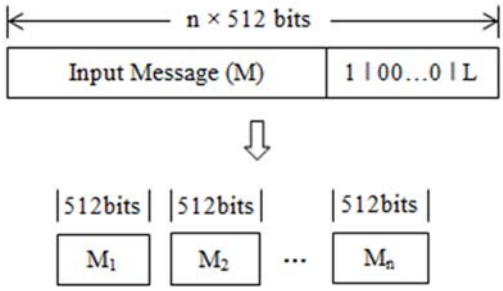


Figure 2: The Dividing Process

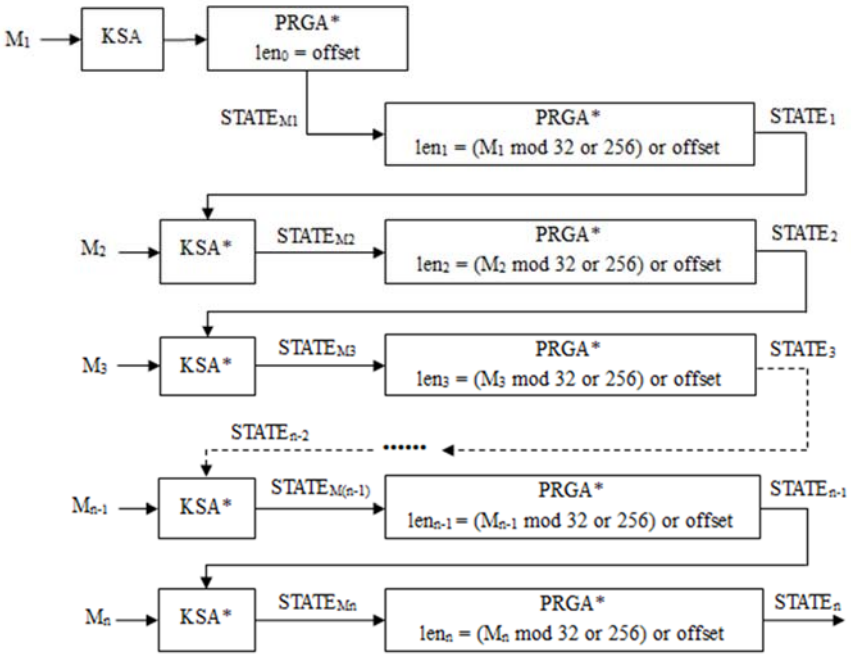


Figure 3: The Compression Step

After the message appending with padding bits and L, it divides the appended message into one or multiple of 512-bit message blocks, notated by $M_1, M_2, \cdots, M_n$. The dividing process is illustrated in Figure 2.

Step 2: Compression

Figure 3 illustrates the compression process. M_1 and the offset value, which is a non-negative integer, are inputted to initialize S. Note that M_1 is treated as the key to the KSA algorithm.

$$\mathrm{STATE_{M1}} = \mathrm{PRGA^*(offset, KSA(M_1))}$$

Then PRGA* updates S according to len_1 as below:

$$\mathrm{STATE_1} = \mathrm{PRGA^*(len_1, STATE_{M1})}$$

$$\text{where } len_1 = \begin{cases} \mathrm{M_1 \bmod 256} & \text{if } (\mathrm{M_1 \bmod 256}) \neq 0 \\ \text{offset} & \text{if } (\mathrm{M_1 \bmod 256}) = 0 \end{cases}$$

For $k > 1$ $(k = 2, 3, \ldots, n)$, S are updated step by step as follows:

$$\mathrm{STATE_{Mk}} = \mathrm{KSA^*(M_k, STATE_{k-1})}$$

$$\mathrm{STATE_k} = \mathrm{PRGA^*(len_k, STATE_{Mk})}$$

$$\text{where } len_k = \begin{cases} \mathrm{M_k \bmod 256} & \text{if } (\mathrm{M_k \bmod 256}) \neq 0 \\ \text{offset} & \text{if } (\mathrm{M_k \bmod 256}) = 0 \end{cases}$$

In the compression process, how many times PRGA* runs is controlled by len_k. Regarding how to calculate "M_k mod 256", the default method is listed in the following: every 8 digits binary can convert to a non-negative integer between 0 and 255, since the length of M_k is 512 bits M_k can be divided into 64 1-byte blocks, then add them up, modulo 256 to calculate an integer as the result of "M_k mod 256". For example, dividing a M_k into 64 1-byte blocks, such as 00000000 00000001 00000010 ..., the result of $(M_k \ mod \ 256)$ is $(0 + 1 + 2 + \cdots) \ mod \ 256$.

Step 3: Output

The first part of the output step is performed as follows: Using the final output of the compression step, the 256-byte $STATE_n$ as the key to KSA, and then PRGA to generate a 512 bytes output, and discards the first 256 bytes and only keeps the last 256 bytes in the calculation. XORing $STATE_n$ with the last 256 bytes of the PRGA output to generate a 256-byte output which is denoted as O_{256}. The process of the first part of the output step is illustrated in Figure 4.

The second part of the output step is used to reduce the size of the hash value, which is illustrated in Figure 5. The final hash value is generated from the least significant bit of each odd number and the final hash value is 128-bit long.

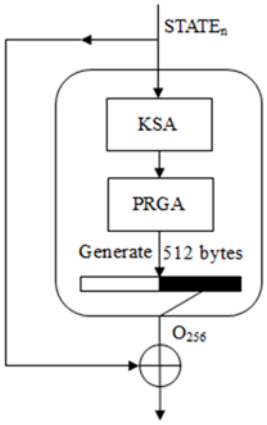


Figure 4: The First Part of the Output Step

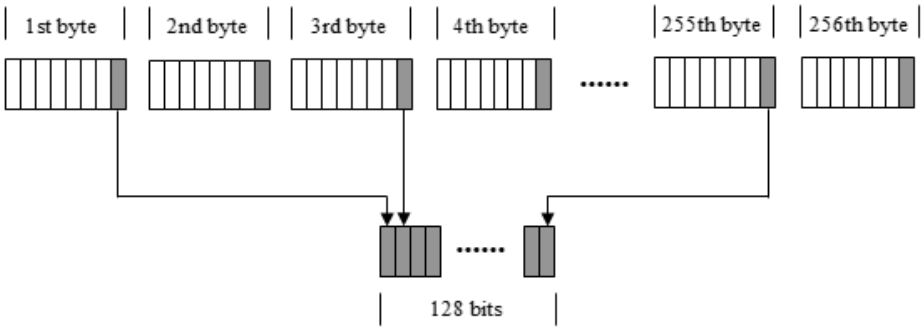


Figure 5: The Second Part of the Output Step