SEIS 720

Building the Advanced Encryption Standard Algorithm in AngularJS

Jason Baker
Graduate Programs in Software
Fall 2015

University of St. Thomas, St. Paul, Minnesota

Overview

The Advanced Encryption Standard (AES) is one of the most popular block encryption algorithms used throughout the world today. Also known as Rijndael, AES was developed by Belgian security experts Joan Daemen and Vincent Rijmen. The National Institute for Standards and Technology (NIST) selected AES as the encryption standard for the United States in 2000. The standards organization selected AES over competing algorithms due to its "elegance, efficiency, security, and principled design" (Daemen & Rijmen, 2002).

I decided to take on the challenge of writing the AES algorithm in order to broaden my understanding of the block cipher, and to provide a platform that other students could use to experiment with AES. The paper, *Advanced Encryption Standard by Example*, written by Adam Berent, heavily influenced my algorithm implementation.

Algorithm design

AES is a block cipher that works on 16-byte blocks of data. The algorithm uses a key to encrypt the data, which may be of 128-bit, 192-bit, or 256-bit lengths. The original Rijndael algorithm allowed for varying block lengths, however, NIST decided to standardize on 16-byte blocks.

The AES algorithm is comprised of two major processes, each containing multiple sub-processes (functions). The two major processes are key expansion and iterative cipher rounds. The following information represents a very basic description of the encryption algorithm. The actual algorithm implementation requires an understanding of numerous requirements, which are elaborated in much greater detail in public design documents (see NIST FIPS-197).

Key expansion is the first major process performed by the cipher algorithm. When a key is provided to the algorithm, the key is expanded to increase its length. The key expansion process allows the algorithm to use a different 128-bit key for each round of the iterative cipher rounds. For example, the key expansion process expands a 16-byte (128-bit) key to 176-bytes, and a 32-byte (256-bit) key to 240-bytes.

The key expansion process uses a series of substitution and diffusion functions exercised on 4-byte words representing subsets of the key. These functions are executed over multiple iterations to produce an expanded key. The functions include the following:

Rotate word: performs a circular shift on 4-byte (1 word) of the key

Sub word: applies an S-box substitution to each element in a 4-byte word

Key offset: returns a 4-byte word of the expanded key based on an offset

Round constant: returns a specified constant value based on the current key expansion round number

The iterative cipher rounds perform the encryption and decryption of a provided message. AES leverages substitution-permutation functions and the mathematics of finite fields to produce results that are invertible. This feature is important because it means that data, which is encrypted by AES, may be decrypted and returned to its original form.

During the cipher encryption process, the message is converted into a state matrix and processed through multiple iterative encryption rounds depending on the length of the key. The algorithm uses 10 rounds of cipher functions to encrypt a message using a 128-bit key, and 14 rounds of functions to encrypt a message using a 256-bit key.

Each round of cipher functions uses a different key based on a 16-byte slice of the expansion key. The algorithm may utilize up to four different functions during each encryption round:

Substitute Bytes: each byte in the 4x4-byte state matrix is replaced with a byte in an S-box

Shift rows: each row (4-byte word) in the 4x4 state matrix is shifted by a specified number of offsets

Mix columns: each column in the state matrix is transformed by a fixed matrix via multiplication over a finite field

Add round key: the state matrix is combined with a round key via a bitwise XOR operation

The decryption process is almost the same as the encryption process, with the exception that many of the functions are executed in reverse order.

Application development

I wrote the AES algorithm in Javascript since it's my language *du jour* and it's widely supported in every modern web browser and operating system. While Javascript is an interpreted language, it’s not necessarily slow because Javascript engines are highly optimized by companies like Google and Microsoft.

Sometimes Javascript presents challenges when implementing an algorithm that relies on bitwise operations. Variables in Javascript are loosely typed and math is always done in double-precision floating point. However, these issues were not a significant factor when implementing the AES algorithm due to its byte-oriented nature. All data was converted into 8-bit decimal values in this implementation (actually stored as 32-bit values).

