AES implementation in TinyOS

Paolo Toldo, Matteo Saloni, Nicola Manica

June 19, 2008

Abstract

AES is the new standard of cryptography. We present a simple AES implementation for a wireless sensors network showing that in addition with other tecniques it can be used.

1 Introduction

Security in a sensors network is an important and an interesting topic. The importance resides into the diffusion of the sensors networks: in fact the use of such networks is more and more widespread in many applications. They are used in many fields, for example ocean and wildlife monitoring, manufacturing machinery performance monitoring, building safety and earth-quake monitoring and also for many military applications. For an extensive discussion see [1].

In our project we implement Advanced Encryption Standard (AES). The use of this algorithm is not very spread in WSN due to the fact that the limited processing and memory resources on sensor nodes make the implementation not efficient and thus slow in running time [3].

Anyway we decide to study it because we think that with few assumptions, this encryption/decryption mechanism can be implemented and also it can be very useful to add security to WSN communications.

2 The Problem

2.1 AES

The advanced encryption algorithm is one of the most popular algorithms used in symmetric key cryptography.

The standard foresees that an implementation of the algorithm has to support at least one of the three different key length: 128, 192 and 256 bits. According as the key length other parameters are imposed explicity by the standard (block size (Nb) and numbers of rounds (Nr)). The algorithm use moreover an initialization vector (IV) to prevent that sending the same data is produced the same cypertext. This IV is normally summed to the key and then trasmitted added to the packets.

The input and output for the AES algorithm each consist of sequences of 128 bits. Other input and output are not permitted.

Moreover the standard defines several different steps which have to be applied:

- Chiper transformation
  - SubBytes()
  - ShiftRows()
  - MixColumns()
  - AddRoundKey()
- Key expansion
- Inverse chiper
  - invSubBytes()
  - invShiftRows()
  - invMixColumns()
  - Inverse of the AddRoundKey()

For the complete description of the algorithm see [2].

2.2 TinyOS and nesC

TinyOS is a free and open source operating system targeting WSN. It is an embedded operating system written in the nesC programming language as a set of cooperating tasks and processes.

... TinyOS applications are written in nesC, a dialect of the C programming language optimized for the memory limitations of sensor networks.

TinyOS programs are built out of software components, some of which present hardware abstractions. Components are connected to each other using interfaces. TinyOS provides interfaces and components for common abstractions such as packet communication, routing, sensing, actuation and storage ([4]).

2.3 Implementing AES in TinyOS

The main problem to implement the AES in TinyOS is derived from the origin of the algorithm because when it was designed, the main aim was the efficiency and the speed to encrypt a message.

Instead, in a WSN the main issue is the memory occupation while the time to encrypt a message do not represent a big problem because usually the time requested to routing packets is greater.

For these reasons AES are not used in WSN. In our project we focus on memory consumption of the algorithm and show that it can be efficiently used on WSN.

3 Our Solution

3.1 Assumptions

In our implementation we assumed the followings:

- pre-shared key: the simmetric key needed for AES algorithm is store directly in the sensors. Therefore in our implementation we didn't concern about distribution of keys.
- fixed key length: we concern only in key size of 128 bit.
- fixed IVs: in a common implementation a new IV is generated every packet sended. Instead here we used a fixed matrix that allow us to change in every step the key. This can be a lack of security because every 256 message is probably that the key is used in combination with the same IV.

  In addition, rather than sending the entire IV, a simple integer is exchanged between sender and receiver to coordinate their use of the matrix. We decided that because the best method to generate a completely new IV is using an hash function that we didn't implement and it is outside of the target of our project.
- simple checksum: for the previous reason we use a simple sum to simulate the behavioural of the checksum.

