**Skipjack Block Cipher**

For COMP 4109

Benjamin Sweett (100846396)

December 8th, 2014

Table of Contents

Introduction 3

Overview 3

Motivation 3

Goals 3

Objectives 3

Background 4

History 4

Encryption 4

Decryption 6

Approach 7

Results 8

Implementation 8

Known Issues 9

Cryptanalysis 10

Conclusion 11

References 13

# Introduction

The document below outlines the scope and results of the Skipjack project that is to be submitted to Dr. John Howat for his Applied Cryptography (COMP 4109) course. It contains information about the Skipjack block cipher along with the output feedback mode. The report also includes a discussion the details my implementation and some problems I encountered, and a basic cryptanalysis of the cipher.

## Overview

The US National Security Agency (NSA) created the Skipjack Cipher in 1987. It is a block cipher encryption algorithm that uses 80-bit keys and 64-bit blocks and runs 32 rounds using an unbalanced Feistel network. It is structurally similar to the DES algorithm covered in class. It was initially designed to encrypt audio data transmissions in government mobile devices and uses the key exchange algorithm (KEA) for sharing keys between peers. The algorithm also supports four block cipher modes of operation for encryption over multiple blocks. All of the modes of operations can also be used for DES. It uses bitwise XOR operations on four 16-bit words in two rule functions in combination with a permutation function that utilizes a predefined table of magic numbers. Using the inverse of the permutation and the rule functions performs decryption.

## Motivation

My motivation for studying this algorithm came from its secret nature. The algorithm specifications were only release in 1998 and only recently has it been proven to be insecure by today’s computing and security standards. I found its shifting and permuting rounds quite interesting and felt like that it would make an interesting topic to explore in greater detail. I was also motivated by its similarity to the DES algorithm that we studied in class.

## Goals

The main goal of this project was to explore the Skipjack block cipher in greater detail. I hoped to analyze the cipher and learn more about why it is no longer secure. I presented my findings to the class in a concise manor in order to impart what I learned about the cipher. I also hoped to learn more about cipher implementations and design. Overall the project aimed to demonstrate why the Skipjack cipher was original used and why it is no longer used today.

## Objectives

The first objective was to create an implementation of the Skipjack algorithm using the Java programming language. I also planned to include some short data test cases for the algorithm to show how it worked. I then intended to implement two block cipher modes of operations that were supported by the Skipjack algorithm. Cipher-Block Chaining (CBC) and Output Feed-Back Mode (OFB) were planned as part of the implement to be used with some larger data test cases. I also gave a short presentation on my findings during the study of this algorithm to the class. This document represents the last objective and is used to describe the algorithm in greater detail, comment on my implementation, and demonstrate a basic cryptanalysis.

# Background

## History

The Skipjack Cipher began development as part of a family of ciphers called Type I in 1980 under the secrecy of the National Security Agency (NSA). It was initially designed along side the NSA’s Key Exchange Algorithm (KEA) and was to be kept as a classified cipher unlike the DES block cipher that was published a few years prior. By 1987 the Skipjack cipher had been standardized and was implemented on security microprocessor called Capstone. Capstone was only ever embedded in one information security system, which was used in a variety of mobile devices owned and operated by the US government. The purpose of the cipher was to encrypt sensitive telecommunications data throughout government agencies. In order to ensure the public could be confident with the security of the cipher a number of outside experts were allowed to study it in 1993. The algorithm was declassified in 1998 after the public cryptography community created a similar algorithm that contained some of the techniques used in the Skipjack cipher. Since its declassification a dozen or so theoretical attacks have been published by experts the claim they can break the cipher faster than an exhaustive search. The cipher was official deemed insecure by National Institute of Standards and Technology (NIST) in 2010 due to its small key size.