I designed the AES code library and instrumentation interface around the AngularJS MVC framework. While a framework wasn't an absolute necessity for this project, it presented a number of benefits. First, to my knowledge the implementation of the AES algorithm in this framework had never been attempted before. Second, the framework provided an excellent set of methods to instrument the algorithm. Finally, it allowed me to rapidly incorporate a full test suite into the application.

I kept the application functionality fairly limited since it was designed to be a learning tool and not a commercial implementation. While key sizes from 128-bit to 256-bit are fully supported, only a single 16-byte block can be encrypted or decrypted at a time -- basically representing Electronic Code Book (ECB) mode. You can't use this implementation of the algorithm to encrypt an entire document.

The application does not represent a reference implementation. While I attempted to follow best practices in code design, I valued readability over brevity and performance. The code implementation makes use of several algorithmic shortcuts by leveraging pre-calculated tables instead of performing raw computations. A form of logging is built into the AES functions for testing and analysis purposes.

I implemented proper unit testing wherever possible to validate the AES algorithm functions. A full test suite containing over 40 tests was used to validate each of the individual functions within the key expansion and iterative cipher rounds. During the software development process, I would write a test with provided inputs and expected outputs, and then begin writing the function to pass this test (i.e., Test Driven Development). I used test vectors from the official NIST FIPS-197 document, and various other sources, to validate my implementation.

I faced two challenges during the development process. The first challenge was related to representing state in the algorithm. Conceptually, the AES algorithm represents state using a 4x4 matrix. It's possible to implement this representation using a one-dimensional array with offsets or a two-dimensional array. It seemed like a two-dimensional array would better align with the matrix concept. Javascript doesn't support two-dimensional arrays, so I had to replicate the state matrix using an array of arrays structure.

The second challenge involved implementing the mix column function during the iterative cipher rounds. This is probably the most challenging part of the algorithm from an implementation standpoint. Fortunately, I was able to leverage a shortcut offered by Berent using two substitution tables to reduce the number of calculations.

One of the requirements of the mix column function was deceptively simple: Any number multiplied by zero equals zero. Initially, I implemented this requirement incorrectly by substituting a zero in the first part of the calculation rather than simply carrying the zero value to the end of the calculation. Interestingly, the test encryption vectors worked properly despite this improper implementation, whereas the test decryption vectors failed. This led to a few tense hours of in-depth troubleshooting to determine exactly where the decryption process was failing. Eventually, I was able to identify the defect and correct the implementation.

Project installation

The application may be installed on any environment supported by Node.js (Linux, Mac, Windows, etc). Both node and the node package manager (npm) must be installed before building this application. The installation process is easy and should take less than 10 minutes (even less if you already have Node installed on your machine).

The installation process requires 3 steps:

1. Install node on your system (if not already installed)
2. Clone the project repository to your system
3. Install required node packages into the project

The easiest way to install node on your platform is via one of the packages at: https://nodejs.org/en/download/

If you are on a Mac and use homebrew, you can install node by typing:

```
brew install node
```

Once node is installed on your machine, install the Grunt task runner CLI using the command:

```
npm install -g grunt-cli
```

This installs the grunt command globally on your system so that you can run it from any directory. Grunt automates all of the tasks associated with setting up and running the application.

Next, clone the AES application by typing (in an appropriate directory location on your machine):

```
git clone https://github.com/jasondbaker/aesproject.git
```

Cloning the project will create a new subdirectory called `aesproject` within your current working directory.

Go to the root of the newly cloned directory (`cd aesproject`) and install the required node packages by typing:

```
npm install
```

It may take a few minutes to download and install the packages. Note, some warning messages may appear during the package installation.