3.2 Interfaces

We implement the algorithm as a standard TinyOS interface. Our program consists in many small operations, each one consists in one function, but the most importants and the only accessible from the outside are:

- char aes_encrypt(unsigned char *input, unsigned char *output, unsigned char *key, int size): this is the encryption procedure. It takes as input the data (128 bits), the key, the key size and he gives as output the encrypted data (128 bits). This functions calls in order the other ausiliary procedure.
- char aes_decrypt(unsigned char *input, unsigned char *output, unsigned char *key, int size): this is the decryption procedure. It has the same signature of the previous procedure but it does the inverse. The procedures called are not the same as the encryption because in AES the two operations are not symmetric.

The others ausiliary functions are needed only inside the encryption/decryption procedure and they corresponds to the steps in the algorithm.

A better solution could be to hidden the "private functions" using another interface that link in a correct way our interface.

3.3 Implementation

To implement the AES in TinyOS we started from the C code of SSH. There are some basic issue that we must know before starting to convert the C code to nesC code.

First of all, there is no opportunity to use local variables: all of these must be converted into globals and adding new ones every time there is a domain collision. This was reflected in a big grow of the number of variables that are needed to maintain an index for the algorithm's loops.

Further is the conversion between 32 bit integer (that are most used into C implementation) and 8 bit integer (used in TinyOS to save memory). We implement some functions to do the conversion and so we can transmit value of 16 or 32 bits.

4 Testing

4.1 Scenarios

In this section we present some scenarios that we used to test our application. This scenarios are evaluated with the tossim simulator due to the impossibility to test directly on the motes.

In the following example we use a nice features of the tossim simulator, that is the possibility to create a packet and to inject it in the simulated net. This allow us to send the key to the nodes without inserting it directly to the code. Our application in fact, the nodes wait since they receives the key and only after that the nodes start to send and to receive data.

The first one is the simplest: two sensor with the preshared key are communicated. The example must show that the two sensors are able to communicate correctly.

The second scenario is slightly different. From the previous situations we add another node that have a wrong key. In this situation the third node received the packets sended by the others (because they are sended in broadcast) but it must be not able to understand the data (also having the IV).

4.2 Results

In the experiments show that our algorithm is correct. In the first scenario we send a data and we show that is correctly received. The data sended is 3 2 3 164 95 8 200 137 185 127 89 128 3 139 131 89 and the data crypted is 245 251 254 127 173 91 85 214 20 120 182 3 164 164 246 118. The receiver read the data 245 251 254 127 173 91 85 214 20 120 182 3 164 164 246 118 and after the decryption is 3 2 3 164 95 8 200 137 185 127 89 128 3 139 131 89 , the same data sended.

The second scenario we highlite the same considerations for the first two node but for the third is slightly different. The third node received the data 245 251 254 127 173 91 85 214 20 120 182 3 164 164 246 118 but trying the decryption with a wrong key the data is 146 233 124 217 208 33 238 187 71 167 60 115 195 143 65 88.

We show these results in figure 1. The nodes boot the application (lines 1-3) and after the all the nodes waits for the key.

Our simulation inject a packet that spread the key over the nodes. The right key is received by the nodes 1 and 2 and a wrong key is received by the third mote.

From line 16 to 24 the communication process starts. The second and the first node encrypt and send respectively a packet over the channel.

After line 26 the third mote (the one that has a wrong key) received the data and it is not able to understand the communication (line 29). Also our simple checksum signal that the packet is not correct (line 30).

Instead, the last five lines show the decryption process in node 1. It has the correct key and it is able to decrypt correctly the data.

4.3 Comparison with other implementations

To verify that we implement the "correct AES algorithm" and not another encryption algorithm, we compare our results with other well-know algorithm.

First we compare with the ssh implementation and the result is shown in and with the JavaScript implementation founded at this web site. The comparisons are performed setting the same IV, key and data.

The results are the same as our implementation and we can conclude we have implemented AES algorithm.

4.4 Efficiency and memory consumption

Our implementation is taken from ssh code and so, the complexity of the alghoritm is more or less the same.

The memory needed to allocate all the variables from algorithm is estimated in about 1082 byte.

In our implementation we use a matrix to create a new IV every time the algorithm work. This is a waste of memory, as we just said, the best way to create a new IV is using an hash function.

This allocation can also be very efficiently reduced if, instead of using a fixed S-box and a reverse S-box, we calculate every time the corresponding value. This modified reduce the memory allocation but increase the complexity.

5 Conclusions

In our project we implement correctly the AES algorithm. The tests point out that it works fine and with the same results of other implementation.

As we said before, to avoid some lack of security the algorithm must integrate an hash function. The data exchanged between sender and receiver can be used as input of the hash function. In this manner every packet has a new IV and the ripetition of this depends only in the size of the integer sended in clear. This is also can be extended using a larger integer value. With an 32 bits integer the repetition probability is 2^{32} multiplied with the probability that the hash function return different values from different inputs.

Moreover the hash function must be used to add a integrity check to the packet: in our application we don't care about the trasmission error and the sum operation that we perform is clearly not a good solution.

In addition, a very nice improving could be to implement our code following NetSec. This allow applications that uses this framework to use AES encryption without effort like the work presented in [3].

Finally, in our implementation we spare some space implementing the decryption procedure. In real applications, the decryption alghorithm is only needed into basestation, that is probably not affected by the low memory space like the sensors.

References

[1] Madhukar Anand, Eric Cronin, Micah Sherr Matthew, A. Blaze Zachary, and G. Ives Insup Lee. Sensor network security: More interesting than you think. In *Departmental Papers (CIS)*, University of Pennsylvania, 2006.

[2] National Insitute of Standards and Technology (NIST). Advanced encryption standard (aes). In *Federal Information Processing Standard Publication 197*, November 2001.

[3] Andrea Vitaletti and Gianni Palombizio. Rijndael for sensor networks: is speed the main issue? In *WCAN*, 2006.

[4] Wikipedia. Wikipedia, the free encyclopedia.

A Figure

```
DEBUG (1): Boot of the Application 1
DEBUG (3): Boot of the Application 3
DEBUG (2): Boot of the Application 2

DEBUG (3): Wating for the key
DEBUG (1): Wating for the key
DEBUG (2): Wating for the key
DEBUG (3): Received packet of length 22.
DEBUG (3): Key set: 221 161   2   4   5 255   7   8  10  11  12  13  15  16  17  18
DEBUG (2): Received packet of length 22.
DEBUG (2): Key set:   0   1   2   3   5   6   7   8  10  11  12  13  15  16  17  18
DEBUG (1): Received packet of length 22.
DEBUG (1): Key set:   0   1   2   3   5   6   7   8  10  11  12  13  15  16  17  18


DEBUG (2): SecureComunicationC: timer fired, counter is 1.
DEBUG (2): Crypted Data:   245 251 254 127 173  91  85 214  20 128 102   3 164 164 246 118
DEBUG (2): Original:         3   2   3 164  95   8 200 137 185 127  89 128   3 139 131  89
DEBUG (2): SecureComunicationC: packet sent.

DEBUG (1): SecureComunicationC: timer fired, counter is 1.
DEBUG (1): Crypted Data:   245 251 254 127 173  91  85 214  20 128 102   3 164 164 246 118
DEBUG (1): Original:         3   2   3 164  95   8 200 137 185 127  89 128   3 139 131  89
DEBUG (1): SecureComunicationC: packet sent.

DEBUG (3): Received packet of length 22.
DEBUG (3): Receive 2
DEBUG (3): From net:       245 251 254 127 173  91  85 214  20 128 102   3 164 164 246 118
DEBUG (3): Decrypted Data: 146 233 124 217 200  33 238 187  71 167  60 115 195 143  65  88
DEBUG (3): Error in the decription (2282 - 1503)

DEBUG (1): Received packet of length 22.
DEBUG (1): Receive 2
DEBUG (1): From net:       245 251 254 127 173  91  85 214  20 128 102   3 164 164 246 118
DEBUG (1): Decrypted Data:   3   2   3 164  95   8 200 137 185 127  89 128   3 139 131  89
DEBUG (1): Correct decription
```

Figure 1: Output of Tossim executing the second scenario