## Encryption

The algorithm encrypts one 64-bit block by preforming two rules over 32 rounds. The cipher runs Rule A for 8 rounds followed by Rule B for 8 rounds. It then returns to Rule A for another 8 rounds and switches to Rule B for the last 8 rounds for a grand total of 32 rounds.

wk is one 16-bit data block for k = 1,2,3,4

gk is a computed byte for k = 1,2,3,4,5,6

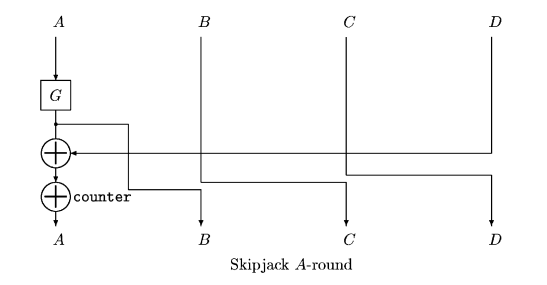
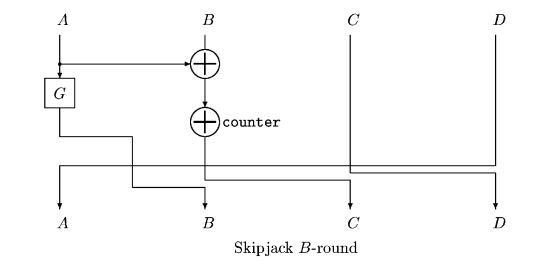
cv4k + i is a computed value from key k for i = 0,1,2,3 modulo the key length

G is a Feistel cipher that takes a 16-bit input with a 4-byte sub key and permutes over 4 rounds as defined in Permutation G.

**Rule A**

* G permutes w1.
* The new w1 is the xor of the G output, w4, and the counter.
* w2 and w3 shift once to the right (becoming w3 and w4).
* The new w2 is the output of G.
* Lastly the counter is incremented by one.

**Rule B**

* G permutes w1.
* The new w2 is the output of G.
* The new w3 is the xor of w1, w2, and the counter.
* w3 and w4 shift once to the right (becoming w1 and w4).
* Lastly the counter is incremented by one.

**Permutation G**

Figure 2 - Rule B

Figure - Rule A

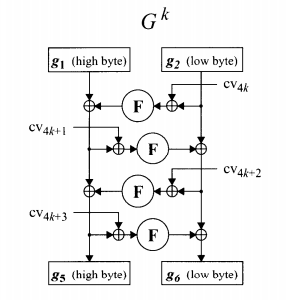
* g1 is the first 8-bits (1 byte) while g2 is the last 8-bits of the input word.
* g3 is the output of F-Table at index g2 xor cv4k and then xor g1.
* g4 is the output of F-Table at index g3 xor cv4k + 1 and then xor g2.
* g5 is the output of F-Table at index g4 xor cv4k + 2 and then xor g3.
* g6 is the output of F-Table at index g5 xor cv4k + 3 and then xor g4.
* g5 concatenated with g6 are returned to the rule as a 16-bit word.

Figure 3 – Permutation G

**F-Table**

* A table of hexadecimal magic numbers as defined by the NSA.
* The numbers are selected by the indexes calculated in Permutation G.

## Macintosh HD:Users:bensweett:Desktop:Screen Shot 2014-11-16 at 2.46.58 PM.pngDecryption

Figure 4 - F-Table

The algorithm decrypts one 64-bit block by preforming two rules over 32 rounds. The algorithm runs Rule B Prime for 8 rounds followed by Rule A Prime for 8 rounds. It then returns to Rule B Prime for another 8 rounds and switches to Rule A Prime for the last 8 rounds for a grand total of 32 rounds.

wk is one 16-bit data block for k = 1,2,3,4

gk is a computed byte for k = 1,2,3,4,5,6

cv4k + i is a computed value from key k for i = 0,1,2,3 modulo the key length

G is a Feistel cipher that takes a 16-bit input with a 4-byte sub key and permutes over 4 rounds

**Rule A Prime**

* G permutes w2.
* The new w1 is the output of G.
* w3 and w4 shift twice to the right (becoming w2 and w3).
* The new w4 is the xor of w2, w1, and the counter.
* Lastly the counter is decremented by one.