A great way to ensure that everything is working properly before launching the application is to run the test suite by typing:

```
grunt test
```

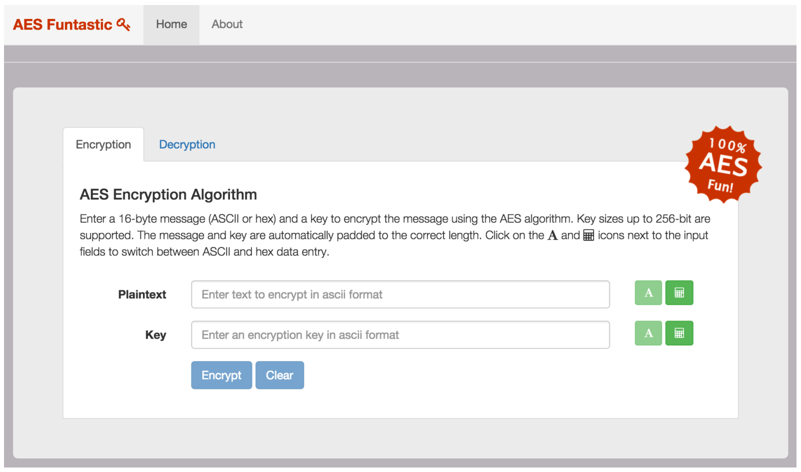
Grunt will run the entire application test suite containing over 40 unit tests.

Using the AES Funtastic Application

Where did the name AES Funtastic come from? I thought writing the code for a sophisticated encryption algorithm was fun, and the fact that I got it working was fantastic! Launching the application is also fantastic. Simply type the following command in the aesproject root directory:

```
grunt serve
```

Grunt will launch a server on port 9000 and a web browser window should automatically open pointing to localhost at this port address. Javascript must be enabled in the web browser to run this application properly. If you are using Internet Explorer, you must be running version 8 or higher. Kill the running grunt process (typically ctrl-c in the terminal) to quit the application.



AES Encryption Algorithm

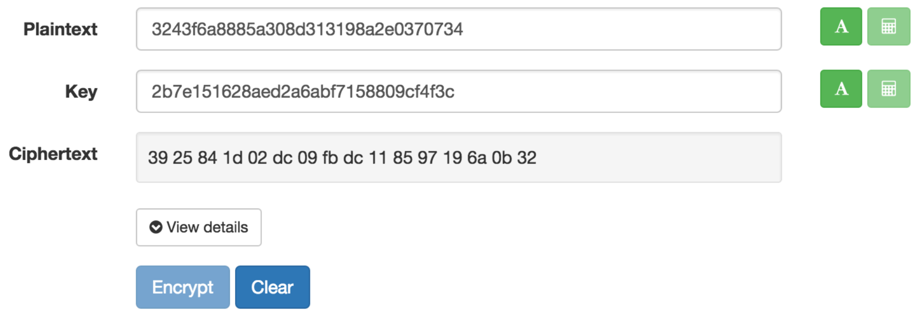
Enter a 16-byte message (ASCII or hex) and a key to encrypt the message using the AES algorithm. Key sizes up to 256-bit are supported. The message and key are automatically padded to the correct length. Click on the A and ▦ icons next to the input fields to switch between ASCII and hex data entry.

Plaintext	Enter text to encrypt in ascii format
Key	Enter an encryption key in ascii format

Figure 1

The application features two main functions – encryption and decryption (see Figure 1). Each function is accessible by clicking on the corresponding Encryption or Decryption panel tab. By default, the encryption panel displays when the application is launched.

