On the Cryptanalysis of AES

Bryce Hickle

2019-12-09

1 Abstract

AES is the most used private-key encryption scheme today. Surveying the current state of cryptanalysis of AES will allow for evaluating whether AES is still secure and what systematic weaknesses can be found. In this survey, techniques such as linear, differential, algebraic, and neural network cryptanalysis are discussed with respect to AES and concludes with an assessment of how cryptanalysis may develop in the future to test AES.

2 Introduction

First, establishing criteria for security must be done to accurately assess the cryptanalytic results of the papers researched. The Advanced Encryption Standard, or AES, is accepted as secure based on asymptotic notions of security. That is to say, correctly guessing messages from their ciphers should be as likely as guessing the messages randomly (with a bounded amount of negligibly such that the chance is slightly higher than perfectly random). Furthermore, for AES to be considered secure in practice, it must be asymptotically secure from chosen-ciphertext, chosen-plaintext, known-plaintext, and ciphertext-only attacks[3].

Second, establishing the specification of AES and its basic functionality is necessary in order to comprehend what, if any, systematic weaknesses exist or whether an assertion of systematic weakness by any papers are reasonable claims. AES is a private symmetric key encryption scheme that has a 128 bit block length with key lengths that can either be 128, 192, or 256 bit[3,4,6]. AES has a substitution and permutation structure as opposed to older standards that used a Feistel structure[3,4,6]. AES is designed to provide diffusion via linear mixing layers and confusion via the non-linear S-box operations[3,4,6]. AES was designed to be resistant from advanced techniques such as differential and linear cryptanalysis[1,2,3]. In this paper, we will begin with linear and differential cryptanalysis (which remain important modern cryptanalytic techniques today), explore several lesser-known techniques, summarize the current state of algebraic attacks, discuss neural network applications of cryptanalysis specific to AES, and conclude with where it is plausible for cryptanalysis of AES to move forward in the future.

3 Linear Cryptanalysis

Linear cryptanalysis is a famous and important metric for testing modern cryptosystems today. The technique was developed by Matsui in the 1990s [3]. It is a known-plaintext attack that exploits linearity between the ciphers of a cryptosystem and their corresponding plaintexts[3]. If such a relationship exists, then it is possible to construct a system of linear equations where the input bits and output bits are combined to produce the key[1,2,3]. This type of attack is computationally efficient to execute (since solving systems of linear equations is trivial) and requires few known-plaintext pairs[3]. Thank fully, AES was designed with S-boxes specifically to make the cryptosystem non-linear to safe-guard against these kinds of attacks[3,4,6]. Linear cryptanalysis is still used to evaluate encryption schemes.

4 Differential Cryptanalysis

Differential cryptanalysis is also an important and famous cryptanalytic technique. Created in the 1980s, differential cryptanalysis was the first technique to break AES's predecessor, DES[3]. The premise of this chosen-plaintext attack is to calculate the changes in output of encryption schemes when input is changed (hence the term differential)[1,2,3]. Illustrated by a worked example in Introduction to Modern Cryptography, if the differentials, Δx and Δy, calculated are larger than what would be calculated in a uniform distribution between ciphers and messages, then a weak point in the systems security has been found and can be used to recover the key[3]. Once a large differential is found, one can generate a set of pairs of plaintexts that have Δx[3]. By then encrypting these pairs and checking Δy, one can then test a limited number of keys that satisfies the differential relationship between the pairs with a negligibly small chance of getting a false positive[3]. The benefit to this technique is that as long as the encryption scheme is not perfectly secure, some differential relationship greater than a perfect uniform distribution will exist[3]. The downside to this technique is that it is both a chosen-plaintext attack instead of a known-plaintext attack, and that the attack sometimes requires an exponentially large number of chosen pairs to encrypt (such is the case with DES and AES)[3]. AES's S-box was designed not just to prevent linear cryptanalysis, but also to prevent differential cryptanalysis by picking a substitution box with good diffusion properties[3,4,6].

5 Notable Mentions

Some lesser-known cryptanalytic techniques include boomerang, truncated differentials, the square attack, and interpolation attacks. Boomerang and truncated differential attacks are both very similar to normal differential cryptanalysis (where boomerang utilizes quadruples instead of pairs for calculating differentials to exploit and truncated differentials use differentials that are not determined completely)[2]. The Square Attack uses multiple sets of chosen-plaintexts with specific mathematical properties and how these properties change as they are passed into the cryptosystem is used to gather statistical insight into the system to then exploit[2]. Interpolation attacks model the system as a higher order non-linear system of equations and attempts to then solve the system for the key (however, this technique is only useful with very few number of equations with very low order)[1,2]. All mentioned attacks were known at the time of the development of AES and it remains immune to such attacks to this day[2]. In fact all cryptanalytic techniques discussed so far have been known since before the development of AES[2,3]. Furthermore, all advancements in discussed attacks so far are not known to be definitively faster than brute-force when used on AES[2].

6 Algebraic Cryptanalysis

Algebraic attacks are one of the most powerful cryptanalytic techniques used against AES today. These types of attacks attempt to model AES as a system of multivariate quadratic equations (MQ equations) and then solve as one would in linear cryptanalysis[2,4,6]. These types of attacks rely on the fact that AES has a relatively simple algebraic structure because of the design of the S-boxes[4]. One method of attack is to setup a MQ system, compute a basis called a zero dimensional Groebner basis for the system, and then solve[6]. Zhao, Cui, and Xie claim that solving the MQ problem this way would be O(2^N) with N being the number of unknown variables[6]. Also, the highest degree of the non-linear polynomials with Zhao's construction is 254 which I believe would imply about 2 to the 255 calculations asymptotically, because of the zeroeth term in characteristic two Galois Finite Fields used to construct multivariate system of equations related to cryptosystems[6]. While theoretically twice as fast as brute-forcing the key, it is so computationally infeasible that AES would be considered secure by asymptotic definitions of security.

Another method of algebraic cryptanalysis is to solve the MQ problem by

adding more equations to the system than is required and then undergoing a process called relinearization[4]. The XL family of algebraic attacks are based on this concept[1,2,4]. The basic approach is to generate a sufficiently large number of linearly independent equations greater than the number of unknown variables (while still being related to the system), then one can replace the higher order variables with a variables of power one, solve linearly, and brute-force the very limited number of options left to get the value of the original variables[4]. A key insight for this approach is the fact that algebraic dependence does not mean that the equation is necessarily linearly dependent, so increasing the order of every term creates more linearly independent equations even though they are algebraically dependent[4]. However, estimates show that XL on AES-128 would take about 2 to the 330 calculations[4]. This failing lead to XSL and the T method of constructing linearly independent equations[4]. The T method attempts to reduce the total number of terms required to solve in the system of equations[4]. It does so by fixing a parameter P which is a positive integer, multiplying each equation with the products of P-1 monomials that appear in equations that come from the S-boxes that do not share any variables with that equation and similar multiplications occur with respect to the key scheduling and subkey addition equations[4]. If this fails to produce enough equations, one can do the T' method which takes all the equations after the T method, reduce any squares of bit monomials in T[4]. Then, one can perform Gaussian eliminations to produce a few more new equations and this T' process can be repeated[4]. In the best case with both of these methods AES-256 could be broken in about 2 to the 255 which again is computationally infeasible and therefore AES remains asymptotically secure[4].

Another algebraic attack is BES. BES attempts to resolve practical issues of converting from GF(2^8) and back to GF(2)[4]. BES is designed as a logically equivalent, formal model of AES that acts on 128-byte blocks and 128-byte keys[4]. With this model of AES combined with previously mentioned XSL algorithms which make use of T and T' prime methods, it is theoretically possible to crack AES-128 in about 2 to the 100 calculations[4]. BES broken via XSL using T and T' methods is arguably the greatest breakthrough in cryptanalysis of AES. However, the validity of the T' prime method is still an open question in cryptanalysis[4]. Even if valid, an attack that requires 2 to the 100 calculations is still a computationally infeasible attack. In my own opinion; if T' method holds, then I would claim that an attack that is 2 to the 28 times faster than brute-force would be sufficient to call AES-128 not asymptotically secure, and I would recommend AES-256 in practice.

7 Neural Network Cryptanalysis

More recently, neural networks have been used in cryptanalysis. If an entity has sufficient number of known-plaintext pairs, it can train a neural network by feeding it the cipher data and then "teaching" based on how closely its guess matched the corresponding message[5]. This technique disregards attempts to find the key completely and instead black-boxes the entire cryptosystem.

Xinyi, Zhao, and Yaqun state that they are able to use neural networks to recover entire bytes 40 percent of the time from their network after training[5]. They also claim to be able to recover more than half of a byte 63 percent of the time and over half of the plaintext overall with 89 percent accuracy on average[5].

While they do provide diagrams and basic specifications for their networks and the training sets used (one neural network was a cascading neural network and the other was a traditional feedforward neural network with mean squared error being used as both the cost function and test metric with the training set being random numbers written to file in one experiment and the other experiment using the Caltech Image Database)[5]. They fail to provide source code, an algorithm, or even a detailed pseudocode representation of their software (besides some vague flowcharts and descriptions)[5].

Curious and skeptical, I wanted to verify these claims. I have one implementation of the random number experiment closely based on the descriptions presented[5]. The training set is a set of text files with random digits as messages and their corresponding ciphers[5]. The cipher is inputted as 128 bits into a neural network with four layers (128, 256, 256, and 128)[5]. No internal activation functions were used in either the papers description nor in my implementation[5]. Mean squared error was used for cost function and 0.5 (0.5 and higher mapping to 1 and less than 0.5 mapping to 0) was used for rounding purposes after training and testing[5]. My implementation did not use thresholds within the code itself. I simply estimated small samples by hand while comparing numbers after training and testing.

My implementation is not fast nor is it perfect, but it did get similar enough results to view the research of Xinyi's as verified and will be provided in this paper in the appendix section along with example input. My code was able to reach over 80 percent similarity based on mean squared error ratings in predicting messages from ciphers after training. Also, my network is able to get the first four bits of every bit correct nearly one hundred percent of the time (considering threshold applied). Some differences between mine and Xiniyi's experiment, based on the description in *Research on Plaintext Restoration of AES Based on Neural Net-*

work, would seem to be their more fine-grain control of how the logistic cost is gathered (they use XOR instead of integer multiplication and sums used in most neural networks for example)[5]. They could have also had more control on the weights (for example, it is not clear if their starting weight were restricted to positives, reals, integers, etc)[5]. None-the-less, a system that can predict a message given some cipher with over 80 percent chance of being correct is much higher than the probability of guessing the message randomly. A potential flaw in both the paper's and my own experiment may be the encoding of the messages themselves since random digits, encoded in utf-8 for example, will have a very low variance in their numerical value byte-wise. One of the reasons my code is so good at getting the first four bits of the message from a cipher is that the first four bits of the message never changed from byte to byte. Regardless, the ability to get a single byte correct from the set of confusion that is the output of AES-256 is a feat not met by any other technique in cryptanalysis of AES. If these kinds of attacks are not considered side-channel in nature, then they are the most powerful pure cryptanalytic techniques we have today.

If confirmed with further research, neural network attacks clearly prove that AES is no longer asymptotically secure based on the definition presented in this report. That does not mean that AES is immediately broken for: the number of known-plaintext pairs needed inorder to develop a training set is large enough to be extremely unlikely in practice, the amount of access required to perform the training is of the neural network would place the target in a near full-compromise situation where side-channel attack would most likely be more useful anyway, and getting part of a byte correct is still not the correct message from a human interaction standpoint in the real world. All of these factors weaken the practicality of these kinds of attacks, but it also does nothing to refute AES's fundamental insecurity (again if results continue to be verified).

8 Conclusion

In conclusion, AES is one of the most well tested encryption schemes ever created. It was specifically designed to not break under powerful cryptanalytic techniques such as linear cryptanalysis and differential cryptanalysis. These attacks not only are still used to verify sound cryptographic principles such as the principle of confusion and diffusion, but they also serve as inspiration for newer techniques such as truncated differentials and algebraic cryptanalysis. Based on all these techniques, the ingeniousness and simplicity of the non-linearity of the S-boxes

in AES may prove to be its down-fall once mathematical consensus is formed on how to approach algebraic techniques in cryptanalysis. However, neural network cryptanalysis, if not refuted nor designated as implementation-based exploits, will beat algebraic techniques to the punch in terms of breaking AES. By a rigorous standard of asymptotic security it appears to already do so.

References

[1] DEWU, X., AND WEI, C. A survey on cryptanalysis of block ciphers, Nov 2010.

[2] KAMINSKY, A., KURDZIEL, M., AND RADZISZOWSKI, S. An overview of cryptanalysis research for the advanced encryption standard, 2010.

[3] KATZ, J., AND LINDELL, Y. *Introduction to Modern Cryptography*, 2 ed. CRC Press, 2015.

[4] NOVER, H. Algebraic cryptanalysis of aes: An overview, 2004.

[5] XINYI, ZHAO, AND YAQUN. Research on plaintext restoration of aes based on neural network, Nov 2018.

[6] ZHAO, KAIXIN, CUI, JIE, XIE, AND ZHIQIANG. Algebraic cryptanalysis scheme of aes-256 using gröbner basis, Feb 2017.

1 Appendix A

```
"""
Implementation of Cryptanalytic
Neural Network based on 'Research
 on Plaintext Restoration of AES
Based on Neural Network'

Specifically the second experiment
where
the input is standard in of text
files containing random digits

"""
import random
import pyAesCrypt
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from bitstring import ConstBitStream
import os

"""
Gather relevant data from the training set and format
properly for the neural network while
keeping input and correct output in-line
"""
def getData():
#128 to 256 to 256 to 128 layer dimensions
    cipherList = []
    messageList = []
    #gather data from files
    string0 = "m"
    string1 = "c"
    for i in range(0, 1000):
        stringTemp = str(i)
```

```
        fileName = string0 + stringTemp + ".txt"
        fileM = open(fileName, encoding ="utf8")
        for line in fileM:
            for char in line:
                messageList.append(ord(char))
        fileM.close()
        cipherName = string1 + stringTemp + ".txt.aes"
        byte = os.path.getsize(cipherName)
        fileC = open(cipherName, 'rb')
        #handling bitstream from encrypted data
        bitData = ConstBitStream(bytes = fileC.read(byte),
 length = byte*8)
        while(bitData.pos < byte*8):
            B = bitData.read(8).uint
            cipherList.append(B)
        fileC.close()
    #now have sets of bytes ready for training split
up for training and testing phases
    tempC = (len(cipherList)//4) * 3
    tempM = (len(messageList)//4) * 3
    #output organized data to use in network
    aList = [cipherList[1:tempC],messageList[1:tempM],
cipherList[tempC:-1],messageList[tempM:-1]]
    return aList

#need also to organize bytes into bits since
neural network is 128 bit input
def charMake128(aList):
    outputList = []
    finalList = []
    bList = []
    tempList = []
    counter = 0

    if(len(aList) % 16 != 0):
        while(len(aList) % 16 != 0):
            aList.pop()
```

```
    for elem in aList:
        if(counter != 15):
            tempList += [elem]
            counter += 1
        else:
            counter = 0
            tempList += [elem]
            bList.append(tempList)
            tempList = []

    for l in bList: #16 elements
        for num in l:
            #break each num into list of bits
            temp = bin(num)[2:]
            if(len(temp)<8):
                while(len(temp)<8):
                    temp = "0" + temp #padding
            for char in temp:
                bit = int(char)
                bit = [bit]
                outputList += bit
        finalList.append(outputList)
        outputList = []

    return finalList


def breakAES(aList):
    #call helpers to organize data
    cipherData = charMake128(aList[0])
    cipher = cipherData[1:10000]
    messageData = charMake128(aList[1])
    mess = messageData[1:10000]
    testCipher = charMake128(aList[2])
    testC = testCipher[1:100]
    testMessage = charMake128(aList[3])
    testM = testMessage[1:100]
    #network architecture below
```

```
    model = keras.Sequential([
        keras.layers.Dense(256, input_shape=(128,)),
        keras.layers.Dense(256),
        keras.layers.Dense(128)
    ])
    #stochastic gradient descent, logistic calculation,
MSE similarity metric
    sgd = keras.optimizers.SGD(lr=0.01)
    model.compile(optimizer = sgd,
loss="mean_squared_error",
 metrics=["mse"])

    model.fit(cipher, mess, epochs=10)

    test_loss, test_acc = model.evaluate(testC, testM)

    a = model.predict(testC)
    print("predicted")
    print(a)
    print("was actualy")
    print(testM)

#below used for encrypting message data
def enc():
    string0 = "m"
    string1 = "c"
    bufferSize = 64 * 1024
    for i in range(0, 1000):
        stringTemp = str(i)
        fileName = string0 + stringTemp + ".txt"
        cipherName = string1 + stringTemp + ".txt.aes"

        pyAesCrypt.encryptFile(fileName, cipherName,
"training", bufferSize)

#below used for making messages similarily in paper
def messageMaker():
    string0 = "m"
```

```
    for i in range(0, 1000):
        stringTemp = str(i)
        fileName = string0 + stringTemp + ".txt"
        theFile = open(fileName, "w")
        for i in range(0,16384):
            digit = random.randint(0,9)
            digit = str(digit)
            theFile.write(digit)

        theFile.close()


def main():
    data = getData()
    breakAES(data)

if __name__ == "__main__":
    main()
```