**Rule B Prime**

* G permutes w2.
* The new w1 is the output of G.
* The new w2 is the xor of the output of G, w4, and the counter.
* w4 and w1 shift twice to the right (becoming w3 and w4).
* Lastly the counter is decremented by one.

**Permutation G Prime**

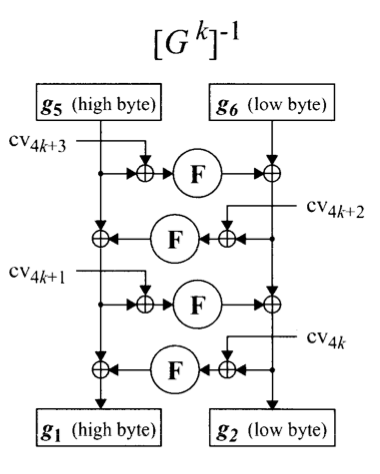
* g1 is the last 8-bits (1 byte) while g2 is the first 8-bits of the input word.
* g3 is the output of F-Table at index g2 xor cv4k + 3 and then xor g1.

Figure 5 - Permutation G Prime

* g4 is the output of F-Table at index g3 xor cv4k + 2 and then xor g2.
* g5 is the output of F-Table at index g4 xor cv4k + 1 and then xor g3.
* g6 is the output of F-Table at index g5 xor cv4k  and then xor g4.
* g6 concatenated with g5 are returned to the rule as a 16-bit word.

**F-Table**

* Decryption uses the same F-Table as shown in encryption.

# Approach

My implementation was done in Java 8 in the Eclipse development environment. I started by working with the cipher itself with single block encryption. I used the official NSA documentation as a guide to make sure my implementation was correct. I implemented the F-Table and the permutation function G and then added separate functions for Rule A and B respectively. I then implemented the encrypt function and used a test vector provided in the documentation to make sure it worked correct before moving on to decryption. The prime functions for decryption were for the most part fairly straightforward being the inverse of their encryption variants. The G-prime operation however proved to be challenging, as its definition in the NSA documentation was much shorter and took some careful consideration.

After laying down the 64-bit cipher I implemented basic JUnit tests for encrypting and decrypting single blocks of hexadecimal data as taken from the NSA documentation. Once I was sure the block cipher was working correctly I tackled the modes of operation. I started with Output Feedback mode (OFB) because it did not require padding. After encountering some issues with signed data types (see Known Issues) I put the OFB class aside and implemented Cipher Block Chaining (CBC) and block padding. However due to a similar issue mentioned above I had to remove it from my implementation, as decryption did not work correctly. After partially fixing the issue I added large data test cases for OFB to confirm that it was functioning properly.

With the mode of operation complete I setup a hexadecimal encoder and decoder and built a console application to demonstrate my block cipher. I created functionality for randomly generating an IV, setting the key, and encrypting text to and from the console and a text file.

After I had worked out some the issues with my implementation I exported the [blockcipher](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/blockcipher) package and the utils package to a .JAR file. I added this file as a release library without the demo application so that the Skipjack cipher can be used with OFB in any future projects.

The last steps were to cleanup my source code, add some exception handling to the console application, give my final presentation, and prepare this document.

# Results

## Implementation

My Implementation is organized into five man packages of source code. The first and most important package is the [blockcipher](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/blockcipher) package. It contains the [SkipJackCipher](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/blockcipher/SkipJackCipher.java), the [OutputFeedback](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/blockcipher/OutputFeedback.java), and the [HexEncoder](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/blockcipher/HexEncoder.java) class. These objects contain the source for single block encryption and decryption with the Skipjack algorithm, multiple block encryption and decryption, IV generation, key scheduling, and hex encoding and decoding. The [SkipJackCipher](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/blockcipher/SkipJackCipher.java) class uses three primitive data types to hold the internal state of the block being encrypted or decrypted. Longs are used for two purposes; firstly they hold the full 64-bit internal state block and secondly they are used to hold each individual word (one through four) in the rule functions. Integers are only necessary for accessing the index of the F-Table in the permutation function and the round key. They are casted from longs to perform the get and then casted back to longs before being returned as word that makes up the internal state. The last type is the integer array, which is used for holding the round keys and the F-Table. The [HexEncoder](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/blockcipher/HexEncoder.java) object contains only two functions one for encoding and the other for decoding to and from input strings. This object along with the [OutputFeedback](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/blockcipher/OutputFeedback.java) class utilizes Google’s primitive [UnsignedLong](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/primitives/UnsignedLong.html) data type for storing the blocks of data.

The second package is the [tests](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/tests) package. It comprises of JUnit test cases for encryption and decryption using Output Feedback and the basic cipher. They are intended to test the functionality of the mode and the cipher and were also used for debugging purposes. The JUnit test cases also provide a short snippet of code for utilizing this project’s release as a library.

The third package is the [utils](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/utils) package. It contains general Java utility classes and functions from previous assignments that have been repurposed for this project. The [Utilities](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/utils/Utilities.java) class contains a series of cryptography related functions and type converters. The [FileReader](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/utils/FileReader.java) and [FileComposer](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/utils/FileComposer.java) are designed to read and write data to and from a given file while the [DirectoryManager](https://github.com/venom889/COMP4109-SkipJackProject/blob/master/src/com/comp4109/skipjack/utils/DirectoryManager.java) is used for setting the current working directory and file path. The file related utilities are only used for the demonstration application described below.

The last two packages ([application](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/application) and [demo](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/demo)) contain a demonstration application that uses the [blockcipher](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/blockcipher) and [utils](https://github.com/venom889/COMP4109-SkipJackProject/tree/master/src/com/comp4109/skipjack/utils) package to display the results of my implementation. The source in both these folders uses the Java System out calls to display a series of menus with options. The System in calls are then used to read user input via the console and invoke an event handler. The user has the option to encrypt from either the console or a file found in a directory that gets created during startup. The application also allows the user to set the key and the IV of the cipher.

## Known Issues

SkipJackCipher cipher = new SkipJackCipher();

OutputFeedback OFB = new OutputFeedback(cipher);

HexEncoder encoder = new HexEncoder();

OFB.generateRandomIV();

OFB.setKey("0123456789");

String pt = "I would also like to encrypt this message.";

String ct = "1BTÂÙ?L]34rÀ¹ÁõòÉ^ÃÚî%¬wD¼«ìÐ’\* ;:Ø+T\"NR~ê"

// Encrypting

UnsignedLong[] encoded = encoder.encodeMessage(pt);

UnsignedLong[] cipherblocks = OFB.encrypt(encoded);

String ciphertext = encoder.decodeMessage(cipherblocks);

// Decrypting

UnsignedLong[] encoded2 = encoder.encodeMessage(ct);

UnsignedLong[] plaintextblocks = OFB.decrypt(encoded2);

String plaintext = encoder.decodeMessage(plaintextblocks);

Figure 6 – Example code snippet

While working on my implementation I discovered a number of issues with the language I chose to implement the cipher. Java as a programming language unfortunately is quite painful for use for bitwise operations on unsigned data types. Part of Java’s design and use of automatic memory management disallows the support of C/C++ style pointer arithmetic and unsigned data types. Bytes, shorts, integers, and longs are all by default signed values meaning they can always be negative or positive and have a fixed decimal number range. This causes problems with my implementation because it requires me to use a large data type (a long rather than an integer) to hold each block. Storing negative values in the data types changes the output of the cipher from its initial design. The NSA documents I based my implementation on specifically stated that the data is unsigned.

In my initial implementation of the Skipjack cipher I was able to use bit masks on each word and block to keep the data in the positive range. However I had major problems once I began programming the cipher modes of operation. Output Feedback mode requires the operations to be done on a larger data type than a long. I tried a number of solutions in order to keep the data unsigned however they tampered with the output of the cipher and made decryption impossible. The closet solution I could work with was to use the [UnsignedLong](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/primitives/UnsignedLong.html) class that comes included with the Google Guava Libraries. This class allowed me to continually enforce unsigned data types on each block inside the modes of operation. This enforcement however came with a cost on OFB. The last ASCII character in every string is missing from the encrypted string. This means that when the cipher text is decrypted the original message will be one character or 8-bits (1 byte) short. As a temporary solution I added an extra character to every message about to be encrypted.

I also encountered an issue with encoding and decoding cipher text. The encoding and decoding process only works in one direction. By this I mean that my implementation can encode plain ASCII text, encrypt it, and decode it to cipher text; but it cannot take the same ASCII cipher text encode it and then decrypt it. The decryption process will usually fail or output the more cryptic text. I was able to prove that this was an issue with the hexadecimal encoding in the [JUnit test case for Output Feedback mode](https://raw.githubusercontent.com/venom889/COMP4109-SkipJackProject/master/src/com/comp4109/skipjack/tests/OFBSkipJackTests.java). I was able to pass the cipher text blocks in the form of [UnsignedLongs](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/primitives/UnsignedLong.html) to the decryption function and decode the result to get the same plaintext that I entered into the encryption function to generate the cipher text blocks. This confirms that the issue is in fact caused somewhere in my encoding and decoding process.

## Cryptanalysis

Because the Skipjack cipher was designed after DES it was thought by many to be more secure. The techniques used in the algorithm were quite unique and they were intensively studied by experts in abstract algebra and combinatorics years before they were used in the design of the cipher. However once the algorithm was declassified in 1998 many people in the cryptography community criticized the algorithm for its simple key schedule. Shortly after its release researchers began to find holes in its design that allowed it to be attacked. A number of theoretical attacks have been published since then, that prove the cipher can be attacked slightly faster than an exhaustive search. The chart in Figure 7 (on the next page) illustrates the number of attacks that have been found to date with the cipher. It should be noted that none of these attacks work on every round in the cipher. The highest found can speed up an attack up to round 31 before the attacker would have to result to using a brute force attack on the last round. I will not explore these attacks in any more detail in this document because they can be quite long and could be a project on their own, however if you wish to explore them further I have included the published report in my sources.

By today’s cryptographic standards the Skipjack cipher is considered unsafe due to it’s small key size. An 80-bit key is below the recommended key length and because of its simple key schedule the short key size poses an even larger threat to Skipjack than to other ciphers. However it has been proposed by some experts that the cipher is more vulnerable to non-related key attacks than to key related attacks. Regardless, the National Institute of Standards and Technology (NIST) has deemed the Skipjack algorithm insecure for agencies as of 2010. They claim it only provides temporary security and should be avoided in favor of a more modern and secure cipher. The recommended bit lengths of cryptographic keys today should be 128-bits to 256-bits based on a variety of sources and algorithms. Since the Skipjack cipher falls short of that length it is considered a legacy algorithm. I could find no information on if using the Skipjack cipher multiple times provided any more security.

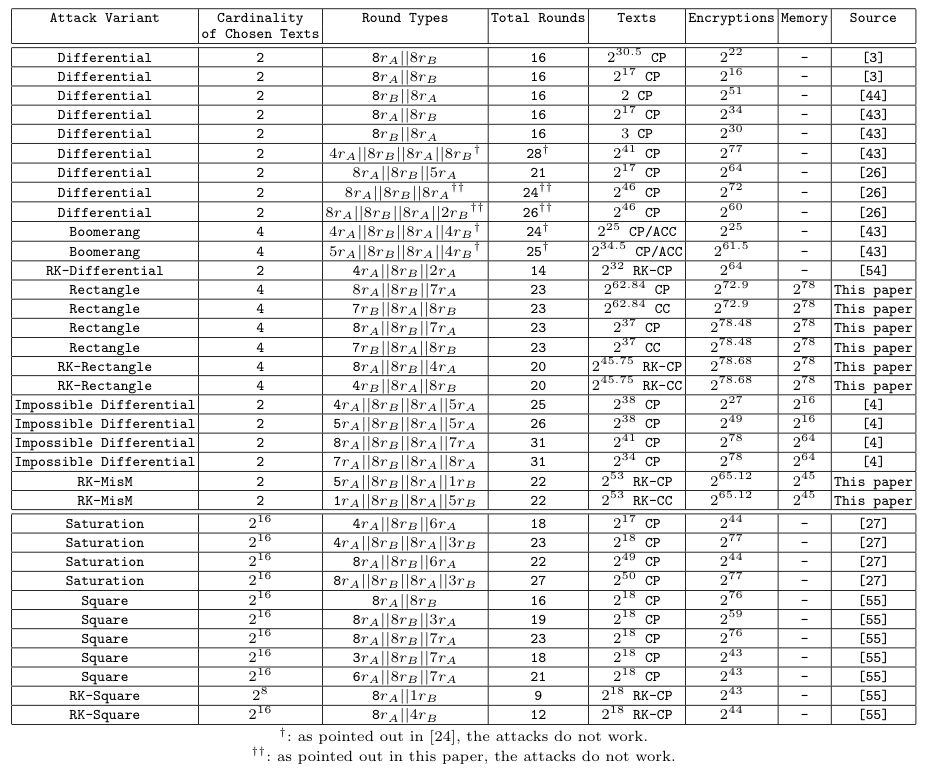


Figure 7 – Known attacks on the Skipjack cipher

# Conclusion

Having completed my implementation of the Skipjack cipher I found that the cipher holds a lot of secrets. Much of what I learned from the cipher came from its initial release document by the NSA in 1998. For example the NSA has still not released its design process behind the F-Table. The magic numbers were given with the documentation and never explained. Because of this the cipher itself sits mostly forgotten in today’s world of cryptography. It was obviously used extensively by the US government throughout the 90s but by the time it was released publically it had little to no time to grow in the commercial sector as the Advanced Encryption Standard (AES) was published the same year. Its short life span can also be attributed to the time period in which it was used. The computing power of machines in the late 90s and early 20s grew at a very rapid rate and the Skipjack algorithm did not have the scalability to keep up. It was designed for a very specific purpose and to be used on much smaller devices.

I personally, found a lot of difficulty in implementing the cipher in a more modern language than it would have been originally written in. Java proved to more of a burden than an asset in my implementation. The cipher was simple enough to write but required unsigned data types to work correctly. Next time I hope to research my language and cipher more closely to find a better pairing or approach to implementing a cryptography solution.

Even though the cipher is now considered to be one of the past, it still gave me a unique topic to explore because of its unbalanced Feistel structure and design secrecy. Overall the Skipjack cipher proved to be an interesting area of study as it gave me a better look at one of the uncommon encryption algorithms.

# References

NSA, May 29th, 1998. Skipjack and KEA Algorithm Specifications. pp. 1-18.

KIM, J. and PHAN, R.C.-W., 2009. Advanced differential-style cryptanalysis of the NSA’s skipjack block cipher. Cryptologia, 33(3), pp. 246- 270.

Barker, E. and Roginsky, A., January 2011. Transitions: Recommendation for Transitioning the Use of Cyrptographic Algorithms and Key Lengths. pp. 3-4.

Denning, D. E.; Brickell, E. F.; Kent, S. T.; Maher, D. P.; Tuchman, W., July 28, 1993 SKIPJACK Review Interim Report The SKIPJACK Algorithm.

Damien Giry, November 11th, 2014. BlueKrypt: Cryptographic Key Length Recommendation. Accessed on: November 29th, 2014. http://www.keylength.com/en/