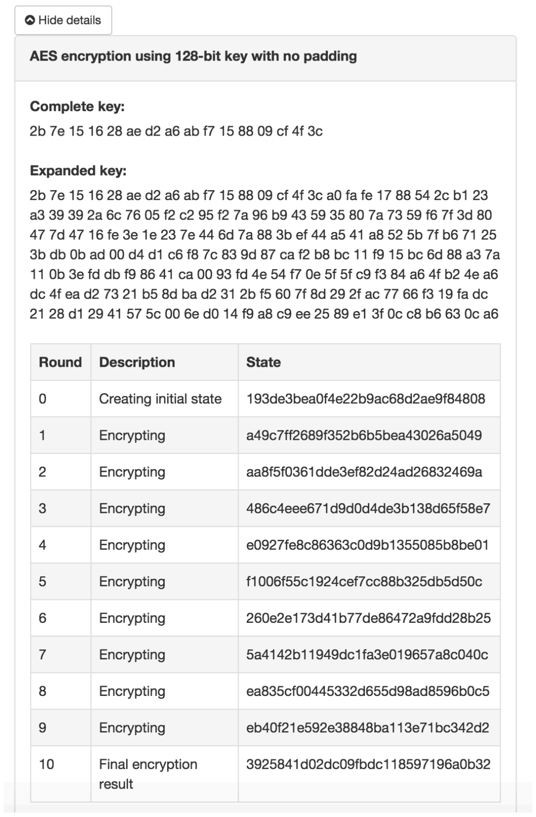
The application operation is fairly straightforward, although the interface hides much of the underlying complexity. You can type in a plaintext string up to 16-characters (bytes), and a key of any length up to 32 bytes. The application will automatically pad plaintext and key entries to meet requirements. For example, if the provided key is 20-bytes, the application will automatically pad the key to create a 24-byte (192-bit) key.



Plaintext	3243f6a8885a308d313198a2e0370734
Key	2b7e151628aed2a6abf7158809cf4f3c
Ciphertext	39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32

Figure 2

It's possible to enter plaintext and key values in either ASCII or hex format by using the input selector buttons located to the right of the input fields. Hex values are entered with two characters (8-bit), and a string of hex values may contain optional spaces. For example, a hex string such as "00 01 02 03 0a 0b 0c" is accepted just as easily as "000102030a0b0c".



Hide details

AES encryption using 128-bit key with no padding

Complete key:

2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Expanded key:

```
2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c a0 fa fe 17 88 54 2c b1 23
a3 39 39 2a 6c 76 05 f2 c2 95 f2 7a 96 b9 43 59 35 80 7a 73 59 f6 7f 3d 80
47 7d 47 16 fe 3e 1e 23 7e 44 6d 7a 88 3b ef 44 a5 41 a8 52 5b 7f b6 71 25
3b db 0b ad 00 d4 d1 c6 f8 7c 83 9d 87 ca f2 b8 bc 11 f9 15 bc 6d 88 a3 7a
11 0b 3e fd db f9 86 41 ca 00 93 fd 4e 54 f7 0e 5f 5f c9 f3 84 a6 4f b2 4e a6
dc 4f ea d2 73 21 b5 8d ba d2 31 2b f5 60 7f 8d 29 2f ac 77 66 f3 19 fa dc
21 28 d1 29 41 57 5c 00 6e d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6
```

Round	Description	State
0	Creating initial state	193de3bea0f4e22b9ac68d2ae9f84808
1	Encrypting	a49c7ff2689f352b6b5bea43026a5049
2	Encrypting	aa8f5f0361dde3ef82d24ad26832469a
3	Encrypting	486c4eee671d9d0d4de3b138d65f58e7
4	Encrypting	e0927fe8c86363c0d9b1355085b8be01
5	Encrypting	f1006f55c1924cef7cc88b325db5d50c
6	Encrypting	260e2e173d41b77de86472a9fdd28b25
7	Encrypting	5a4142b11949dc1fa3e019657a8c040c
8	Encrypting	ea835cf00445332d655d98ad8596b0c5
9	Encrypting	eb40f21e592e38848ba113e71bc342d2
10	Final encryption result	3925841d02dc09fbdc118597196a0b32

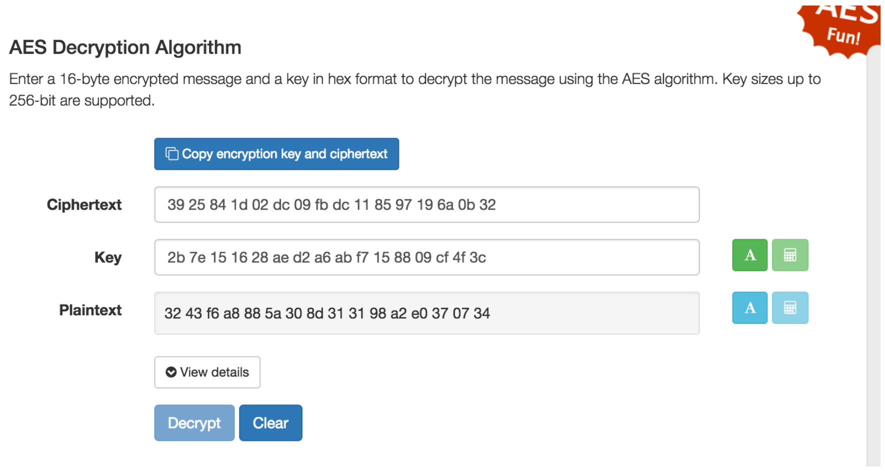
Figure 3

Once the plaintext and key values are entered into the form, click on the Encrypt button to perform an AES encryption. The application will display the computed ciphertext (see Figure 2).

One of the interesting features of the AES application is tracking the transformation of the key and plaintext values as the algorithm progresses through multiple cipher rounds. This application provides a way to see the state changes throughout the encryption process. Click on the "View details" button to see the detailed algorithm logs (see Figure 3).

It wouldn't be any fun if you could encrypt data but not decrypt it as well. Simply click on the Decryption panel tab to access the decryption input form, and enter in the ciphertext and a key. Note that the ciphertext must be in hex format. Click the Decrypt button to execute the decryption process. The application will display the resulting plaintext. Like the encryption operation, you can click on the "View details" button to see detailed decryption logs.

You may notice a special button here if you didn't clear the previous encryption results. If you click on the "Copy encryption key and ciphertext" button, you can easily import the ciphertext and key values from the encryption form. This way you can use the application to encrypt some data and then quickly decrypt it to verify a complete encryption/decryption cycle (see Figure 4).



AES Decryption Algorithm

Enter a 16-byte encrypted message and a key in hex format to decrypt the message using the AES algorithm. Key sizes up to 256-bit are supported.

Ciphertext	39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
Key	2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
Plaintext	32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Figure 4

The plaintext result from the decryption process is displayed in hex format, but this isn't very helpful when verifying plaintext that was encrypted in ASCII format (unless you can "see" hex characters!). The application allows you to easily switch between HEX and ASCII display of the decrypted plaintext by clicking on the buttons to the right of the plaintext field.

You can clear the data from either of the forms at any time by clicking on the "Clear" button at the bottom of each form. The application is smart enough to clear encryption or decryption results if you decide to change any of the input values. This makes it easy to quickly test different combinations of plaintext or ciphertext inputs.

Conclusion

Implementing the AES algorithm in the AngularJS framework was rewarding and intellectually challenging. It was rewarding from the standpoint that I was able to complete a project that was very different from anything I've ever attempted. It was intellectually challenging because it required great persistence to understand the purpose and requirements for each individual cipher function. I look forward to potentially expanding the project in the future, and making it a tool that other students and instructors find useful.

Works Cited

Berent, A. (n.d.). *AES Notes*. Retrieved 11 1, 2015, from Advanced Encryption Standard by Example: http://www.adamberent.com/documents/AESbyExample.htm

Daemen, J., & Rijmen, V. (2002). *The Design of Rijndael.* Heidelberg, Germany: Springer.

National Institute of Standards and Technology (NIST). (2001, 11 26). *Federal Information Processing Standards Publication 197.* Retrieved 11 1, 2015, from Computer Security Resource Center: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf