Winter 1-4-2021

Analyzing and Creating Playing Card Cryptosystems

Isaac A. Reiter
Kutztown University of Pennsylvania, ireit426@live.kutztown.edu

Analyzing and Creating Playing Card Cryptosystems

Isaac Reiter

Department of Mathematics

Kutztown University of Pennsylvania

2020

Advisor: Dr. Eric Landquist

Abstract

Before computers, military tacticians and government agents had to rely on pencil-and-paper methods to encrypt information. For agents that want to use low-tech options in order to minimize their digital footprint, non-computerized ciphers are an essential component of their toolbox. Still, the presence of computers limits the pool of effective hand ciphers. If a cipher is not unpredictable enough, then a computer will easily be able to break it.

There are $52! \approx 2^{225.58}$ ways to mix a deck of cards. If each deck order is a key, this means that there are $52! \approx 2^{225.58}$ different ways to encrypt a given message. To create some perspective, most computer ciphers feature either 2^{128} or 2^{256} different ways of encrypting the same message. Hence, a cipher created from a deck of cards has the potential to emulate the security of many computer ciphers.

Dr. Landquist and I spent the summer of 2019 examining existing playing card ciphers. This led to the main focus of this paper: the creation of a unique, secure playing card cipher. Because of the inspiration provided by the cipher VIC, I am calling our original cipher VICCard. VICCard has gone through multiple versions, each better than the last. Its security is rooted in its combination of numerous cryptographic principles, including a substitution checkerboard, columnar transpositions, lagged Fibonacci generators, and junk letters. As evidenced by certain randomness tests, VICCard has the potential to extensively randomize an English plaintext.

Contents

Acknowledgements

I gratefully acknowledge the support of the Honors Program of Kutztown University of Pennsylvania, for which this paper is intended. Also, I am very grateful for the opportunities provided by the KUBEARS grant (Kutztown University Bringing Experience About Research to Students). Much of the preliminary work for this research was completed under this grant in the summer of 2019. Most importantly, I would like to thank my research advisor, Dr. Eric Landquist. I am remarkably fortunate to have had Dr. Landquist as my professor and mentor throughout my undergraduate experience.

1 Introduction

1.1 Cryptography's Journey from Painting to PC

Cryptology is the science of safe, secure communication. It examines how to transform a message (called the plaintext) into an encoded form (called the ciphertext). An effective cryptologist must be proficient in two tasks: cryptography and cryptanalysis. Cryptography is the study of creating effective ciphers. Cryptanalysis is the study of breaking these ciphers. Cryptography is "code writing", and cryptanalysis is "code breaking." The creator of a secure cipher uses both of these skills. He first uses cryptography to create his cipher, and he then uses cryptanalysis to see whether his cipher is as secure as he thinks.

Egyptian hieroglyphics are one of the oldest instances of cryptography. For example, the tomb of Khnumhotep II featured a myriad of pictures and symbols. These pictographs told the story of the deceased with a beautiful visual display [11]. By contrast, cryptography is more frequently used for less aesthetic purposes: war and espionage.

Just as weapons of war have become more refined, cryptography has undergone careful attention and development. Humble ciphers such as Julius Caesar's cipher and the Vigenere cipher have given way to more advanced creations such as Rasterschlüssel 44 and VIC. As the ciphers became more complicated, they became more secure. However, they also became more impractical. As a result, cryptography's next step in its evolution was to enlist the help of machines. The most compelling example of this is the German Enigma machine. In order to break this cipher, the Allies enlisted cryptographers to fight fire with fire. To break the Enigma cipher they built an even better machine: a computer. If you are curious about the triumphant story of cracking the German cipher, "The Imitation Game" is a fascinating watch. Computer ciphers have now become the norm, encrypting everything from government secrets to emails between friends.

The advantage of computer ciphers is their ability to use formidable n-bit encryption. A cipher with n-bit encryption uses a pool of 2^n possible keys, meaning that there are 2^n

possible ways of encrypting any given message. Typically, computer ciphers use 128-bit or 256-bit encryption. This prevents the ciphers from being cracked through brute force attempts that test every possible key.

Although cryptography has become mechanized since WWII, cryptographers have not discounted the strength and security of ciphers that are executed by hand. In fact, computerized ciphers can be based on the general cryptographic principles found in hand ciphers.

1.2 Why Playing Cards?

Here is an important observation. There are 52! ways to permute a deck of cards. This means that there are $52 \times 51 \times 50 \times 49 \times \cdots \times 3 \times 2 \times 1$ ways to arrange a deck of cards. To give you an idea of the scope of this number, consider the following scenario that was adapted from a quote of Stephen Fry. Imagine a trillion universes, each of which contains a trillion planets. Each of these planets contains a trillion people, and each person has a trillion decks of cards. If everyone can shuffle all of their decks one time per second, it would take over two and a half trillion years before every possible deck order has been created [5]. Simply put, Fermilab estimates that there are approximately anywhere from 10^{49} to 10^{50} atoms that make up the earth [4]. This means that there are more ways to shuffle a deck of cards than there are atoms that compose the earth. Since $52! \approx 2^{225.58}$, a deck of cards has the potential to provide 225.58-bit encryption. This is enough to compete with the security provided by typical computer ciphers. From this arises the following question: can we use playing cards to create a secure, efficient hand cipher?

1.3 Existing Playing Card Ciphers

Given that the field of playing card ciphers is remarkably specialized, not a lot of playing card ciphers have been created. Aaron Toponce has a great website that lists most if not all of the publicly known playing card ciphers [19]. Before creating my own cipher, it was important to look at the work that has already been done. Performing cryptanalysis on existing ciphers can help determine both the strengths and weaknesses that tend to occur in playing cards ciphers. With this knowledge, one is better equipped to maximize the former and minimize the latter. Two playing card ciphers in particular are of interest.

First, Card-Chameleon is a playing card cipher created by Matthew McKague for his master's thesis [10]. His intention was to create a hand version of the computer algorithm RC4. At first glance, Card-Chameleon's straightforward, easy to remember algorithm makes it attractive. However, scrutiny of this cipher revealed a fatal weakness. Assuming a random key for each letter, Card-Chameleon encrypts any given letter into the exact same letter with probability $\frac{1}{13}$. Here's why this is a weakness. For each plaintext letter, the encryption algorithm should be such that every letter has the same probability of occurring. In other words, a plaintext letter should have a $\frac{1}{26}$ probability of encrypting to any other letter. With Card-Chameleon, however, it is disproportionately likely that a letter will encrypt to itself. Unfortunately, this deviation from the magic $\frac{1}{26}$ probability is too significant to overlook [1].

Second, Chaocipher is a cryptosystem that was created by John F. Byrne in 1918 [7]. Although Chaocipher has been around for over a century, the disclosure of the Chaocipher algorithm occurred as recently as 2010 [15]. As he was examining previously invented playing card ciphers, Toponce had the idea of adapting the Chaocipher algorithm to playing cards [18]. Given the respectable security of Chaocipher, I did not find a weakness that was as severe as that in Card-Chameleon. The closest thing to a weakness is the existence of plaintext/ciphertext pairs (or pt/ct pairs). A pt/ct pair is when two identical plaintext letters encrypt to the same ciphertext characters, such as two `a`'s encrypting to two `o`'s. Greg Mellen noticed that when he divided messages encrypted by Chaocipher into blocks of 13 letters, pt/ct pairs rarely occurred within these blocks [14]. Moshe Rubin hypothesized that pt/ct pairs will only occur if the two plaintext letters are separated by a distance of eight letters [14]. In order to put a rest to this question, I wrote a program that took two `a`'s

[1]For a description of how Card-Chameleon works and the proof of this weakness, see Appendix A.

and tried every 1-letter, 2-letter, 3-letter, 4-letter, and 5-letter combination between these two a's. After testing all 12,356,630 of these cases, the program did not find any pt/ct pairs. However, it did find pt/ct pairs with certain 6-letter combinations. As a result, we can say for certain that at least six letters must be between two plaintext characters for a pt/ct pair to occur.

1.4 The Current Approach

Analyzing existing ciphers revealed a general trend among them. Most if not all playing card ciphers are stream ciphers. This means that they encrypt plaintexts one letter at a time. The typical strategy is to first encrypt a letter and then alter the deck order before encrypting the next letter. In creating a unique cipher, I used a different approach. I focused my efforts on creating a block cipher. With a block cipher, the plaintext is encrypted in blocks of letters. Specifically with our cipher, we are encrypting the entire message at once in one large block.

2 VICCard

2.1 The Hollow Nickel Case

In 1953, Jimmy Bozart was a young 13-year-old boy living in Brooklyn. He delivered newspapers for the Brooklyn Eagle. On June 22, he was counting his tips when he noticed that one of his nickels was lighter than the others. As the nickel slipped from his fingers, it hit the floor and cracked neatly into two pieces. Inside, the nickel was completely hollow. Furthermore, it contained a tiny piece of microfilm with numbers on it [3].

When local police officers heard of this discovery, they scrambled to track down Jimmy and his nickel. Just in case he carelessly spent his valuable discovery, they examined the Bingo money from the church and ice cream money from a Good Humor vendor. They eventually found Jimmy, who willingly gave them the nickel. Realizing the potential gravity of what they possessed, the New York police officers turned the coin over to the FBI [2].

In their research, the FBI investigators looked into whether the coin was simply a trick nickel meant for gags or magic tricks. This theory failed due to the imprecision with which the nickel was made. The hollow part was not big enough to contain much. Being a magician myself, I have handled high-quality hollow coins. The craftsman has to balance two factors. First, they have to make sure that the hollow coin is not too big. Otherwise, it will excite suspicion from the audience. On the other hand, the coin cannot be too small. If it is, anything that the magician is trying to hide inside the coin can easily get stuck. No feeling is worse than realizing mid-performance that your props are not cooperating. Jimmy's hollow nickel was not crafted with this much precision [3].

The FBI had to solve two questions: what was the coin's purpose, and what was the meaning of the numbers on the microfilm? In an exceptional stroke of luck, both questions were answered by Russian spy Reino Häyhänen. Häyhänen did not begin his espionage career by choice. Because he was fluent in Finnish, he was drafted as a translator for the Communist secret police during the Finnish-Soviet war. Upon the end of the war, he remained in Finland in order to report anti-Soviet individuals. Häyhänen became a member of the Communist party in 1943, and in 1948 the KGB assigned him a new task. Assuming the identity of Eugene Nicolai Maki, an immigrant from America to Estonia, he was to act as a Soviet spy in the United States. In 1957, Häyhänen contacted the U.S. embassy in Paris, desiring to defect to the United States. Following his defection, Häyhänen gave FBI officials the details of his operations. Most importantly, he revealed how hollow coins, such as the one found by Jimmy, were used to exchange information. Soviet agents agreed on inconspicuous locations called "dead drops" in which they placed secret containers such as hollow coins [3].

The only remaining piece of the puzzle was to decrypt the message that was inside the coin. Häyhänen thoroughly explained the cipher that was used to encrypt the message on the microfilm inside the nickel [3]. It was encrypted using a Nihilist cipher called VIC [20].

The Nihilists were a Russian group that opposed the Russian tsar. In the 1880s, the Nihilists used ciphers in order to communicate, which became known as the Nihilist ciphers [6]. In a bizarre twist of fate, the message in the nickel was actually intended for Häyhänen himself. After he accidentally spent the nickel, it traveled from person to person until it eventually landed into Jimmy's inquisitive hands [2].

The CIA has an excellent description of how VIC works [8]. Häyhänen presented this description at the 1957 trial of Colonel Rudolf Abel. Since the CIA states that those attending the trial were either bored or confused by the description of VIC, I will spare you the details. Instead, I will describe the specific aspects of VIC that provided the inspiration for VICCard. If you are interested in more details about Cold War espionage and the Rudolf Abel trial in particular, definitely watch "Bridge of Spies." Not only is it certified fresh by Rotten Tomatoes, it is also directed by Steven Spielberg and features Tom Hanks as the lead [17].

2.2 The VICCard Cipher

To pay homage to the mind-numbing security of VIC, I have decided to entitle my original cipher VICCard. VICCard is an original playing card cipher that combines numerous cryptographic strategies together. The basic strategy of VIC is to use a checkerboard to convert letters to cards and then to perform various operations on these cards. With VICCard, we are using a similar strategy. There are four steps to this cipher. First, use a checkerboard to convert the letters of the message into cards. Second, perform columnar transpositions on these cards. Third, apply lagged Fibonacci generators to these cards. Finally, use the same checkerboard to convert the cards back into letters. Although VICCard has gone through multiple versions, these four steps remained the fundamental structure. In order to demonstrate each of these steps, we will follow cryptographic tradition and encrypt the message `Attack At Dawn` as an example. The following deck of cards will be our key. In our notation, $9\diamondsuit$ is the top card and $A\heartsuit$ is the bottom card when the deck is held face up.

[$A\heartsuit$, $7\heartsuit$, $K\clubsuit$, $8\heartsuit$, $K\diamondsuit$, $J\clubsuit$, $K\spadesuit$, $K\heartsuit$, $4\spadesuit$, $8\diamondsuit$, $4\heartsuit$, $7\clubsuit$, $3\clubsuit$, $10\diamondsuit$, $Q\heartsuit$, $10\clubsuit$, $5\diamondsuit$, $2\spadesuit$, $J\spadesuit$, $A\clubsuit$, $9\clubsuit$, $4\clubsuit$, $3\diamondsuit$, $3\heartsuit$, $8\clubsuit$, $7\diamondsuit$, $5\spadesuit$, $5\heartsuit$, $2\clubsuit$, $A\diamondsuit$, $8\spadesuit$, $10\spadesuit$, $6\heartsuit$, $9\spadesuit$, $10\heartsuit$, $6\diamondsuit$, $Q\diamondsuit$, $6\clubsuit$, $2\diamondsuit$, $J\diamondsuit$, $7\spadesuit$, $5\clubsuit$, $4\diamondsuit$, $J\heartsuit$, $Q\spadesuit$, $6\spadesuit$, $3\spadesuit$, $Q\clubsuit$, $9\heartsuit$, $2\heartsuit$, $A\spadesuit$, $9\diamondsuit$]

2.3 Step 1: Converting Letters to Cards

We will have cards represent letters according to the following table. Notice that the lowercase letters are represented by the black cards and that the uppercase letters are represented by the red cards.

	Spades ($\spadesuit$)												
Card	A	2	3	4	5	6	7	8	9	10	J	Q	K
Letter	`a`	`b`	`c`	`d`	`e`	`f`	`g`	`h`	`i`	`j`	`k`	`l`	`m`
	Clubs ($\clubsuit$)												
Card	A	2	3	4	5	6	7	8	9	10	J	Q	K
Letter	`n`	`o`	`p`	`q`	`r`	`s`	`t`	`u`	`v`	`w`	`x`	`y`	`z`
	Hearts ($\heartsuit$)												
Card	A	2	3	4	5	6	7	8	9	10	J	Q	K
Letter	`A`	`B`	`C`	`D`	`E`	`F`	`G`	`H`	`I`	`J`	`K`	`L`	`M`
	Diamonds ($\diamondsuit$)												
Card	A	2	3	4	5	6	7	8	9	10	J	Q	K
Letter	`N`	`O`	`P`	`Q`	`R`	`S`	`T`	`U`	`V`	`W`	`X`	`Y`	`Z`

Table 1: Letter encoding for VICCard

In order to convert the letters of the plaintext into cards, we will use a checkerboard. The checkerboard in Table 2 is created by dealing the cards into 4 columns of 13 cards.

Suppose that we are encrypting the plaintext `Attack At Dawn`. We will convert these letters to cards one letter at a time. We will start with the letter `A`, which is represented by $A\heartsuit$. First, I find $A\heartsuit$ in the checkerboard. This card is in the **A** row and the $\clubsuit$ column. Hence, `A` encrypts to $A\clubsuit$. Next, we move onto the letter `t`, which is represented by $7\clubsuit$. This card is in the **Q** row and the $\clubsuit$ column. Hence, `t` encrypts to $Q\clubsuit$. Continuing this pattern, the plaintext `Attack At Dawn` is converted to $A\clubsuit, Q\clubsuit, Q\clubsuit, Q\diamondsuit, 8\diamondsuit, 6\heartsuit, A\clubsuit, Q\clubsuit, J\clubsuit, Q\diamondsuit, 3\heartsuit, 7\heartsuit$.

Typically with effective cryptographic checkerboards, each plaintext letter is represented by 2 or more numbers. Instead of using this string of cards, we will break up the cards into

	$\clubsuit$ (0)	$\heartsuit$ (1)	$\spadesuit$ (2)	$\diamondsuit$ (3)
A (1)	$A\heartsuit$	$10\diamondsuit$	$5\spadesuit$	$J\diamondsuit$
2 (2)	$7\heartsuit$	$Q\heartsuit$	$5\heartsuit$	$7\spadesuit$
3 (3)	$K\clubsuit$	$10\clubsuit$	$2\clubsuit$	$5\clubsuit$
4 (4)	$8\heartsuit$	$5\diamondsuit$	$A\diamondsuit$	$4\diamondsuit$
5 (5)	$K\diamondsuit$	$2\spadesuit$	$8\spadesuit$	$J\heartsuit$
6 (6)	$J\clubsuit$	$J\spadesuit$	$10\spadesuit$	$Q\spadesuit$
7 (7)	$K\spadesuit$	$A\clubsuit$	$6\heartsuit$	$6\spadesuit$
8 (8)	$K\heartsuit$	$9\clubsuit$	$9\spadesuit$	$3\spadesuit$
9 (9)	$4\spadesuit$	$4\clubsuit$	$10\heartsuit$	$Q\clubsuit$
10 (10)	$8\diamondsuit$	$3\diamondsuit$	$6\diamondsuit$	$9\heartsuit$
J (11)	$4\heartsuit$	$3\heartsuit$	$Q\diamondsuit$	$2\heartsuit$
Q (12)	$7\clubsuit$	$8\clubsuit$	$6\clubsuit$	$A\spadesuit$
K (0)	$3\clubsuit$	$7\diamondsuit$	$2\diamondsuit$	$9\diamondsuit$

Table 2: Checkerboard from Deck Order

two rows. The first row is all of the face values of the cards, and the second row is all of the suits. The face values are represented with the numbers 0 through 12, and the suits are represented with the numbers 0 through 3.

`A`	`t`	`t`	`a`	`c`	`k`	`A`	`t`	`D`	`a`	`w`	`n`
$A\clubsuit$	$Q\clubsuit$	$Q\clubsuit$	$Q\diamondsuit$	$8\diamondsuit$	$6\heartsuit$	$A\clubsuit$	$Q\clubsuit$	$J\clubsuit$	$Q\diamondsuit$	$3\heartsuit$	$7\heartsuit$
1	12	12	12	8	6	1	12	11	12	3	7
0	0	0	3	3	1	0	0	0	3	1	1

Table 3: Letters Converted to Face Values and Suits

At this point, it is important to take notice of a security feature of this checkerboard. Notice that every letter is represented by a black card and a red card. The black card represents the lowercase version, and the red card represents the uppercase version. In the above example, I used the black cards to represent each lowercase letter and the red cards to represent each uppercase letter. However, I did not have to do this. For each plaintext letter, we can either use the black card or the red card to encrypt it. For example, instead of using $A\heartsuit$ for the letter `A`, we can instead use the black card option $A\spadesuit$. This is equivalent to regularly encrypting the letter `a`. Similarly, instead of encrypting `t` as $Q\clubsuit$, we can encrypt it as $Q\diamondsuit$. This is exactly like regularly encrypting the letter `T`. In other words, the option of choosing either the black card or the red card for each letter is equivalent to the option of changing the case of each letter. For example, based on how we choose black and red cards, we can encrypt the message `Attack At Dawn` as `aTtacK AT daWN` . When the decoder reverses this process, he will get the latter plaintext message. The letters are in the wrong cases, but it is still readable. Hence, having this choice for each letter does not compromise the message.

This feature has great potential for increasing security. Suppose that we have a plaintext of N letters. Since there are two choices for each letter, a black card or a red card, the encoder has 2^N possible ways of using the same deck to encrypt a particular message. Recall that there are about $2^{225.58}$ possible decks. Combining these two together, there are $2^{225.58+N}$ possible ways of encrypting the same message. In other words, every letter in the plaintext adds a bit to the pool of possible keys.

2.4 Step 2: Columnar Transpositions

In Step 1, we performed a substitution: cards were substituted for letters. In Step 2, we will perform a transposition. Here, none of the numbers are going to be altered. Instead, they are going to be rearranged via a columnar transposition. We will use two columnar transpositions: one for the row of face values and one for the row of suits. Here is how we perform columnar transpositions. First, we need a key for each transposition. We will use the order of the clubs for the face values: [K♣, J♣, 7♣, 3♣, 10♣, A♣, 9♣, 4♣, 8♣, 2♣, 6♣, 5♣, Q♣]. Also, we will use the order of the hearts excluding the king for the suits: [A♡, 7♡, 8♡, 4♡, Q♡, 3♡, 5♡, 6♡, 10♡, J♡, 9♡, 2♡]. To transpose the face values, we create a grid with the clubs on top. Then, we fill in the grid from left to right with the face values.

K♣	J♣	7♣	3♣	10♣	A♣	9♣	4♣	8♣	2♣	6♣	5♣	Q♣
1	12	12	12	8	6	1	12	11	12	3	7	

Next, we read the face values out of the grid from top to bottom based on the numerical order of the clubs. We first read 6 from the A♣ column, 12 from the 2♣ column, 12 from the 3♣ column, and so forth to get the following new row of face values: (6 12 12 12 7 3 12

11 1 8 12 1). Similarly, to perform the transposition of the suits we create a grid with the hearts on top. Then, we again fill in the grid from left to right.

$A\heartsuit$	$7\heartsuit$	$8\heartsuit$	$4\heartsuit$	$Q\heartsuit$	$3\heartsuit$	$5\heartsuit$	$6\heartsuit$	$10\heartsuit$	$J\heartsuit$	$9\heartsuit$	$2\heartsuit$
0	0	0	3	3	1	0	0	0	3	1	1

Reading out the suits from the top to bottom based on the numerical order of the suits, we get the following new row of suits: (0 1 1 3 0 0 0 0 1 0 3 3). In summary, performing these columnar transpositions gives us the following two new rows of face values and suits:

6	12	12	12	7	3	12	11	1	8	12	1
0	1	1	3	0	0	0	0	1	0	3	3

2.5 Step 3: Lagged Fibonacci Generators

The third step of VICCard is to use two lagged Fibonacci generators. Frequently in cryptography, we employ the help of random strings of numbers. However, the problem with using humongous strings of random numbers is that they are wildly impractical. Instead, it is more common to use pseudorandom strings of numbers. These are strings of numbers that appear to be random and have a lot of the properties of randomness, even though they were not created in a purely random way.

A lagged Fibonacci generator is one such method of creating a pseudorandom string of numbers. Instead of sharing the entire string of numbers, the sender and receiver only share a small string of a few digits. This is called the seed. For example, suppose that we are using the first five digits of π as the seed: (3 1 4 1 5). Here is the procedure for creating an indefinitely long string of numbers by using a lagged Fibonacci generator. We begin with the first two numbers: 3 and 1. We add these together to get 4, and we attach this number to the end of the seed: (3 1 4 1 5 4). Next, we move onto the next two numbers: 1 and 4. Adding these together gives us 5, which we attach to the end of the previous string of numbers: (3 1 4 1 5 4 5). Again, we add the next two numbers (4 and 1) to get 5, which is again attached to the end: (3 1 4 1 5 4 5 5). Continuing this process indefinitely, we can create a pseudorandom string of numbers of any desired length. Also, it is important to note that this addition is performed modulo 10. This means that if adding two numbers produces a number that is greater than 10, we divide this number by ten and use the remainder. For example, adding 5 and 7 gives us 12, which is 2 in modulo 10.

The convenience of a lagged Fibonacci generator is rooted in the fact that the sender and receiver only need to share the seed. In order to do this with VICCard, we will encode two seeds in the deck. The seed for the lagged Fibonacci generator of the face values is encoded in the order of the spades in the deck. The order of the spades in the current keyed deck is [$K\spadesuit$, $4\spadesuit$, $2\spadesuit$, $J\spadesuit$, $5\spadesuit$, $8\spadesuit$, $10\spadesuit$, $9\spadesuit$, $7\spadesuit$, $Q\spadesuit$, $6\spadesuit$, $3\spadesuit$, $A\spadesuit$]. The order of the face values of these cards yields the following seed: (13 4 2 11 5 8 10 9 7 12 6 3 1). As one more adjustment, we will represent the 13, which came from $K\spadesuit$, as 13 mod 13 = 0. Hence, the seed for the lagged Fibonacci generator of the face values is (0 4 2 11 5 8 10 9 7 12 6 3 1).

Similarly, the seed for the lagged Fibonacci generator of the suits is encoded in the order of the face values of the diamonds. The order of the diamonds in the current keyed deck is [$K\diamondsuit$, $8\diamondsuit$, $10\diamondsuit$, $5\diamondsuit$, $3\diamondsuit$, $7\diamondsuit$, $A\diamondsuit$, $6\diamondsuit$, $Q\diamondsuit$, $2\diamondsuit$, $J\diamondsuit$, $4\diamondsuit$, $9\diamondsuit$]. This gives us the following seed: (13 8 10 5 3 7 1 6 12 2 11 4 9). Since there are only four suits in a deck, we will express this seed in modulo 4. Hence, the seed for the lagged Fibonacci generator of the suits is (1 0 2 1 3 3 1 2 0 2 3 0 1).

We will now add these two string of numbers to the rows of face values and suits using modulo 13 and modulo 4 arithmetic, respectively. In this case, the plaintext is small enough so that we do not have to generate any more numbers. If the plaintext were longer, we would use each seed to create new numbers as detailed above. Adding the numbers from the spade lagged Fibonacci generator to the row of face values, we get the following:

	6	12	12	12	7	3	12	11	1	8	12	1
+	0	4	2	11	5	8	10	9	7	12	6	3
=	6	3	1	10	12	11	9	7	8	7	5	4

Adding the numbers from the diamond lagged Fibonacci generator to the row of suits, we get the following:

	0	1	1	3	0	0	0	0	1	0	3	3
+	1	0	2	1	3	3	1	2	0	2	3	0
=	1	1	3	0	3	3	1	2	1	2	2	3

This gives us the following two new rows of face values and suits:

6	3	1	10	12	11	9	7	8	7	5	4
1	1	3	0	3	3	1	2	1	2	2	3

2.6 Step 4: Converting Cards Back into Letters

The final step in the VICCard cipher is to convert these two rows of numbers back into letters. In order to do this, we will use Table 2 and reverse the algorithm of Step 1. We will start with the first column of numbers, which contains a face value of 6 and a suit of 1 (♡). This tells us to look at the card in the sixth row and the ♡'s column of the checkerboard, which is J♠. Since J♠ represents the letter `k`, the first letter of the ciphertext is `k`. Next, the second column of numbers tells us to look at the card in the third row and the ♡'s column, which is 10♣. This card represents the letter `w`, meaning that the next letter of the ciphertext is `w`. Continuing this pattern, we get the following ciphertext: `kwXUaB qF vFhQ`.

2.7 Summary: The Cards as a Key Container

Something that you might have noticed about VICCard is that playing cards are technically not needed to perform it. Using substitution checkerboards, columnar transpositions, and lagged Fibonacci generators are not unique to playing cards. In fact, this cipher can be entirely executed using numbers instead of face values and suits. However, suppose that we do not use playing cards to execute VICCard. Here, the one sending the message and the one receiving the message must share a remarkable amount of information. They must both know the order of letters in the checkerboard, the keywords used for the transpositions, and the seeds for the lagged Fibonacci generators. The reason for executing this cipher with playing cards is because all of this information is compactly contained in 52 playing cards. This way, the sender and receiver must only share the deck order.

Now that we have used cryptography to create VICCard, the next step is to use cryptanalysis to analyze its security. We will examine each element of VICCard: the substitution checkerboard, the columnar transpositions, and the lagged Fibonacci generators.

3 Substitution Checkerboard

Basic cryptographic substitutions can be found in cryptogram puzzle books. In these books, every English letter is represented by another letter. For example, every `e` is replaced with `w` and every `x` is replaced by `c`. There are $26! \approx 4.03 \times 10^{26}$ possible ways to substitute each English letter for another English letter. At first glance, it seems that a standard cryptogram is remarkably secure. However, if that were true, cryptogram books would not be available in giant puzzle books alongside Sudoku and crosswords. The insecurity of cryptograms is best exemplified by two common cryptanalysis methods. First, analyzing the frequency distribution of English letters is a useful technique of decoding a cryptogram. For example, `e` is the most common English letter. Since `e` encrypts to `w` in the above example, it is likely that `w` will be the most common letter in the ciphertext. Hence, a code breaker could deduce from the high frequency of `w` that it represents `e`. A second attack is the use of plaintext cribs. Whereas the first attack exploits the frequencies of certain letters, cribs are frequently occurring words. For example, if I see a letter by itself in the ciphertext, it is likely that it represents the letter `a`. Similarly, if I see a three letter word in the ciphertext, it is likely that this is either the word `the` or a pronoun. Hence, there is a high chance that I can ascertain the identities of three letters.

Instead of a basic one-to-one substitution, a more secure method is to represent each plaintext character by two or more characters. This technique is know as fractionation, and it is present in ciphers such as Bifid, Trifid, and straddling checkerboard [21]. With Bifid, each English letter is represented by two numbers. We substitute each letter in the plaintext with two numbers and shuffle these numbers around. Then, we substitute each pair of numbers in the final sequence for their English letter equivalents. Trifid substitutes three numbers for each English letter, and a straddling checkerboard encrypts some letters with one number and some letters with two numbers.

The substitution checkerboard in VICCard follows an approach that is similar to Bifid. Each letter is substituted for two numbers; one number represents a face value, and the other number represents a suit. However, an important difference is the ability of VICCard to perform two different substitutions for each letter. We can either use the corresponding red uppercase card or the corresponding black lowercase card. With Bifid and similar substitution checkerboards, we always encrypt a letter with the same group of numbers. Here, VICCard can encrypt the exact same letter in two different ways. This is especially useful when messages use the same letter numerous times throughout the message (such as the `a`'s in `Attack At Dawn`) or when words contain two of the same letter that are next to each other (such as the two `t`'s in `Attack`). In Bifid, a code breaker has a $\frac{1}{26}$ probability of correctly guessing the letter represented by the two numbers. In VICCard, the code breaker has to guess the two cards that represent the same letter, which he has a $\frac{1}{26} \times \frac{1}{26} = \frac{1}{676}$ probability of doing correctly. This further complicates the code breaker's task without severely complicating the encryption process.

4 Columnar Transpositions

On its own, the double columnar transposition is an alluring cipher: it is easy to learn, fast to implement, and not so trivial to break. In attempting to successfully break this cipher, the first option that comes to mind is simply testing every possible keyword [9]. For a nine column transposition, there are $9! = 362,880$ possible keys. A computer can quickly move through each of these keys, easily cracking the cipher. This is why columnar transpositions are typically performed in pairs. There are $(9!)^2 = 131,681,894,400$ possible permutations with a pair of nine column transpositions. A second possible attack is a dictionary attack [9]. Here, the code breaker has a database of about 1 million frequently used keywords, such as names of prominent historical figures. He then tests each keyword to see whether it successfully decodes the message. Yet a third strategy is hill climbing [9]. This involves picking a starting keyword and gradually making small changes to this keyword, such as swapping letters. If the new keyword seems to decode the message better, then this keyword replaces the starting one. With hill climbing, we continue to make these changes until we find the keyword.

A vital feature of all these strategies is that keywords are guessed until the ciphertext is undone in such a way that it "makes sense." This is why transpositions are frequently combined with substitutions. Consider when the letters in a plaintext message are substituted in some way for others. After this, the two columnar transpositions are performed. This complicates these common code-breaking techniques because now it is impossible for any reversal of the transpositions to "make sense." This is why VICCard combines substitutions with transpositions. In fact, VICCard uses three substitutions: initially converting letters to cards, applying lagged Fibonacci generators, and finally converting cards back into letters. Hence, the double columnar transposition in VICCard is valuable in and of itself. However, it becomes remarkably strong when combined with the other cryptographic techniques.

5 Lagged Fibonacci Generator

In assessing the security of a lagged Fibonacci generator, two features must be analyzed. First, what is the period? In other words, how many numbers does the lagged Fibonacci generator create before it starts repeating? In emulating true randomness, we do not want a string of numbers that repeats. Hence, we desire a lagged Fibonacci generator with a large period. Specifically, the security of a lagged Fibonacci generator is maximized when its period is larger than the length of the plaintext. Second, does the distribution of the numbers closely resemble the distribution produced by randomness? For example, there are four different numbers in the lagged Fibonacci generator of the suits. In a truly random string of 4 different numbers, each number occurs approximately $\frac{1}{4}$ of the time. Hence, this lagged Fibonacci generator has a random distribution if the 0's, 1's, 2's, and 3's all occur approximately $\frac{1}{4}$ of the time.

5.1 Modulo 4 Lagged Fibonacci Generator

In analyzing the security of the modulo 4 lagged Fibonacci generator, the seed of which is the order of the diamonds, I first created all the possible periods. These periods each have three 0's, three 2's, three 4's, and four 1's. Hence, there are $\binom{13}{3}\binom{10}{3}\binom{7}{3} = 1,201,200$ seeds to consider. As a result, it was necessary to write a program that created each of these seeds and placed them into text files [2]. After this, I created a program in order to ascertain the period of each seed. It did this by reading each seed in from the text files, used each seed to create a string of about 100,000 numbers, and searched the string of numbers to see when the string began to repeat. Analyzing all the seeds in this way, it became clear that there are three possible periods. 23 seeds have a period of 62, 1019 seeds have a period of 510, and the remaining 1,200,158 seeds have a period of 15,810. Overall, this is very good news: $\frac{1,200,158}{1,201,200} \approx 99.9\%$ of the seeds have a respectable period of 15,810. 15,810 characters can fill almost eight pages of a Word document in MLA format, assuming that there are no spaces. This is more than what is needed to send a typical encoded message.

Once I knew the period of each seed, I then determined the distribution of 0's, 1's, 2's, and 3's produced by each seed. I accomplished this by writing a program which used each seed

[2]The C++ code for this program is in Appendix B.

to produce a string of numbers until right before it started repeating. In other words, the program produced strings of 62 numbers from the seeds that have a period of 62, produced strings of 510 numbers from the seeds that have a period of 510, and so forth. It then went through each string of numbers and counted the number of occurrences of 0's, 1's, 2's, and 3's. On average, each seed created a distribution that very closely approximated 25% of 0's, 1's, 2's, and 3's. Table 4 shows the average number of 0's, 1's, 2's, and 3's for each period.

Seeds with a period of 62			
Number of 0's	Number of 1's	Number of 2's	Number of 3's
15.3913	16.8696	14.6087	15.1304
Seeds with a period of 510			
Number of 0's	Number of 1's	Number of 2's	Number of 3's
127.421	128.495	126.579	127.505
Seeds with a period of 15,810			
Number of 0's	Number of 1's	Number of 2's	Number of 3's
3952.47	3953.32	3951.8	3952.4

Table 4: Average Distribution for the Modulo 4 lagged Fibonacci generator

Table 5 shows the percent error of each average relative to 25% of the period. As the period increases, the percent error decreases.

Seeds with a period of 62			
Percent Error of 0's	Percent Error of 1's	Percent Error of 2's	Percent Error of 3's
-0.7%	8.8%	-5.8%	-2.4%
Seeds with a period of 510			
Percent Error of 0's	Percent Error of 1's	Percent Error of 2's	Percent Error of 3's
-0.06196%	0.7804%	-0.7224%	0.003922%
Seeds with a period of 15,810			
Percent Error of 0's	Percent Error of 1's	Percent Error of 2's	Percent Error of 3's
$-7.590 \times 10^{-4}\%$	$2.075 \times 10^{-2}\%$	$-1.771 \times 10^{-2}\%$	$-2.530 \times 10^{-3}\%$

Table 5: Percent Errors of the Distribution for the Modulo 4 lagged Fibonacci generator

5.2 Modulo 13 Lagged Fibonacci Generator

In order to analyze the security of the modulo 13 lagged Fibonacci generator, which is encoded in the order of the spades, I followed a similar process. I first examined the periods and then analyzed the distributions. Before any of this could be done, however, I had to determine which seeds to test. There are $13! = 6,227,020,800$ possible seeds, making it impractical to test all of them. Instead, I chose a random sampling of seeds to test. Specifically, I analyzed all the seeds that start with (8 4 12 5), all the seeds that start with (12 6 9 8), all the seeds that start with (1 9 12 7), all the seeds that start with (3 2 11 7), and all the seeds that start with (3 1 5 2). There are $9! = 362,880$ seeds in each of these categories, meaning that I analyzed $5 \times 362,880 = 1,814,400$ seeds. Of course, before testing any of these seeds, I had to create five text files containing each of these categories of seeds. I did this by creating a program that created every possible permutation of any number of objects. Since the seed contains 13 numbers, I used the 13 setting of my program. Furthermore, I filled in the first four numbers for each category (like 8 4 12 5). This program then created every possible permutation of the nine remaining numbers and wrote all the seeds to a text file with the appropriate label. For example, the program wrote all seeds beginning with (8 4 12 5) to a file entitled "EightFourTwelveFive.txt" [3]. Once I compiled this respectable sample size, I began testing each seed.

Recall that most of the seeds of the modulo 4 lagged Fibonacci generator have a period of 15,810. We are not as concerned with the exact periods of the modulo 13 seeds. As long as the periods of the latter are larger than those of the former, then the periods of the modulo 13 seeds are secure for our current purposes. In order to test this, I created a program that took each seed in the previously created files, used each to create a string of slightly more than 15,810 numbers, and searched the string to see if the pattern repeated. Of all the 1,814,400 seeds that were tested, none of them repeated within 15,810 numbers. Just out of curiosity, I took a random seed (8 4 12 5 0 7 9 10 2 3 1 11 6) and tried to determine its period. After creating a string of 3,000,000 numbers, the pattern of numbers still did not repeat. As a result, we can conclude that the modulo 13 lagged Fibonacci generator has a respectable, secure period.

[3]The C++ code for this program is in Appendix C.

Next, I needed to make sure that the lagged Fibonacci generator produces a uniform distribution of the numbers 0 through 12. In order to do this, I created yet another program that used each seed to create a string of 13,000 numbers. It then counted the number of occurrences of each number in each string. Tables 6 through 11 show the average results for each of the five groups of seeds and the overall average for all 1,814,400 seeds.

Seeds That Begin with 8 4 12 5	
Number	Average Number of Occurrences
0	1003.31
1	998.494
2	1003.03
3	999.091
4	1001.6
5	998.142
6	998.786
7	1001.39
8	998.25
9	998.963
10	999.007
11	1000.6
12	999.329

Table 6: Average Distribution for Modulo 13 lagged Fibonacci generator (8 4 12 5)

Seeds That Begin with 12 6 9 8	
Number	Average Number of Occurrences
0	999.686
1	999.94
2	998.537
3	1001.64
4	1000.73
5	999.861
6	1001.15
7	1000.81
8	999.764
9	1000.17
10	999.243
11	997.879
12	1000.6

Table 7: Average Distribution for Modulo 13 lagged Fibonacci generator (12 6 9 8)

Seeds That Begin with 1 9 12 7	
Number	Average Number of Occurrences
0	1002.66
1	999.659
2	999.788
3	999.473
4	1003.42
5	999.352
6	999.374
7	997.423
8	1000.33
9	998.763
10	999.384
11	1001.46
12	998.911

Table 8: Average Distribution for Modulo 13 lagged Fibonacci generator (1 9 12 7)

Seeds That Begin with 3 2 11 7	
Number	Average Number of Occurrences
0	1000.46
1	997.528
2	1000.79
3	999.103
4	999.776
5	1004.38
6	998.065
7	1001.08
8	999.167
9	1001.5
10	999.008
11	1000.48
12	998.668

Table 9: Average Distribution for Modulo 13 lagged Fibonacci generator (3 2 11 7)

If these strings of 13,000 numbers were truly random, we would expect approximately 1000 of each number. As demonstrated by these tables, the lagged Fibonacci generator in question provides an incredible approximation of this distribution. As a result, we can conclude that the modulo 13 lagged Fibonacci generator is safe to use.

Seeds That Begin with 3 1 5 2	
Number	Average Number of Occurrences
0	1000.46
1	998.955
2	1000.42
3	998.035
4	1001.65
5	998.509
6	1000.09
7	1003.48
8	1001.21
9	998.733
10	999.764
11	997.494
12	1001.19

Table 10: Average Distribution for Modulo 13 lagged Fibonacci generator (3 1 5 2)

Results for all 1,814,400 Seeds		
Number	Average Number of Occurrences	Percent Error
0	1001.3152	0.13152%
1	998.9152	-0.10848%
2	1000.513	0.0513%
3	999.4684	-0.05316%
4	1001.4352	0.14352%
5	1000.0488	0.00488%
6	999.493	-0.0507%
7	1000.8366	0.08366%
8	999.7442	-0.02558%
9	999.6258	-0.03742%
10	999.2812	-0.07188%
11	999.5826	-0.04174%
12	999.7396	-0.02604%

Table 11: Average Distribution for Modulo 13 lagged Fibonacci generator Overall

6 Updates of VICCard

The above description of VICCard is the first version of the cipher that was used to outline the basic structure. It is composed of the following basic steps:

1. Convert the plaintext letters to cards using the checkerboard.
2. Perform one columnar transposition on both the face value row and the suit row.

3. Add one pseudorandom string of numbers to both the face value row and the suit row.
4. Use the same checkerboard in Step 1 to convert the cards back into letters.

In an attempt to improve VICCard, I created a C++ program for each version of the cipher. After using these programs to examine a particular version, I created the subsequent version by making alterations to the individual steps. Hence, this basic four-step structure remains the same throughout each update.

6.1 VICCard 2.0

The second version of VICCard alters the second step. When cryptographers use columnar transpositions, they are typically performed two at a time. These double columnar transpositions provide much more security without adding too much more work. In order to perform an additional columnar transposition, we need another key word. Since we are performing double columnar transpositions on both the face value row and the suit row, four keys are required in total. It is important that the keys for the columnar transpositions (the clubs and hearts) are kept separate from the seeds for the lagged Fibonacci generators (spades and diamonds). Hence, we will take all four keys from the clubs and the hearts. For the first transposition of the face value row, we will use the order of the first 8 clubs. For the second transposition of the face value row, we will use the order of the first 6 hearts. For the first transposition of the suit row, we will use the order of the remaining 5 clubs. For the second transposition of the suit row, we will use the order of the remaining 7 hearts.

For example, consider the above encryption example of `Attack At Dawn`. After converting the letters to cards, we create the following two rows of numbers:

1	12	12	12	8	6	1	12	11	12	3	7
0	0	0	3	3	1	0	0	0	3	1	1

We will start with the face value row. First, we perform a columnar transposition based on the order of the first eight clubs.

7♣	3♣	1♣	4♣	8♣	2♣	6♣	5♣
1	12	12	12	8	6	1	12
11	12	3	7				

The result of the above transposition is (12 3 6 12 12 12 7 12 1 1 11 8). Next, we perform a columnar transposition based on the order of the first six hearts.

A♡	4♡	3♡	5♡	6♡	2♡
12	3	6	12	12	12
7	12	1	1	11	8

The result of the above transposition is (12 7 12 8 6 1 3 12 12 1 12 11). Now, we move onto permuting the row of suits. The first transposition is performed using the last five clubs.

K♣	*J*♣	10♣	9♣	*Q*♣
0	0	0	3	3
1	0	0	0	3
1	1			

The result of the above transposition is (3 0 0 0 0 0 1 3 3 0 1 1). Finally, the second permutation of the suits is accomplished using the last seven hearts.

7♡	8♡	*K*♡	*Q*♡	10♡	*J*♡	9♡
3	0	0	0	0	0	1
3	3	0	1	1		

The result of the above transposition is (3 3 0 3 1 0 1 0 0 1 0 0). Hence, the final result of the Step 2 columnar transpositions is the following:

12	7	12	8	6	1	3	12	12	1	12	11
3	3	0	3	1	0	1	0	0	1	0	0

Full Example of VICCard 2.0

To demonstrate VICCard 2.0 in its entirety, we will completely encrypt the plaintext `Attack At Dawn`. We will use the same keyed deck that we used to demonstrate the first version of VICCard:

$$[A\heartsuit,\ 7\heartsuit,\ K\clubsuit,\ 8\heartsuit,\ K\diamondsuit,\ J\clubsuit,\ K\spadesuit,\ K\heartsuit,\ 4\spadesuit,\ 8\diamondsuit,\ 4\heartsuit,\ 7\clubsuit,\ 3\clubsuit,\ 10\diamondsuit,\ Q\heartsuit,\ 10\clubsuit,\ 5\diamondsuit,\ 2\spadesuit,\ J\spadesuit,\ A\clubsuit,\ 9\clubsuit,\ 4\clubsuit,\ 3\diamondsuit,\ 3\heartsuit,\ 8\clubsuit,\ 7\diamondsuit,\ 5\spadesuit,\ 5\heartsuit,\ 2\clubsuit,\ A\diamondsuit,\ 8\spadesuit,\ 10\spadesuit,\ 6\heartsuit,\ 9\spadesuit,\ 10\heartsuit,\ 6\diamondsuit,\ Q\diamondsuit,\ 6\clubsuit,\ 2\diamondsuit,\ J\diamondsuit,\ 7\spadesuit,\ 5\clubsuit,\ 4\diamondsuit,\ J\heartsuit,\ Q\spadesuit,\ 6\spadesuit,\ 3\spadesuit,\ Q\clubsuit,\ 9\heartsuit,\ 2\heartsuit,\ A\spadesuit,\ 9\diamondsuit]$$

Before beginning encryption, we will randomly change the cases of the letters in the plaintext: `aTTACk aT DaWN`. A lowercase letter signifies when a black card is chosen, and an uppercase letter signifies when a red card is chosen. Now, we will go through the four steps of VICCard.

Step 1: We begin by using the substitution checkerboard in Table 2 to convert the plaintext into cards. The first letter of the plaintext is `a`. Since `a` is represented by $A\spadesuit$, we look for $A\spadesuit$ in the checkerboard. This card is in the **Q** row and the $\diamondsuit$ column, implying that `a` is converted to the numbers 12 and 3. The second letter of the plaintext is `T`. Since `T` is represented by $7\diamondsuit$, we look for $7\diamondsuit$ in the checkerboard. This card is in the **K** row and the $\heartsuit$ column, implying that `a` is converted to the numbers 0 and 1. Continuing this for the rest of the message, we get the following strings of face values and suits:

a	T	T	A	C	k	a	T	D	a	W	N
12	0	0	1	11	6	12	0	11	12	1	4
3	1	1	0	1	1	3	1	0	3	1	2

Step 2: Next, we perform the double columnar transpositions. We will start with the face values. The first transposition uses the order of $A\clubsuit$ through $8\clubsuit$. In this deck, that order is $[7\clubsuit, 3\clubsuit, A\clubsuit, 4\clubsuit, 8\clubsuit, 2\clubsuit, 6\clubsuit, 5\clubsuit]$.

$7\clubsuit$	$3\clubsuit$	$1\clubsuit$	$4\clubsuit$	$8\clubsuit$	$2\clubsuit$	$6\clubsuit$	$5\clubsuit$
12	0	0	1	11	6	12	0
11	12	1	4				

This transposition gives us this string of face values: (0 1 6 0 12 1 4 0 12 12 11 11). The second transposition uses the order of $A\heartsuit$ through $6\heartsuit$. In this deck, that order is $[A\heartsuit, 4\heartsuit, 3\heartsuit, 5\heartsuit, 6\heartsuit, 2\heartsuit]$.

$A\heartsuit$	$4\heartsuit$	$3\heartsuit$	$5\heartsuit$	$6\heartsuit$	$2\heartsuit$
0	1	6	0	12	1
4	0	12	12	11	11

This transposition gives us the following face values: (0 4 1 11 6 12 1 0 0 12 12 11). Now we will transpose the suits. The key of the first transposition is the order of $9\clubsuit$ through $K\clubsuit$, which is $[K\clubsuit, J\clubsuit, 10\clubsuit, 9\clubsuit, Q\clubsuit]$.

$K\clubsuit$	$J\clubsuit$	$10\clubsuit$	$9\clubsuit$	$Q\clubsuit$
3	1	1	0	1
1	3	1	0	3
1	2			

This gives us these suits: (0 0 1 1 1 3 2 1 3 3 1 1). The second transposition uses the order of $7\heartsuit$ through $K\heartsuit$. In this deck, that order is $[7\heartsuit, 8\heartsuit, K\heartsuit, Q\heartsuit, 10\heartsuit, J\heartsuit, 9\heartsuit]$.

$7\heartsuit$	$8\heartsuit$	$K\heartsuit$	$Q\heartsuit$	$10\heartsuit$	$J\heartsuit$	$9\heartsuit$
0	0	1	1	1	3	2
1	3	3	1	1		

This gives us this order of suits: (0 1 0 3 2 1 1 3 1 1 1 3). In summary, the two double columnar transpositions produce the following two rows of face values and suits:

0	4	1	11	6	12	1	0	0	12	12	11
0	1	0	3	2	1	1	3	1	1	1	3

Step 3: Now, we add the numbers produced from the lagged Fibonacci generators to these face values and suits. The seed for the face values is the order of the spades

$[K♠, 4♠, 2♠, J♠, 5♠, 8♠, 10♠, 9♠, 7♠, Q♠, 6♠, 3♠, A♠]$, and the seed for the suits is the order of the diamonds $[K\diamondsuit, 8\diamondsuit, 10\diamondsuit, 5\diamondsuit, 3\diamondsuit, 7\diamondsuit, A\diamondsuit, 6\diamondsuit, Q\diamondsuit, 2\diamondsuit, J\diamondsuit, 4\diamondsuit, 9\diamondsuit]$. We will convert the former to integers modulo 13 (0 4 2 11 5 8 10 9 7 12 6 3 1), and we will convert the latter to integers modulo 4 (1 0 2 1 3 3 1 2 0 2 3 0 1). If the plaintext were longer, we would use these seeds to generate longer strings of numbers. We now add these numbers to the face values and suits.

	0	4	1	11	6	12	1	0	0	12	12	11
+	0	4	2	11	5	8	10	9	7	12	6	3
=	0	8	3	9	11	7	11	9	7	11	5	1

	0	1	0	3	2	1	1	3	1	1	1	3
+	1	0	2	1	3	3	1	2	0	2	3	0
=	1	1	2	0	1	0	2	1	1	3	0	3

This gives us the following face values and suits:

0	8	3	9	11	7	11	9	7	11	5	1
1	1	2	0	1	0	2	1	1	3	0	3

Step 4: Lastly, we finish the encryption by converting the cards back into letters, using the substitution checkerboard of Step 1. The first column of numbers has a face value of 0 and a suit of 1. Looking in the 0 row and the 1 column, we find the $7\diamondsuit$. Since the $7\diamondsuit$ represents the letter `T`, the first ciphertext character is `T`. The second column of numbers has a face value of 8 and a suit of 1. Looking in the 8 row and the 1 column, we find $9♣$. Since $9♣$ represents the letter `v`, the next ciphertext character is `v`. Continuing this, we get the ciphertext `TvodCm Yq nBZX`.

0	8	3	9	11	7	11	9	7	11	5	1
1	1	2	0	1	0	2	1	1	3	0	3
T	v	o	d	C	m	Y	q	n	B	Z	X

6.2 VICCard 3.0

One of the most important principles in analyzing a cipher is diffusion. The principle of diffusion states that a secure cipher should have most of the ciphertext dependant on a single plaintext character. Practically, this means that changing one plaintext letter should change most of the ciphertext. As it stands, VICCard 2.0 does not produce a significant amount of diffusion. For example, the above deck order was used to encrypt the message `attack at dawn` in order to produce the ciphertext `tBexDf Ov xBKa`. Then, the same deck order was used to encrypt `attbck at dawn` in order to produce the ciphertext `sBFxDf Ov xBKa`. Changing one letter of the plaintext only changed the first and third letters of the ciphertext. The first step of VICCard 2.0 reveals why there is a lack of diffusion. After we use the checkerboard to convert `attack at dawn` to playing cards, we get the following two rows of numbers:

12	12	12	12	8	6	12	12	9	12	3	7
3	0	0	3	3	1	3	0	0	3	1	1

Here are the two rows of numbers when we apply the same checkerboard to `attbck at dawn`:

12	12	12	5	8	6	12	12	9	12	3	7
3	0	0	1	3	1	3	0	0	3	1	1

The only column of numbers that changes is the fourth one. After we apply Steps 2 and 3 to these streams of numbers, we get the following results for the first and second plaintexts, respectively:

12	11	1	6	11	7	0	8	6	11	5	12
0	3	2	0	0	3	2	1	0	3	3	3

12	11	7	6	11	7	0	8	6	11	5	12
2	3	2	0	0	3	2	1	0	3	3	3

After we apply Step 1 to the plaintexts, we get two sets of numbers in which the two numbers in the fourth column are different. The columnar transpositions of Step 2 most likely separate the two numbers. After Step 3 adds the lagged Fibonacci generated numbers to these sets of numbers, this results in only two numbers in the first and third columns being different. Hence, only two letters of the ciphertext change.

In order to add more diffusion, VICCard 3.0 creates a new version of Step 1. Instead of using a checkerboard, VICCard 3.0 uses a second deck. This deck contains all the clubs, hearts, spades, and diamonds arranged numerically. Since we are using two decks, we will refer to the first keyed deck as the plaintext deck, and we will refer to this second deck as the ciphertext deck.

$$[A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, 2\heartsuit, A\spadesuit, 9\diamondsuit]$$

$$[A\clubsuit, 2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, K\diamondsuit]$$

This starting position is equivalent to when the plaintext deck is placed in the checkerboard. For instance, just as $A\heartsuit$ goes in the A row and the $\clubsuit$ column, so here $A\heartsuit$ and $A\clubsuit$ are in the same positions in their respective decks. Just as $Q\heartsuit$ is in the 2 row and the $\heartsuit$ column, so here $Q\heartsuit$ and $2\heartsuit$ are both the fifteenth card of their respective decks. However, this order will not remain the same. As we encrypt a message, we will permute both deck orders.

Again, we will encrypt `attack at dawn` as an example. Since $A\spadesuit$ represents `a`, we find $A\spadesuit$ in the plaintext deck. It is the fifty-first card in the deck. We now look at the fifty-first card of the ciphertext deck, which is $Q\diamondsuit$. Hence, `a` encrypts to $Q\diamondsuit$, which is 12 over 3 in number form. With previous versions of VICCard, we would simply move onto encrypting the next letter `t`. By contrast, here we will permute each of the decks before we do so. We will start with the ciphertext deck. First, we cut the cards so that the ciphertext card that was used ($Q\diamondsuit$) is moved to the face of the deck.

$$[K\diamondsuit, A\clubsuit, 2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit]$$

Second, we take the fifty-first card ($J\diamondsuit$) and move this to the twenty-sixth position.

$$[K\diamondsuit, A\clubsuit, 2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, J\diamondsuit, Q\heartsuit, K\heartsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, Q\diamondsuit]$$

Now, we will permute the plaintext deck in a slightly different manner. First, we cut the cards so that the plaintext card that was used ($A\spadesuit$) is moved to the back of the deck.

$$[A\spadesuit, 9\diamondsuit, A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, 2\heartsuit]$$

Second, we move the fiftieth card ($Q\clubsuit$) to the twenty-sixth position.

$$[A\spadesuit, 9\diamondsuit, A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, Q\clubsuit, 3\heartsuit, 8\clubsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, 9\heartsuit, 2\heartsuit]$$

Now we move onto encrypting the rest of the message. After encrypting each letter, we perform this permutation. Eventually, this gives us the following two rows of face values and suits:

By contrast, these are the two rows that result from using the same decks to convert the plaintext `attbck at dawn` to cards:

12	0	1	2	1	11	5	6	4	8	12	4
3	0	1	0	0	1	0	1	3	0	3	2

12	0	1	8	0	11	5	6	4	8	9	3
3	0	1	1	3	1	0	1	3	0	3	2

Notice that there are more changes in these two grids of numbers. After we apply the transpositions and lagged Fibonacci generators, we get two vastly different ciphertexts. The plaintext `attack at dawn` encrypts to `WmLqscGvUrQY`, and `attbck at dawn` encrypts to `elvvtbVfpXlJ`. Hence, this version of VICCard produces much more diffusion than previous iterations.

This seemingly arbitrary way of permuting the two decks is based on Chaocipher, a playing card cipher discussed in the introduction. Chaocipher operates by having two discs: a plaintext disc and a ciphertext disc. Each of these discs contains the 26 letters of the English alphabet in two random orders. Chaocipher encrypts messages by mapping letters from the plaintext disc to the ciphertext disc. After each encryption of a letter, Chaocipher permutes the letters on both discs. Taking inspiration from this idea, Aaron Toponce created a version of Chaocipher that is executed with playing cards. With Toponce's version, the 26 black cards are the plaintext disc and the 26 red cards are the ciphertext disc [18]. With VICCard 3.0, we are doing something slightly different. Here, each "disc" has 52 cards instead of 26, allowing us to encrypt both uppercase and lowercase letters.

Full Example of VICCard 3.0

To demonstrate the full VICCard 3.0 algorithm, let's encrypt `Attack At Dawn`. Here is the same keyed deck that we have been using:

$$[A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, 2\heartsuit, A\spadesuit, 9\diamondsuit]$$

Also, we will change the cases of some of the letters before getting started: `ATtack At dawN`.

Step 1: To convert the plaintext into cards, we will use the method from Chaocipher. This requires using the keyed deck and a second ciphertext deck:

$$[A\clubsuit, 2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, K\diamondsuit]$$

Starting with the first plaintext letter `A`, we find this letter's card $A\heartsuit$ in the keyed deck. It is the first card from the left. Looking at the card in the same position in the ciphertext deck, which is $A\clubsuit$, we see that `A` is turned into the card $A\clubsuit$. Of course, we represent $A\clubsuit$ as a face value of 1 and a suit of 0. Before encrypting the next letter `T`, we will permute both decks. For the keyed deck, we cut the cards so that $A\heartsuit$ is on bottom. Since it is already on the bottom, we do not have to worry about this step. We then take the fiftieth card, which is $2\heartsuit$, and move this card to the twenty-sixth position:

$$[A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 2\heartsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, A\spadesuit, 9\diamondsuit]$$

For the ciphertext deck, we cut the cards so that $A\clubsuit$ is on top:

$$[2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, K\diamondsuit, A\clubsuit]$$

We then move the fifty-first card, which is $K\diamondsuit$, to the twenty-sixth position:

$$[2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, K\diamondsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, A\clubsuit]$$

Now, we can encrypt the second letter. Continuing this, we get the face values and suits as shown in Table 12.

1	1	1	0	1	11	7	6	5	10	1	1
0	2	1	3	0	1	0	1	3	2	2	3

Table 12: Result of Chaocipher-like Substitution

Step 2: For the next step, we will perform the double columnar transpositions. Tables 13 and 14 display the face value transpositions, and Tables 15 and 16 display the suit transpositions. Table 17 shows the final results of the transpositions.

7♣	3♣	1♣	4♣	8♣	2♣	6♣	5♣
1	1	1	0	1	11	7	6
5	10	1	1				
Result: 1 1 11 1 10 0 1 6 7 1 5 1							

Table 13: First Face Value Transposition

A♡	4♡	3♡	5♡	6♡	2♡
1	1	11	1	10	0
1	6	7	1	5	1
Result: 1 1 0 1 11 7 1 6 1 1 10 5					

Table 14: Second Face Value Transposition

K♣	*J*♣	10♣	9♣	*Q*♣
0	2	1	3	0
1	0	1	3	2
2	3			
Result: 3 3 1 1 2 0 3 0 2 0 1 2				

Table 15: First Suit Transposition

7♡	8♡	*K*♡	*Q*♡	10♡	*J*♡	9♡
3	3	1	1	2	0	3
0	2	0	1	2		
Result: 3 0 3 2 3 2 2 0 1 1 1 0						

Table 16: Second Suit Transposition

1	1	0	1	11	7	1	6	1	1	10	5
3	0	3	2	3	2	2	0	1	1	1	0

Table 17: Final Results of the Transpositions

Step 3: Next, we add the numbers from the lagged Fibonacci generators to the face values and suits.

Here are the results for the face values:

	1	1	0	1	11	7	1	6	1	1	10	5
+	0	4	2	11	5	8	10	9	7	12	6	3
=	1	5	2	12	3	2	11	2	8	0	3	8

Here are the results for the suits:

	3	0	3	2	3	2	2	0	1	1	1	0
+	1	0	2	1	3	3	1	2	0	2	3	0
=	0	0	1	3	2	1	3	2	1	3	0	0

Thus, here are the last two rows of face values and suits:

1	5	2	12	3	2	11	2	8	0	3	8
0	0	1	3	2	1	3	2	1	3	0	0

Step 4: Finally, we will convert these face value and suits back into letters using the method from Chaocipher. Like Step 1, we will again use the keyed deck and the ciphertext deck. However, there is one important difference. In Step 1, we went from the keyed deck to the ciphertext deck. Here, we will do the opposite and move from the ciphertext deck to the keyed deck.

$$[A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, 2\heartsuit, A\spadesuit, 9\diamondsuit]$$

$$[A\clubsuit, 2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, K\diamondsuit]$$

The first column of numbers has a face values of 1 and a suit of 0, which tells us to find $A\clubsuit$ in the ciphertext deck. Since this card is the first card from the left, we go to the card in the same position in the keyed deck, which is $A\heartsuit$. Hence, our first ciphertext character is `A`. Now, we permute each deck as per Chaocipher.

$$[A\heartsuit, 7\heartsuit, K\clubsuit, 8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 2\heartsuit, 7\diamondsuit, 5\spadesuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, A\spadesuit, 9\diamondsuit]$$

$$[2\clubsuit, 3\clubsuit, 4\clubsuit, 5\clubsuit, 6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, K\diamondsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, A\clubsuit]$$

The second column of numbers gives us the card $5\clubsuit$. This is the fourth card from the left in the ciphertext deck, which tells us to find the corresponding card $8\heartsuit$ in the keyed deck. Hence, the second ciphertext character is `H`. Before continuing, we again permute each deck:

$$[8\heartsuit, K\diamondsuit, J\clubsuit, K\spadesuit, K\heartsuit, 4\spadesuit, 8\diamondsuit, 4\heartsuit, 7\clubsuit, 3\clubsuit, 10\diamondsuit, Q\heartsuit, 10\clubsuit, 5\diamondsuit, 2\spadesuit, J\spadesuit, A\clubsuit, 9\clubsuit, 4\clubsuit, 3\diamondsuit, 3\heartsuit, 8\clubsuit, 2\heartsuit, 7\diamondsuit, 5\spadesuit, A\heartsuit, 5\heartsuit, 2\clubsuit, A\diamondsuit, 8\spadesuit, 10\spadesuit, 6\heartsuit, 9\spadesuit, 10\heartsuit, 6\diamondsuit, Q\diamondsuit, 6\clubsuit, 2\diamondsuit, J\diamondsuit, 7\spadesuit, 5\clubsuit, 4\diamondsuit, J\heartsuit, Q\spadesuit, 6\spadesuit, 3\spadesuit, Q\clubsuit, 9\heartsuit, A\spadesuit, 9\diamondsuit, 7\heartsuit, K\clubsuit]$$

$$[6\clubsuit, 7\clubsuit, 8\clubsuit, 9\clubsuit, 10\clubsuit, J\clubsuit, Q\clubsuit, K\clubsuit, A\heartsuit, 2\heartsuit, 3\heartsuit, 4\heartsuit, 5\heartsuit, 6\heartsuit, 7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, K\diamondsuit, A\spadesuit, 2\spadesuit, 3\spadesuit, 4\clubsuit, 4\spadesuit, 5\spadesuit, 6\spadesuit, 7\spadesuit, 8\spadesuit, 9\spadesuit, 10\spadesuit, J\spadesuit, Q\spadesuit, K\spadesuit, A\diamondsuit, 2\diamondsuit, 3\diamondsuit, 4\diamondsuit, 5\diamondsuit, 6\diamondsuit, 7\diamondsuit, 8\diamondsuit, 9\diamondsuit, 10\diamondsuit, J\diamondsuit, Q\diamondsuit, A\clubsuit, 2\clubsuit, 3\clubsuit, 5\clubsuit]$$

The final ciphertext is `AHpyBd qc pgKy`.

6.3 VICCard 4.0

Because VICCard performs columnar transpositions, we will take advantage of a simple yet effective cryptography technique: junk letters. Adding meaningless junk letters throughout the encryption process is an efficient way of confusing a code breaker. For example, consider our favorite plaintext `Attack At Dawn`. Before sending this message, suppose that I attach a conglomeration of random letters to the end of the message, giving me `AttackAtDawnwerutWnDbDGslw`. After performing the steps of VICCard, the meaningful and meaningless letters will be inextricably mashed and mixed together. When a code breaker then goes to decode the ciphertext, he will have a difficult time determining which aspects of the ciphertext came from meaningful letters and which came from meaningless letters. By contrast, reversing VICCard will produce the above plaintext, automatically separating the wanted from the unwanted characters. In this way, creating a message with random characters is like mixing magnetic metal filings with nonmagnetic filings. They both look the same, and the only way to properly separate them is by using a magnet. Similarly, whereas the decryption algorithm makes decoding the message easy, the code breaker is constantly guessing whether information in the ciphertext is relevant or not.

To create the fourth version of VICCard, I implemented random characters in two ways. First, the person using VICCard has the option of adding random letters to the end of their plaintext before beginning encryption. Second, VICCard adds random face values and suits during the columnar transpositions. For example, we will again work with the following rows of face values and suits:

1	12	12	12	8	6	1	12	11	12	3	7
0	0	0	3	3	1	0	0	0	3	1	1

Consider the face values. We will perform the following columnar transposition with seven columns:

7	3	1	4	2	6	5
1	12	12	12	8	6	1
12	11	12	3	7		

Notice that the bottom row is partially filled. In previous versions of VICCard, we performed the transposition with this partially filled row. In VICCard 4.0, we fill this bottom row with junk face values before completing the transposition. If the bottom row is completely filled, we will add an entire row of junk face values.

7	3	1	4	2	6	5
1	12	12	12	8	6	1
12	11	12	3	7	5	9

Now, we perform the columnar transposition: (12 12 8 7 12 11 12 3 1 9 6 5 1 12). Next, we will execute a columnar transposition with six columns, filling in the partially filled bottom row:

1	4	3	5	6	2
12	12	8	7	12	11
12	3	1	9	6	5
1	12	2	5	6	3

These two transpositions produce the following string of face values: (12 12 1 11 5 3 8 1 2 12 3 12 7 9 5 12 6 6).

Notice that during the transpositions, the random numbers are spread throughout the stream of face values. When the decoder is undoing the transpositions, the random characters are easily eliminated. For example, undoing the above transposition produces the following stream of numbers: (12 12 8 7 12 11 12 3 1 9 6 5 1 12 2 5 6 3). Currently, the random characters are at the end of the message. However, the decoder still needs to determine how many random characters there are. All they have to do is divide the string of numbers into sections of seven numbers and eliminate what remains: (12 12 8 7 12 11 12 | 3 1 9 6 5 1 12 | 2 5 6 3). The reason for this is that the first columnar transposition produces a string of numbers that is divisible by 7. Since the second columnar transposition adds anywhere from 1 to 6 more numbers, all the leftover numbers must be junk letters.

Here is another important change that I made to VICCard. In VICCard 2.0, recall that the transpositions of the face values use 8 and 6 columns and that the transpositions of the suits use 5 and 7 columns. Unfortunately, adding random numbers prevents us from doing this. If we perform these transpositions while adding random characters, it is likely that we will end up with strings of face values and suits that have different lengths. As a result, I changed the columnar transpositions such that the face values and suits are both transposed with seven columns first and six columns second. Specifically, VICCard transposes the face values using the first seven clubs and the first six hearts, and it transposes the suits using the remaining seven hearts and the remaining six clubs. Although using only 7 or 6 columns for transpositions is not optimal, the addition of random characters more than makes up for this.

Finally, the last major security feature that I added to VICCard 4.0 is another method of creating diffusion. Suppose that I am going to perform a seven-column transposition with the following string of face values: (12 5 3 7 5 8 9 3 11 10 6 7 4 8 5 3). There are sixteen numbers, which means that we must add 5 more random numbers. We will do that now before placing the face values into the transposition grid: (12 5 3 7 5 8 9 3 11 10 6 7 4 8 5 3 2 7 5 4 0). Before performing the transposition, we will divide this message into groups of seven: (12 5 3 7 5 8 9 | 3 11 10 6 7 4 8 | 5 3 2 7 5 4 0). We will add the numbers of the first group to those of the second group and add the new numbers of the second group to those of the third group, using modulo 13 addition. Adding the first group to the second group gives the following result: (12 5 3 7 5 8 9 | 2 3 0 0 12 12 4 | 5 3 2 7 5 4 0). Adding the second group to the third group gives the following result: (12 5 3 7 5 8 9 | 2 3 0 0 12 12 4 | 7 6 2 7 4 3 4). Now, we place these numbers into the grid to transpose them. This step is done before each transposition. We divide the numbers into groups of seven for the seven-column transpositions and groups of six for the six-column transpositions. Also, we use modulo 13 arithmetic for the face values and modulo 4 arithmetic for the suits. This increases diffusion because a change in any number will result in changes of multiple numbers after it when they are added together.

As an example, I encrypted `Attack At Dawn` and `Attbck At Dawn` using VICCard 4.0 to demonstrate this new method of increasing diffusion in combination with the power of adding random letters. Furthermore, to allow the new features to shine, I used the substitution checkerboard instead of the Chaocipher-like method of permutation. Using the same deck, `AttackAtDawn` encrypts to `ugevmrbKuSlrooGacL` and `AttbckAtDawn` encrypts to `uvfvRbbKYSaywQZack`. Out of the 18 letters in the ciphertext, 11 are different.

Full Example of VICCard 4.0

I am not going to lie. This one is intense. This version was simply a case study in how much cryptography we can jampack into one cipher. The final version of VICCard, which we will discuss next, is not as crazy.

Once again, let's return to `Attack At Dawn` and use the same keyed deck. As always, we will change some of the cases before encrypting: `attAcK at dAwN`. Now, we also will add some random junk letters to the end: `attAcKatdAwNRkEu`.

Step 1: We will apply the Chaocipher method to the keyed deck and the ciphertext deck to get the following face values and suits:

12	0	1	4	0	12	5	6	4	10	12	0	2	7	5	12
3	0	1	0	3	3	0	1	3	0	1	2	2	2	1	0

Step 2: Let's start with the face values. Since there are 16 face values, we must add 5 random face values to make the total divisible by 21:

12 0 1 4 0 12 5	6 4 10 12 0 2 7	5 12 8 3 12 4 8

Before we transpose the face values, we perform cipher block chaining. We take the first group of numbers and add them to the second group modulo 13:

12 0 1 4 0 12 5	5 4 11 3 0 1 12	5 12 8 3 12 4 8

We then add the second group of numbers to the third group:

12 0 1 4 0 12 5	5 4 11 3 0 1 12	10 3 6 6 12 5 7

Now that we have performed the cipher block chaining, we can now transpose the face values based on the order of $A♣$ through 7♣.

7♣	3♣	A♣	4♣	2♣	6♣	5♣
12	0	1	4	0	12	5
5	4	11	3	0	1	12
10	3	6	6	12	5	7

The result is (1 11 6 0 0 12 0 4 3 4 3 6 5 12 7 12 1 5 12 5 10).

For the second transposition of the face values, we first add 3 junk letters so that the number of face values is divisible by 6: (1 11 6 0 0 12 0 4 3 4 3 6 5 12 7 12 1 5 12 5 10 3 11 3). Before transposing the face values, we divide them into groups of 6 and perform cipher block chaining to get the following: (1 11 6 0 0 12 1 2 9 4 3 5 6 1 3 3 4 10 5 6 0 6 2 0). Now we will perform the transposition based on the order of A♡ through 6♡.

A♡	4♡	3♡	5♡	6♡	2♡
1	11	6	0	0	12
1	2	9	4	3	5
6	1	3	3	4	10
5	6	0	6	2	0

The result is (1 1 6 5 12 5 10 0 6 9 3 0 11 2 1 6 0 4 3 6 0 3 4 2).

Next, we follow the same process to transpose the suits. We add five random suits so that the number of suits is divisible by seven: (3 0 1 0 3 3 0 1 3 0 1 2 2 2 1 0 3 2 3 0 1). After cipher block chaining based on blocks of seven, this becomes (3 0 1 0 3 3 0 0 3 1 1 1 1 2 1 3 0 3 0 1 3). Now we will transpose the suits based on the order of 7♡ through K♡.

7♡	8♡	K♡	Q♡	10♡	J♡	9♡
3	0	1	0	3	3	0
0	3	1	1	1	1	2
1	3	0	3	0	1	3

The result is (3 0 1 0 3 3 0 2 3 3 1 0 3 1 1 0 1 3 1 1 0).

For the final transposition, we add three more junk suits so that the total is divisible by six: (3 0 1 0 3 3 0 2 3 3 1 0 3 1 1 0 1 3 1 1 0 1 0 3). Next comes the cipher block chaining based on blocks of six: (3 0 1 0 3 3 3 2 0 3 0 3 2 3 1 3 1 2 3 0 1 0 1 1). Finally, we transpose the suits based on the order of 8♣ through K♣.

K♣	J♣	10♣	9♣	8♣	12♣
3	0	1	0	3	3
3	2	0	3	0	3
2	3	1	3	1	2
3	0	1	0	1	1

The result is (3 0 1 1 0 3 3 0 1 0 1 1 0 2 3 0 3 3 2 1 3 3 2 3). Here are the face values and suits after the transpositions:

1	1	6	5	12	5	10	0	6	9	3	0	11	2	1	6	0	4	3	6	0	3	4	2
3	0	1	1	0	3	3	0	1	0	1	1	0	2	3	0	3	3	2	1	3	3	2	3

Step 3:

We first add the numbers from the face value lagged Fibonacci generator to the face values,

	1	1	6	5	12	5	10	0	6	9	3	0	11	2	1	6	0	$\cdots$
$+$	13	4	2	11	5	8	10	9	7	12	6	3	1	4	6	0	3	$\cdots$
$=$	1	5	8	3	4	0	7	9	0	8	9	3	12	6	7	6	3	$\cdots$

	$\cdots$	4	3	6	0	3	4	2
$+$	$\cdots$	0	5	6	3	6	5	9
$=$	$\cdots$	4	8	12	3	9	9	11

and then add the numbers from the suit lagged Fibonacci generator to the suits,

	3	0	1	1	0	3	3	0	1	0	1	1	0	2	3	0	3	3	2	1	3	3	2	3
$+$	1	0	2	1	3	3	1	2	0	2	3	0	1	1	2	3	0	2	0	3	2	2	1	3
$=$	0	0	3	2	3	2	0	2	1	2	0	1	1	3	1	3	3	1	2	0	1	1	3	2

giving us the following face values and suits:

1	5	8	3	4	0	7	9	0	8	9	3	12	6	7	6	3	4	8	12	3	9	9	11
0	0	3	2	3	2	0	2	1	2	0	1	1	3	1	3	3	1	2	0	1	1	3	2

Step 4: Finally, we use the Chaocipher method to convert these face values and suits into ciphertext letters. As a quick reminder, remember that in Step 4 we move from the ciphertext deck to the keyed deck. Following this process gives us the following ciphertext: `AhlTOiVEkZaxgNMEZVwgvZuQ`.

6.4 VICCard 5.0

Up to this point, we have been using double columnar transpositions. However, this is not exactly what VIC did. Although VIC does use two columnar transpositions, the second transposition is special. The VIC cipher reads numbers out of the grid in exactly the same way: from the top to the bottom based on the keyword. However, VIC places numbers into the grid differently.

For example, we will use the following order of six hearts to perform a face value transposition: [5♡, 6♡, 3♡, A♡, 2♡, 4♡]. Before filling the transposition grid, we will use the key to divide the grid into triangular sections. We will represent these sections by filling them with the letter **T**. We will start with the smallest number of the key, which is 1. We section off all cells to the right of and including the part of the grid under the 1. We repeat this process on the next row, starting at the next column over. We continue to section off cells until we run out of columns, producing a triangular.

We do this for all numbers in the key. The following grid shows the results of sectioning off cells based on the numbers 1-4.

5	6	3	1	2	4
			T	T	T
				T	T
					T
				T	T
					T
		T	T	T	T
			T	T	T
				T	T
					T
					T

We will transpose the following sequence of face values: (4 0 8 12 5 1 6 8 9 0 1 9 1 8 7 11 12 8 9 6 3 2 2 12 7 0 12 7). Since there are 28 face values, we know that we will completely fill four rows and partially fill a fifth row. With this in mind, we begin by filling in all the cells that have not been sectioned off.

5	6	3	1	2	4
4	0	8	T	T	T
12	5	1	6	T	T
8	9	0	1	9	T
1	8	7	11	T	T
12	8	9	6		T

Now, we place the rest of the face values in the sectioned off cells.

5	6	3	1	2	4
4	0	8	3	2	2
12	5	1	6	12	7
8	9	0	1	9	0
1	8	7	11	12	7
12	8	9	6		

Finally, we finish the transposition by reading the numbers out from the grid: (3 6 1 11 6 2 12 9 12 8 1 0 7 9 2 7 0 7 4 12 8 1 12 0 5 9 8 8).

This unique transposition contributed to the remarkable security of VIC. As a result, it makes sense to apply it to VICCard. With VICCard 5.0, the first transposition for the face values and the suits is a normal columnar transposition. By contrast, the second transposition for the face values and the suits is this special transposition.

In addition, I made another crucial change to VICCard. I removed the Chaocipher-like permutation and reverted back to the checkerboard substitution. The reason for this is that the Chaocipher-like permutation makes VICCard remarkably impractical. I attempted to hand-encrypt a message of 56 characters using VICCard 4.0, and it took me about three hours. Furthermore, I make a few errors that completely compromised the message. This is because after each letter encryption, Chaocipher permutes the plaintext and ciphertext decks. If the encoder permutes these improperly, the rest of the message will be encoded improperly. Although adding it to VICCard does increase its security, it makes hand encryption infeasible. Instead, I reserved the Chaocipher-like permutation solely for the computer version of VICCard, where human error does not come into play.

Finally, I added two small options to VICCard that lend to its security. First, I added the option of placing junk letters at both the beginning and the end of the message. Second, the encoder can "cut" the message before encrypting it. This is similar to cutting a deck of cards. In order to mark where he cuts the cards, he should place a marker, such as `xx`, where the intelligible words begin. Let's apply these two strategies to encrypting `Attack At Dawn`. First, we add random characters to the beginning and the end to get `EdfFxxAttackAtDawndsjfSDRmk`. Second, we will "cut" the message at the `k` in `Attack`, giving us `kAtDawndsjfSDRmkEdfFxxAttac`. Now, we can move onto the four steps of VICCard to encrypt the message. When the decoder decrypts the ciphertext, he will get `kAtDawndsjfSDRmkEdfFxxAttac` as his message. In order to read it, all he needs to do is "cut" the marker `xx` to the end: `AttackAtDawndsjfSDRmkEdfFxx`. Removing the junk letters at the end, he can now read the message: `AttackAtDawn dsjfSDRmkEdfFxx`.

Full Example of VICCard 5.0

For the last time, we will encrypt our favorite message `Attack At Dawn`. With VICCard 5.0, we will perform quite a few modifications to the plaintext before encryption. First, we will change some of the cases `ATTacKaTdaWN`. Second, we will add some random letters to the beginning and the end `jWvxxATTacKaTdaWNMvHsi`. Third, we will cut the message at a random point `acKaTdaWNMvHsijWvxxATT`. Notice that we purposefully added to the two `x`'s to mark where the message begins. After all of this prep work, we can begin encryption.

Step 1: For Step 1, we revert back to the substitution checkerboard method that was described in the first and second versions. Here is the result of using the checkerboard to convert the plaintext into face values and suits:

a	c	K	a	T	d	a	W	N	M	v	H	s	i	j	W	v	x	x	A	T	T
12	8	5	12	0	9	12	1	4	8	8	4	12	8	6	1	8	6	6	1	0	0
3	3	3	3	1	0	3	1	2	0	1	0	2	2	2	1	1	0	0	0	1	1

Step 2:

As usual, we will start with the double columnar transposition of the face values. The first columnar transposition is nothing special. It is the exact same type of transposition that we have been doing. After adding six random face values to the end so the number of face values is divisible by seven (12 8 5 12 0 9 12 1 4 8 8 4 12 8 6 1 8 6 6 1 0 0 3 4 5 3 0 3), we perform cipher block chaining based on blocks of seven (12 8 5 12 0 9 12 0 12 0 7 4 8 7 6 0 8 0 10 9 7 6 3 12 5 0 9 10). Then we perform a plain columnar transposition based on the order of A♣ through 7♣.

7♣	3♣	A♣	4♣	2♣	6♣	5♣
12	8	5	12	0	9	12
0	12	0	7	4	8	7
6	0	8	0	10	9	7
6	3	12	5	0	9	10

The result is (5 0 8 12 0 4 10 0 8 12 0 3 12 7 0 5 12 7 7 10 9 8 9 9 12 0 6 6).

The second transposition begins normally. We add two random face values so the total is divisible by six (5 0 8 12 0 4 10 0 8 12 0 3 12 7 0 5 12 7 7 10 9 8 9 9 12 0 6 6 7 7), and we perform cipher block chaining on blocks of six letters (5 0 8 12 0 4 2 0 3 11 0 7 1 7 3 3 12 1 8 4 12 11 8 10 7 4 5 4 2 4). Before placing the face values in the grid, however, we need to create triangular sections based on the key, which is the order of A♡ through 6♡. Since we are only dealing with 30 face values, we only have to worry about the top five rows.

A♡	4♡	3♡	5♡	6♡	2♡
T	T	T	T	T	T
	T	T	T	T	T
		T	T	T	T
			T	T	T
				T	T

We first fill in the parts of the grid that are not sectioned off,

A♡	4♡	3♡	5♡	6♡	2♡
T	T	T	T	T	T
5	T	T	T	T	T
0	8	T	T	T	T
12	0	4	T	T	T
2	0	3	11	T	T

and then fill in the triangular sections:

$A\heartsuit$	$4\heartsuit$	$3\heartsuit$	$5\heartsuit$	$6\heartsuit$	$2\heartsuit$
0	7	1	7	3	3
5	12	1	8	4	12
0	8	11	8	10	7
12	0	4	4	5	4
2	0	3	11	2	4

The result is: (0 5 0 12 2 3 12 7 4 4 1 1 11 4 3 7 12 8 0 0 7 8 8 4 11 3 4 10 5 2).

Now will we do the same thing for the suits. We add five random suits so the total is divisible by seven (3 3 3 3 1 0 3 1 2 0 1 0 2 2 2 1 1 0 0 0 1 1 3 3 2 0 0 2), perform cipher block chaining based on blocks of seven (3 3 3 3 1 0 3 0 1 3 0 1 2 1 2 2 0 0 1 2 2 3 1 3 2 1 2 0), and then transpose the suits based on the order of $7\heartsuit$ through $K\heartsuit$:

$7\heartsuit$	$8\heartsuit$	$K\heartsuit$	$Q\heartsuit$	$10\heartsuit$	$J\heartsuit$	$9\heartsuit$
3	3	3	3	1	0	3
0	1	3	0	1	2	1
2	2	0	0	1	2	2
3	1	3	2	1	2	0

The result is (3 0 2 3 3 1 2 1 3 1 2 0 1 1 1 1 0 2 2 2 3 0 0 2 3 3 0 3).

In preparation for the second transposition, we add two random suits so the total is divisible by six (3 0 2 3 3 1 2 1 3 1 2 0 1 1 1 1 0 2 2 2 3 0 0 2 3 3 0 3 1 1), perform cipher-block chaining based on blocks of six letters (3 0 2 3 3 1 1 1 1 0 1 1 2 2 2 1 1 3 0 0 1 1 1 1 3 3 1 0 2 2), and create triangular sections in the grid based on the order of $8\clubsuit$ through $K\clubsuit$. Since there are 30 suits, we only are concerned with the first five rows of the grid.

$K\clubsuit$	$J\clubsuit$	$10\clubsuit$	$9\clubsuit$	$8\clubsuit$	$Q\clubsuit$
				T	T
					T
			T	T	T
				T	T
					T

We then fill the grid based on the triangular sections and transpose the suits.

The result is (1 1 3 0 1 3 1 3 1 0 2 1 1 2 0 0 1 1 2 3 1 1 1 2 2 3 3 0 2 1).

$K♣$	$J♣$	10♣	9♣	8♣	$Q♣$
3	0	2	3	1	1
3	1	1	1	1	1
0	1	1	3	3	1
2	2	2	1	0	2
1	3	0	0	1	2

In summary, here are the two rows of face values and suits:

0	5	0	12	2	3	12	7	4	4	1	1	11	4	3	···
1	1	3	0	1	3	1	3	1	0	2	1	1	2	0	···

···	7	12	8	0	0	7	8	8	4	11	3	4	10	5	2
···	0	1	1	2	3	1	1	1	2	2	3	3	0	2	1

Step 3: For Step 3, we add the numbers from the spade lagged Fibonacci generator to the face values,

	0	5	0	12	2	3	12	7	4	4	1	1	11	4	3	···
$+$	13	4	2	11	5	8	10	9	7	12	6	3	1	4	6	···
$=$	0	9	2	10	7	11	9	3	11	3	7	4	12	8	9	···

	···	7	12	8	0	0	7	8	8	4	11	3	4	10	5	2
$+$	···	0	3	0	5	6	3	6	5	9	4	5	10	6	3	3
$=$	···	7	2	8	5	6	10	1	0	0	2	8	1	3	8	5

and we add the numbers from the diamond lagged Fibonacci generator to the suits,

	1	1	3	0	1	3	1	3	1	0	2	1	1	2	0	···
$+$	1	0	2	1	3	3	1	2	0	2	3	0	1	1	2	···
$=$	2	1	1	1	0	2	2	1	1	2	1	1	2	3	2	···

	···	0	1	1	2	3	1	1	1	2	2	3	3	0	2	1
$+$	···	3	0	2	0	3	2	2	1	3	1	2	3	1	3	2
$=$	···	3	1	3	2	2	3	3	2	1	3	1	2	1	1	3

which gives us the following face values and suits:

0	9	2	10	7	11	9	3	11	3	7	4	12	8	9	⋯
2	1	1	1	0	2	2	1	1	2	1	1	2	3	2	⋯

⋯	7	2	8	5	6	10	1	0	0	2	8	1	3	8	5
⋯	3	1	3	2	2	3	3	2	1	3	1	2	1	1	3

Step 4: Finally, we use the substitution checkerboard from Step 1 to convert these face values and suits into the ciphertext: `OqLPmYJwConRscJfLchjIXOTgvewvk`.

7 The Final Product

Admittedly, keeping track of all these updates can be confusing. Thus, here is a concise summary of the final version of VICCard [4]. As stated before, VICCard has maintained its basic four-step structure. All the changes were focused on improving each individual step.

Before beginning encryption, it is important that we prepare the keyed deck and the plaintext. To prepare the deck, we write down the orders of each of the suits. This will make it much more convenient to look up the keys for the transpositions and lagged Fibonacci generators. To prepare the plaintext, we add junk letters to the end, add junk letters to the beginning, and cut the message. Now, we are ready to begin encryption.

First, we begin by using the deck order as a substitution checkerboard. Using the checkerboard, we convert the letters of the plaintext into face values and suits. For each plaintext letter, we can either substitute it with its corresponding black card or its corresponding red card.

Second, we perform a double columnar transposition on the face values and a double columnar transposition on the suits. Before each transposition of the face values, we add enough random face values to the end so that the message will fit perfectly into the transposition grid. Also, we perform cipher-block chaining on the face values before transposing them. The key of the first face value transposition is the order of the first seven clubs, and the key of the second face value transposition is the order of the first six hearts. Again, before

[4]See Appendix D for the C++ code that I created to implement VICCard 5.0.

each transposition of the suits, we add the requisite number of random suits and perform cipher block chaining. The key of the first suit transposition is the order of the remaining seven hearts, and the key of the second suit transposition is the order of the remaining six clubs.

Third, we add the numbers from the lagged Fibonacci generators to the face values and the suits. The seed for the face values is the order of the spades, and the seed for the suits is the order of the diamonds. The face value seed is taken modulo 13, and the suit seed is taken modulo 4.

Finally, we convert the final order of face values and suits back into letters. To do this, we use the substitution checkerboard from Step 1.

8 Randomness Tests

The fundamental concept of cryptography is randomness. The more unpredictable a cipher is, the harder it usually is to break. For example, recall our discussion of lagged Fibonacci generators. To ensure that the lagged Fibonacci generators of VICCard are a reliable option, it was necessary to examine the randomness of the numbers that it created. We did this by focusing on the periods and the number distributions.

To test the randomness of VICCard 5.0, I encrypted six plaintexts: the Declaration of Independence, "Paul Revere's Ride" by Henry Wadsworth Longfellow, the Gettysburg Address, the lyrics to "All I Ask Of You", the opening to *A Tale of Two Cities*, and Psalm 23. After encrypting each plaintext using VICCard 5.0, I analyzed the letter distributions of the ciphertexts. Since we are using uppercase and lowercase letters, the ciphertexts are composed of 52 different letters. In a truly random string of letters, each of the letters will occur about $\frac{1}{52}$ of the time. Hence, if the six ciphertexts have a letter distribution that closely resembles a random distribution, then we have strong evidence in favor of the randomness of VICCard 5.0.

8.1 Chi-Square Test on Ciphertexts

At this point, I enlisted the help of the Chi-Square test of fitness. This test tells us how closely a set of measured data resembles the expected data. In this case, I used the Chi-Square test to determine how closely the measured distributions of the six ciphertexts "fit" with truly random texts. Here is how the Chi-Square test works. First, we calculate the Chi-Square value of a particular ciphertext using the following formula.

$$\chi^2 = \sum_{i=1}^{n} \left(\frac{(O_i - E_i)^2}{E_i} \right)$$

In this formula[5], n is the number of possible outcomes, O_i represents the observed number of occurrences of an outcome, and E_i represents the expected number of occurrences of an outcome. For example, consider the ciphertext of the Declaration of Independence. Since there are 52 types of letters, $n = 52$. The ciphertext has 6600 letters according to the breakdown in Table 18.

To calculate this ciphertext's Chi-Square value, we first calculate each of the individual Chi-Square terms. For example, consider the letter `a`. This letter occurs 105 times. This is the observed number of occurrences, O_i. In a truly random string of letters, the letter `a` would occur about $\frac{1}{52}$ of the time. Since there are 6600 letters in the ciphertext of the Declaration of Independence, the expected number of occurrences E_i is $\frac{6600}{52} \approx 126.923$. Using the formula above, the Chi-Square term for `a` is $\frac{(O_i - E_i)^2}{E_i} \approx \frac{(105-126.923)^2}{126.923} \approx 3.7867$. This calculation is performed for each of the 52 letters, and the final Chi-Square value is the sum of all these 52 calculations: $\chi^2 \approx 54.4558$.

What does χ^2 tell us? This number signifies how well the data emulates perfect randomness by measuring the level of deviation from perfect randomness. The larger the Chi-Square value, the more the data deviates. To determine the amount of deviation, we use the Chi-Square value to calculate the associated p-value. Looking up the above χ^2 in a p-value table,

[5] χ^2 is the symbol for the Chi-Square value.

Letter	Number of Occurrences	Letter	Number of Occurrences
a	105	A	118
b	147	B	112
c	129	C	126
d	119	D	145
e	139	E	150
f	136	F	120
g	122	G	117
h	117	H	127
i	118	I	117
j	122	J	109
k	132	K	144
l	114	L	135
m	126	M	138
n	130	N	133
o	126	O	148
p	134	P	154
q	126	Q	115
r	113	R	124
s	122	S	138
t	114	T	127
u	125	U	117
v	143	V	122
w	117	W	119
x	139	X	130
y	135	Y	112
z	117	Z	136

Table 18: Letter Distribution of the Declaration of Independence

we see that this data has a p-value of 0.3444. This p-value means the following. If I create a string of 6600 letters by choosing each letter at random, there is a 0.3444 probability of getting a string of letters that has a Chi-Square value of 54.4558. In other words, there is a 0.3444 probability that the Declaration of Independence ciphertext is random. The relevance of the p-value is in its ability to measure the level of randomness in a ciphertext.

In order to have a respectable sample size, I encrypted six plaintexts with six different keyed deck and performed a Chi-Square test on each ciphertext. Table 19 contains a summary of the test data.

After compiling this data, I encrypted these six plaintexts a second time, using a different

Plaintext (length)	Ciphertext Length	χ^2	p-value
Psalm 23 (461)	468	42.2222	0.8044
Opening to *A Tale of Two Cities* (475)	480	52.1333	0.4296
"All I Ask Of You" (813)	822	57.6983	0.2415
Gettysburg Address (1149)	1158	56.7703	0.2688
"Paul Revere's Ride" (4054)	4062	39.8552	0.8705
Declaration of Independence (6591)	6600	54.4558	0.3444

Table 19: First Round of Chi-Square Test Results

keyed deck for each encryption. Table 20 contains the data from this second round of tests.

Plaintext (length)	Ciphertext Length	χ^2	p-value
Psalm 23 (461)	468	53.1111	0.3929
Opening to *A Tale of Two Cities* (475)	480	41.3000	0.8320
"All I Ask Of You" (813)	822	37.3285	0.9237
Gettysburg Address (1149)	1158	63.5060	0.1123
"Paul Revere's Ride" (4054)	4062	45.6928	0.6838
Declaration of Independence (6591)	6600	47.2703	0.6226

Table 20: Second Round of Chi-Square Test Results

8.2 Chi-Square Test on One-Time Pads

A second randomness test is a slight variation of the previous Chi-Square test. The first Chi-Square test measures the randomness of the ciphertexts. The second Chi-Square test measures the randomness of the associated one-time pads.

A one-time pad is a random string of numbers that is used to encrypt a message. For example, suppose that I want to encrypt `attack at dawn`. Since my plaintext is 12 letters long, I randomly choose a string of 12 numbers to be my one-time pad: 5 3 6 14 22 19 10 8 2 23 11 15. To encrypt the plaintext, I "add" the one-time pad to the plaintext. Since I cannot add numbers to letters, I first convert `attack at dawn` to numbers as per the following: `a` is represented with the number 1, `b` becomes 2, `c` becomes 3, and so forth. Now, I can add the one-time pad to the plaintext.

	a	t	t	a	c	k	a	t	d	a	w	n
	1	20	20	1	3	11	1	20	4	1	23	14
+	5	3	6	14	22	19	10	8	2	23	11	15
=	6	23	26	15	25	4	11	2	6	24	8	3
	f	w	z	o	y	d	k	b	f	x	h	c

Converting the sum back into letters gives us the ciphertext `fwzoyd kb fxhc`. The reason that we are concerned with one-time pads is because it is the only proven way to create perfect encryption. This is because the key is a truly random string of numbers, and there is no way to crack the cipher other than by trying every key by brute-force.

In this example, we applied a one-time pad to a plaintext in order to generate a ciphertext. Consider doing the reverse process. If we take a ciphertext that we have generated and subtract the plaintext from it, we get the one-time pad that was used to encrypt the plaintext. However, let's say that the plaintext was not encrypted with a one-time pad. In this case, subtracting the plaintext from the ciphertext tells us that the encryption process has the same effect as the calculated one-time pad. If this one-time pad is sufficiently random, this would act as evidence in favor of the cipher's security.

For my second test, I encrypted the same six ciphertexts using six different keys. Instead of performing another Chi-Square test on the ciphertexts, I first subtracted the plaintexts from the ciphertexts in order to find the six one-time pads. I then carried out a Chi-Square test on each one-time pad. Table 21 lists the results. Since the plaintexts are slightly shorter than the ciphertexts, the one-time pads are the same length as the corresponding plaintexts.

Plaintext	χ^2	p-value
Psalm 23	49.1866	0.5460
Opening to *A Tale of Two Cities*	60.4358	0.1717
"All I Ask Of You"	50.5326	0.4921
Gettysburg	29.0783	0.9942
"Paul Revere's Ride"	32.1944	0.9817
Declaration of Independence	63.8245	0.1072

Table 21: First Round of Chi-Square Test Results (One-Time Pad)

Table 22 contains the results from a second round of tests in which the same six plaintexts were encrypted with six different decks.

Plaintext	χ^2	p-value
Psalm 23	62.0456	0.1382
Opening to *A Tale of Two Cities*	63.7200	0.1089
"All I Ask Of You"	47.2066	0.6251
Gettysburg	52.0688	0.4321
"Paul Revere's Ride"	40.7884	0.8463
Declaration of Independence	60.8264	0.1631

Table 22: Second Round of Chi-Square Test Results (One-Time Pad)

8.3 The Washington Test

In the previous tests, I encrypted different plaintexts with different deck keys. With this third test, I encrypted different plaintexts with the same keyed deck. I took the first 29,120 letters of George Washington's Farewell Address and divided them into 28 groups of 1040 letters. I then encrypted each group using the same deck to create 28 ciphertexts of length 1044. Table 23 contains the Chi-Square values and p-values of each ciphertext. The average p-value is 0.4614.

8.4 Interpreting the Results

The rationale for performing these tests is as follows. An English plaintext tends to have a predictable distribution. Letters such as `e` and `t` occur very frequently, whereas letters such as `q` and `z` are comparatively rare. These tests examine whether VICCard can transform a typical English letter distribution into something more like pure randomness.

Executing these Chi-Square tests provides reasonable evidence as to VICCard 5.0's ability to turn an English plaintext into a random message. Notice that most of the ciphertexts and one-time pads have respectable p-values with the occasional outlier. The first and second Chi-Square tests on the ciphertext yielded average Chi-Square values of 0.4932 and 0.5945,

	χ^2	p-value
1	49.49425287	0.533610953
2	43.31800766	0.769024139
3	55.8697318	0.29694992
4	65.63218391	0.08162056
5	60.651341	0.166941401
6	51.5862069	0.450724746
7	64.13793103	0.10236771
8	59.75478927	0.187574286
9	58.65900383	0.215146577
10	45.01149425	0.709235389
11	36.5440613	0.936595831
12	48.79693487	0.561608159
13	49.59386973	0.529616108
14	52.38314176	0.420092715
15	35.8467433	0.946712769
16	53.57854406	0.375647609
17	42.81992337	0.785476088
18	64.9348659	0.090831388
19	63.54022989	0.111762895
20	68.72030651	0.049573757
21	52.38314176	0.420092715
22	55.47126437	0.309943572
23	45.90804598	0.675579632
24	45.21072797	0.701860096
25	62.84291188	0.123565899
26	47.20306513	0.625220692
27	40.92720307	0.842450584
28	38.53639847	0.900501941

Table 23: Chi-Square Test Results of Washington's Farewell Address

respectively. In other words, on average there is about a 50/50 chance that the ciphertext is random. This is further reflected by the average p-value of 0.4614 in the data from Washington's Farewell Address. The first and second Chi-Square tests on the one-time pads yielded average Chi-Square values of 0.5488 and 0.3856, respectively.

Fortunately, many of the p-values are very high. Unfortunately, there are just as many p-values that are less than optimal. Still, this is not necessarily a bad thing. These low p-values are cause for concern if they are the result of a bias within VICCard 5.0. In other words, is there a security weakness of VICCard 5.0 that is making it give us low p-values?

To ensure that this is not the case, I performed what is called a drill-down test. I randomly picked the ninth group from Washington's Farewell Address, encrypted it differently, and found that the letter `D` occurred 31 times, a recognizable deviation from the expected $\frac{1044}{52} \approx 20.0769$ times. I then tracked each `D` through the encryption process to see if there was a feature of the cipher that caused a bias towards encrypting `D` more than any other letter. Given the extensive amount of substitutions that are performed (the checkerboard, the cipher block chaining, and the lagged Fibonacci generators), I was unable to find any factors that steered the cipher towards the letter `D`.

Instead, it appears that the high Chi-Square values are the result of an expected level of variation. For example, the second Gettysburg Address ciphertext has a Chi-Square value of 63.5060 and a p-value of 0.1123. Since the ciphertext had 1158 letters, each letter was expected to occur about $\frac{1158}{52} \approx 22.2692$ of the time. The high Chi-Square is not because every letter significantly deviates from occurring 22.2692 of the time. Instead, there were 7 outlying letters that occurred about 10 times more or less than the expected number. This is an allowable level of variation and is not necessarily a sign of a weakness.

To further confirm this, we will use the Empirical Rule. The Empirical Rule states that with normal data distributions and binomial data distributions, the data tends to follow a bell-shaped curve. Numerically, this means that about 68% of the data falls within one standard deviation of the mean, about 95% of the data falls within two standard deviations, and nearly all of the data falls within three standard deviations [6]. For example, consider the encryption of the declaration of independence. Since we expect each letter to occur $\frac{1}{52}$ of the time and since there are 6600 letters in the ciphertext, the standard deviation is $\sqrt{\left(\frac{1}{52}\right)\left(1-\frac{1}{52}\right)\left(\frac{1}{6600}\right)} \approx 0.0016905$. Furthermore, the expected number of occurrences of each letter is $\frac{6600}{52} \approx 126.92$. This means that we expect 68% of the data to be within $(0.0016905)(6600) \approx 11.16$ of 126.92, 95% of the data to be within $(2)(0.0016905)(6600) \approx 22.31$ of 126.92, and almost all of the data to be within $(3)(0.0016905)(6600) \approx 33.47$ of

[6]The symbol for standard deviation is σ.

126.92. Tables 24 through 27 confirm that the data for the six plaintexts follow this trend.

Plaintext	Percent in σ	Percent in 2σ	Percent in 3σ
Psalm 23	69.23%	96.15%	100.00%
Opening to *A Tale of Two Cities*	69.23%	96.15%	100.00%
"All I Ask Of You"	63.46%	94.23%	100.00%
Gettysburg	61.54%	96.15%	100.00%
"Paul Revere's Ride"	71.15%	100.00%	100.00%
Declaration of Independence	67.31%	96.15%	100.00%

Table 24: Empirical Rule for First Round of Chi-Square Test Results

Plaintext	Percent in σ	Percent in 2σ	Percent in 3σ
Psalm 23	55.77%	96.15%	98.08%
Opening to *A Tale of Two Cities*	69.23%	98.08%	100.00%
"All I Ask Of You"	76.92%	100.00%	100.00%
Gettysburg	61.54%	86.54%	100.00%
"Paul Revere's Ride"	69.23%	94.23%	100.00%
Declaration of Independence	73.08%	96.15%	98.08%

Table 25: Empirical Rule for Second Round of Chi-Square Test Results

Plaintext	Percent in σ	Percent in 2σ	Percent in 3σ
Psalm 23	69.23%	92.31%	100.00%
Opening to *A Tale of Two Cities*	65.38%	94.23%	98.08%
"All I Ask Of You"	69.23%	98.08%	100.00%
Gettysburg	75.00%	100.00%	100.00%
"Paul Revere's Ride"	80.77%	98.08%	100.00%
Declaration of Independence	69.23%	90.38%	98.08%

Table 26: Empirical Rule for First Round of Chi-Square Test Results (One-Time Pad)

Plaintext	Percent in σ	Percent in 2σ	Percent in 3σ
Psalm 23	71.15%	96.15%	98.08%
Opening to *A Tale of Two Cities*	61.54%	92.31%	98.08%
"All I Ask Of You"	73.08%	94.23%	100.00%
Gettysburg	63.46%	98.08%	100.00%
"Paul Revere's Ride"	75.00%	96.15%	100.00%
Declaration of Independence	63.46%	92.31%	100.00%

Table 27: Empirical Rule for Second Round of Chi-Square Test Results (One-Time Pad)

9 Conclusion: Our Contribution to Card Cryptography

9.1 Reflections on the Process

The experiences gained through my Capstone research has been invaluable. First, through it I have made a contribution to a new field. Relatively few cryptographers have explored the idea of playing card ciphers, and I am honored to be a part of this field's development. Second, it has given me a renewed appreciation of computer science. The interdependence of mathematics research and computer science cannot be overstated. In testing the various versions of VICCard and analyzing the resulting data, I often found myself creating and testing new programs. In fact, this research has been a contributing factor in my decision to add a computer science major. Finally, it has furnished me with countless opportunities to publish papers, speak at conferences, and meet others with a mutual passion for cryptography. These experiences will stay with me as I finish my undergraduate studies and commence my graduate work.

9.2 Future Work

Since the inception of playing card ciphers is fairly recent, there is still much to be discovered. Only a handful of cryptographers, such as Aaron Toponce, have experimented with the security possibilities of a deck of cards. In addition to further improvements of VICCard, it would be interesting to explore other cryptographic strategies that are innate in playing cards.

However, cryptography is not limited to just creating ciphers. While reading the descriptions of VICCard, a thought might have crossed your mind. In order to encrypt and decrypt a message, both the message's sender and the message's receiver must know the order of the deck of cards. How do they both know this order? In cryptography, this is known as the key exchange problem. The Diffie-Hellman key exchange is a popular textbook example of this. Suppose that I want to securely communicate with someone that I have never met. In order to do this, we decide to create a secret key number. The Diffie-Hellman key exchange uses exponentiation and modular arithmetic in order to accomplish this [1]. Thus, an interesting question to consider is whether we can create a secure key exchange algorithm using playing cards.

The interesting feature of the Diffie-Hellman key exchange is that it combines secret mathematical operations with public exchanges of certain data. The public exchanges allow both parties to create the same key, but the secret operations prevent third parties from determining the key. If we extend this idea further, we get public key cryptosystems. The most well-know public key cipher is RSA, which makes use of Euler's totient function and prime numbers [13]. Public key cryptosystems exploit this interplay between secret and shared information in order to create a secure cipher. It is called public key because despite the fact that keys are shared through public channels, these keys do not provide eavesdroppers with enough information to crack the cipher. It is worth exploring whether a similar concept can be developed using playing cards.

9.3 Final Thoughts

Ernő Rubik, the inventor of the Rubik's cube, had a fond way of describing his creation. He affirmed that the Rubik's cube "embodies the tension of our most basic contradictions: simplicity and complexity... and so forth" [16]. The cube is simple because a brief glance is enough to figure out the goal of the puzzle. Only a few seconds are needed to discover how the puzzle moves. However, determining the correct sequence of these moves is what makes it complex. It is this blend of simplicity and complexity that has driven the Rubik's cube's worldwide popularity [16].

This is the driving force behind playing card ciphers. A deck of cards is compact, portable, and readily accessible. However, its disarming simplicity is belied by the 225-bits of entropy that are packed into it. Furthermore, drawing out this wellspring of entropy is far from straightforward. The nascent field of playing card ciphers has brought to light many fascinating methods of doing so. In this research, I have presented my own contribution to this developing field.

VICCard 5.0 combines numerous cryptographic techniques. Certain features are reminiscent of VIC, which intensely addled the FBI during the Cold War. VICCard 5.0 also makes unique contributions of its own. It creates a novel substitution checkerboard, and it affords the incredible convenience of containing numerous keys in a single deck. In these regards, VICCard 5.0 distinguishes itself among other playing card ciphers. Furthermore, as demonstrated by the Chi-Square tests, it has the potential to create ciphertexts with respectable levels of randomness.

In a time when computer ciphers have become the industry standard, it is useful to not completely discount low-tech options. Analyzing the features that made hand ciphers secure for hundreds of years continues to inform and inspire our understanding of information security as a whole. In creating VICCard 5.0, my goal has been to show that computer ciphers have not entirely superseded hand ciphers. Additional innovation is still yielding formidable ciphers and fascinating cryptographic principles.

References

[1] Johannes Buchmann. *Introduction to Cryptography, Second Edition.* Springer-Verlag, New York, New York, 2004.

[2] Jim Dwyer. Sidelight to a Spy Saga: How a Brooklyn Newsboy's Nickel Would Turn Into a Fortune. `https://www.nytimes.com/2015/11/04/nyregion/how-a-brooklyn-newsboys-nickel-helped-convict-a-soviet-spy.html`, November 2015. Accessed July 8, 2020.

[3] FBI. Hollow Nickel/Rudolf Abel. `https://www.fbi.gov/history/famous-cases/hollow-nickel-rudolph-abel`. Accessed July 8, 2020.

[4] FermiLab. Physics Questions People Ask Fermilab. `https://www.fnal.gov/pub/science/inquiring/questions/atoms.html`, April 2014. Accessed July 20, 2020.

[5] Stephen Fry. Qi Card Shuffling - 52 Factorial. `https://www.youtube.com/watch?v=SLIvwtIuC3Y`, November 2012. Accessed July 23, 2020.

[6] Edy Victor Haryannto, Muhammad Zulfadly, Daifiria, Muhammad Barkah Akbar, and Ivy Lazuly. Implementation of Nihilist Cipher Algorithm in Securing Text Data With Md5 Verification. *Journal of Physics: Conference Series*, 1361, 2019.

[7] Jeffrey A. Hill. Chaocipher: Analysis and Models. `http://www.chaocipher.com/HillDocs/H03H09.pdf`, April 2009. Accessed June 8, 2020.

[8] David Kahn. Number One From Moscow. `https://www.cia.gov/library/center-for-the-study-of-intelligence/kent-csi/vol5no4/html/v05i4a09p_0001.htm#top`, July 2008. Accessed July 8, 2020.

[9] James Lyons. Cryptanalysis of the columnar transposition cipher. `http://practicalcryptography.com/cryptanalysis/stochastic-searching/`

`cryptanalysis-columnar-transposition-cipher/`, 2012. Accessed July 23, 2020.

[10] Matthew McKague. Design and analysis of RC4-like stream ciphers. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 2005.

[11] Sanjay Kumar Pal, Bimal Datta, and Amiya Karmakar. Cryptography and Network Security: A Historical Transformation. *SCHOLEDGE International Journal Of Multidisciplinary & Allied Studies*, 7(2):30–44, 2020.

[12] Isaac Reiter and Eric Landquist. Determining Biases in the Card-Chameleon Cryptosystem. *CONTACT*. (in press).

[13] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[14] Moshe Rubin. The Chaocipher challenge: Further work in progress. `http://www.mountainvistasoft.com/chaocipher/chaocipher-001.htm`, November 2009. Accessed July 23, 2019.

[15] Moshe Rubin. John F. Byrne's Chaocipher Revealed: An Historical and Technical Appraisal. *Cryptologia*, 35:328–379, 2011.

[16] Ian Scheffler. *Cracking the Cube*. Touchstone, New York, New York, 2016.

[17] Rotten Tomatoes. Bridge of spies. `https://www.rottentomatoes.com/m/bridge_of_spies`. Accessed August 26, 2020.

[18] Aaron Toponce. The Chaocipher Cipher. `https://aarontoponce.org/wiki/crypto/card-ciphers/chaocipher`, October 2018. Accessed May 24, 2019.

[19] Aaron Toponce. Playing card ciphers. `https://aarontoponce.org/wiki/crypto/card-ciphers`, October 2018. Accessed May 28, 2020.

[20] Wikipedia contributors. VIC cipher — Wikipedia, the free encyclopedia, July 2020. Accessed June 3, 2020.

[21] Wikipedia contributors. Transposition cipher — Wikipedia, the free encyclopedia, October 2020. Accessed June 3, 2020.

A Excerpt From Paper About Card-Chameleon

The following description of how Card-Chameleon works is taken from an article written by Dr. Landquist and myself entitled, "Determining Biases in the Card-Chameleon Cryptosystem" [12].

"Card-Chameleon is a stream cipher, encrypting a message one letter at a time with a six-step algorithm similar to IV procedure above.

1. Find the black card that represents the plaintext letter.
2. Look at the paired red card above this black card.
3. Locate the congruent black card in the deck.
4. The paired red card above this black card represents the ciphertext letter.
5. Switch the red ciphertext card with the red card that is on the top of the deck. If the ciphertext card is already on top of the deck, this step can be skipped.
6. Move the top two cards to the bottom of the deck.

"To demonstrate this process, we will encrypt `ATTACK AT DAWN` with Card-Chameleon using the following keyed deck.

$$[7\spadesuit, 3\diamondsuit, 6\spadesuit, K\heartsuit, 4\clubsuit, A\heartsuit, A\clubsuit, 3\heartsuit, 3\spadesuit, 10\diamondsuit, 10\clubsuit, Q\diamondsuit, 6\clubsuit,$$

$$A\diamondsuit, 9\clubsuit, 8\heartsuit, Q\spadesuit, 4\heartsuit, K\spadesuit, J\heartsuit, 3\clubsuit, 8\diamondsuit, Q\clubsuit, 7\heartsuit, A\spadesuit, 10\heartsuit,$$

$$J\spadesuit, 2\diamondsuit, 2\clubsuit, Q\heartsuit, 4\spadesuit, K\diamondsuit, 7\clubsuit, 9\heartsuit, 8\clubsuit, 5\heartsuit, 5\spadesuit, 6\heartsuit, 9\spadesuit,$$

$$6\diamondsuit, 2\spadesuit, 9\diamondsuit, 10\spadesuit, 7\diamondsuit, J\clubsuit, 2\heartsuit, 5\clubsuit, 4\diamondsuit, K\clubsuit, 5\diamondsuit, 8\spadesuit, J\diamondsuit]$$

"In order to encrypt the first `A`, Table 1 tells us to first find $A\spadesuit$. Second, we notice that above (i.e., after) $A\spadesuit$ is $10\heartsuit$. Third, $10\heartsuit$ and $10\spadesuit$ are a congruent pair. Thus, we now find

$10\spadesuit$. Fourth, the card that is above $10\spadesuit$ is $7\diamondsuit$, which represents `T`. Thus, `A` encrypts to `T`. Now we rearrange the deck. We switch $7\diamondsuit$ with the top (right-most) card, which is $J\diamondsuit$. Then we move the top two cards, $8\spadesuit$ and $7\diamondsuit$, to the bottom of the deck. Now the order of the deck is as follows.

$$[8\spadesuit, 7\diamondsuit, 7\spadesuit, 3\diamondsuit, 6\spadesuit, K\heartsuit, 4\clubsuit, A\heartsuit, A\clubsuit, 3\heartsuit, 3\spadesuit, 10\diamondsuit, 10\clubsuit,$$
$$Q\diamondsuit, 6\clubsuit, A\diamondsuit, 9\clubsuit, 8\heartsuit, Q\spadesuit, 4\heartsuit, K\spadesuit, J\heartsuit, 3\clubsuit, 8\diamondsuit, Q\clubsuit, 7\heartsuit,$$
$$A\spadesuit, 10\heartsuit, J\spadesuit, 2\diamondsuit, 2\clubsuit, Q\heartsuit, 4\spadesuit, K\diamondsuit, 7\clubsuit, 9\heartsuit, 8\clubsuit, 5\heartsuit, 5\spadesuit,$$
$$6\heartsuit, 9\spadesuit, 6\diamondsuit, 2\spadesuit, 9\diamondsuit, 10\spadesuit, J\diamondsuit, J\clubsuit, 2\heartsuit, 5\clubsuit, 4\diamondsuit, K\clubsuit, 5\diamondsuit]$$

"Next, we encrypt the letter `T`. First, we find $7\clubsuit$. Second, $9\heartsuit$ is above $7\clubsuit$. Third, since $9\heartsuit$ and $9\spadesuit$ are a congruent pair, we find $9\spadesuit$. Fourth, $6\diamondsuit$ is above $9\spadesuit$. Thus, the letter `T` encrypts to `S`. Again, we rearrange the deck. We switch $6\diamondsuit$ with the top card, which is $5\diamondsuit$. Then we move the top two cards, $K\clubsuit$ and $6\diamondsuit$, to the bottom. This gives us the following deck order.

$$[K\clubsuit, 6\diamondsuit, 8\spadesuit, 7\diamondsuit, 7\spadesuit, 3\diamondsuit, 6\spadesuit, K\heartsuit, 4\clubsuit, A\heartsuit, A\clubsuit, 3\heartsuit, 3\spadesuit,$$
$$10\diamondsuit, 10\clubsuit, Q\diamondsuit, 6\clubsuit, A\diamondsuit, 9\clubsuit, 8\heartsuit, Q\spadesuit, 4\heartsuit, K\spadesuit, J\heartsuit, 3\clubsuit, 8\diamondsuit,$$
$$Q\clubsuit, 7\heartsuit, A\spadesuit, 10\heartsuit, J\spadesuit, 2\diamondsuit, 2\clubsuit, Q\heartsuit, 4\spadesuit, K\diamondsuit, 7\clubsuit, 9\heartsuit, 8\clubsuit,$$
$$5\heartsuit, 5\spadesuit, 6\heartsuit, 9\spadesuit, 5\diamondsuit, 2\spadesuit, 9\diamondsuit, 10\spadesuit, J\diamondsuit, J\clubsuit, 2\heartsuit, 5\clubsuit, 4\diamondsuit]$$

"After repeating these steps for the rest of the message, we get the ciphertext: `TSRXYL YY SQLW`."

As discussed above, Card-Chameleon encrypts a letter to itself with probability $\frac{1}{13}$. The following is the proof of that result which is taken from "Determining Biases in the Card-

Chameleon Cryptosystem":

"The following results outline the main new attack on Card-Chameleon. With a perfect encryption scheme, like the One-Time Pad (OTP), a letter will encrypt to itself with probability $\frac{1}{26}$. However, we have the following.

Theorem A.1 *Assuming that the deck is in a random order for each letter, the probability that a letter in the plaintext will encrypt to itself is* $\frac{1}{13}$.

Proof. Without loss of generality, suppose we are encrypting the letter `A`. There are two types of keyed decks that will encrypt `A` to itself. If `A` encrypts to `A`, then in the encryption process, we first find the black card corresponding with `A`: $A♠$. Let R be the red card above $A♠$. Next, we find, B, the black card that is congruent to R. Finally, we must have $A\heartsuit$ above B, yielding the ciphertext letter `A`. Therefore, the deck contains the sequence $\{A♠, R, \ldots, B, A\heartsuit\}$ or $\{B, A\heartsuit, \ldots, A♠, R\}$. So either $R = A\heartsuit$ and $B = A♠$ or R and B are one of the other 25 congruent pairs.

Case 1: $R = A\heartsuit$ and $B = A♠$, so $A♠$ and $A\heartsuit$ are paired with each other. After $A\heartsuit$ is placed above $A♠$, these two cards can be placed in 26 places in the deck. The rest of the deck can now be permuted in $(25!)^2$ ways. Thus, there are $26 \cdot (25!)^2$ keyed decks and the probability of Case 1 occurring is $\frac{26 \cdot (25!)^2}{(26!)^2} = \frac{1}{26}$.

Case 2: $R \neq A\heartsuit$ and $B \neq A♠$, so $A\heartsuit$ and $A♠$ are not paired with each other, but they are paired with two cards that form a congruent pair. $A♠$ and R can be placed in 26 places in the deck. Next, $A\heartsuit$ and B can be placed in 25 places in the deck. Third, $A♠$ and $A\heartsuit$ can be paired with one of 25 congruent pairs. Finally, the rest of the deck can be permuted in $(24!)^2$ ways. Thus, there are $26 \cdot 25 \cdot 25 \cdot (24!)^2 = 26 \cdot (25!)^2$ keyed decksso the probability of Case 2 occurring is $\frac{26 \cdot (25!)^2}{(26!)^2} = \frac{1}{26}$.

Therefore, the probability of one of these decks occurring is $\frac{1}{26} + \frac{1}{26} = \frac{1}{13}$." ■

B C++ Code For VICCard 5.0

The following code executes the VICCard 5.0 cipher:

```
//  Author: Isaac Reiter

//  Creation Date: September 7, 2020

//  Purpose: This program executes the fifth version of VICCard.

//           VICCard 5.0 is a cipher that was created by Isaac

//           Reiter as part of his Capstone project for the

//           Honors Program of Kutztown University of Pennsylvania.

#include<iostream>

#include<string>

#include<fstream>

#include<iomanip>

#include<cstdlib>

#define MAX 1000 //This defines the maximum length of the plaintext

using namespace std;

//these functions prepare the keyed deck

void createDeck(int deck[52]);

void separateSuits(int deck[52], int spades[13], int hearts[13],

int clubs[13], int diamonds[13]);

void displayDeckOrder(int deck[52]);

void displaySeparatedSuits(int spades[13], int hearts[13],

int clubs[13], int diamonds[13]);

//this function adds junk letters to the beginning

//and end of the plaintext

void addJunk(string& plaintext);

//this function cuts the plaintext
```

```
void cutPlaintext(string& plaintext);

//these functions deal with the letter to card substitution

void firstPolybius(const int deck[52], int numDeck[52],

string plaintext, int firstPoly[2][MAX],

const string bitString, const char chaoStep);

int randomizer(int& cardNumber, const int index, const string bitString);

//these are the functions for the columnar transpositions

void doubleColumnarTranspo(int firstPoly[2][MAX], int& inUse,

const int clubs[13], const int hearts[13], const int rounds);

void firstFaceValueTranspo(int firstPoly[2][MAX], int& inUse,

const int clubs[13], const int rounds);

void secondFaceValueTranspo(int firstPoly[2][MAX], int& inUse,

const int hearts[13], const int rounds);

void firstSuitTranspo(int firstPoly[2][MAX], int& inUse,

const int hearts[13], const int rounds);

void secondSuitTranspo(int firstPoly[2][MAX], int& inUse,

const int clubs[13], const int rounds);

//these are the functions for the LFGs

void laggedFibonacciGenerators(int firstPoly[2][MAX], const int inUse,

const int spades[13], const int diamonds[13]);

void createMod13LFG(int Mod13LFG[MAX], const int spades[13],

const int inUse);

void createMod4LFG(int Mod4LFG[MAX], const int diamonds[13],

const int inUse);

void useLFG(int firstPoly[2][MAX], int Mod4LFG[MAX], int Mod13LFG[MAX],

const int inUse);

//this function deals with the card to letter substitution
```

```
void secondPolybius(const int deck[52], int numDeck[52], const int inUse,
string& ciphertext, int firstPoly[2][MAX], const char chaoStep);
//these function add the Chaocipher-like permutations
void chaoPerm(int letterDeck[52],int numDeck[52],const int locationOfCard);
void permPTDeck(int letterDeck[52], const int locationOfCard);
void permCTDeck(int numDeck[52], const int locationOfCard);
//these functions are used in various parts of the program
void searchDeck(const int deck[52], const int cardNumber,
int& locationOfCard);
int convertLetterToCardNumber(const char plainTextLetter);
char convertCardNumberToLetter(const int cardNumber);
int getFaceValue(const int cardNumber);
int getSuit(const int cardNumber);
int getCardNum(const int faceValue, const int suit);


int main(){
  int deck[52];
  int spades[13];
  int hearts[13];
  int clubs[13];
  int diamonds[13];
  string plaintext;
  int numRounds = 0;
  string bitString;
  char chaoStep;
  int letterDeck[52];
  int numDeck[52];
  int firstPoly[2][MAX];
  srand(time(NULL));
  cout<<"Please enter the plaintext (no spaces please):"<<endl;
  cin>>plaintext;
  cout<<endl;
  addJunk(plaintext);
  cutPlaintext(plaintext);
  int inUse = plaintext.length();
  string ciphertext;
  createDeck(deck);
  separateSuits(deck, spades, hearts, clubs, diamonds);
  //make sure that the deck is properly organized in the arrays
  displayDeckOrder(deck);
  displaySeparatedSuits(spades, hearts, clubs, diamonds);
  cout<<"How many rounds would you like?"<<endl;
  cin>>numRounds;
  cout<<endl<<"What random string of bits would you like? ";
  cout<<"Please enter a string of bits that is at least as long ";
  cout<<"as the plaintext."<<endl;
  cin>>bitString;
  if(bitString.length() < plaintext.length()){
    cout<<"Whoops! You entered too few bits."<<endl<<endl;
    exit(-1);}
  cout<<endl<<"Would you like a Chaocipher-like permutation? ";
  cout<<"Please type in 'y' for yes or 'n' for no."<<endl;
  cin>>chaoStep;
  if((chaoStep != 'y') && (chaoStep != 'n')){
    cout<<"I'm sorry. An invalid character was entered."<<endl;
    exit(-1);}
  for(int rounds = 0; rounds < numRounds; rounds++){
    //use the checkerboard to convert the plaintext into cards
    firstPolybius(deck, numDeck, plaintext, firstPoly, bitString, chaoStep);
    //execute the columnar tranposition
    doubleColumnarTranspo(firstPoly, inUse, clubs, hearts, rounds);
    //add the lagged Fibonacci generators
    laggedFibonacciGenerators(firstPoly, inUse, spades, diamonds);
    //use the checkerboard to convert the cards into the ciphertext
    secondPolybius(deck, numDeck, inUse, ciphertext, firstPoly, chaoStep);
    //prepare for the next round of encryption
    plaintext = ciphertext;
    ciphertext = "";
    bitString = "empty";}
  return 0;}

void createDeck(int deck[52]){
  ifstream deckFile;
  string fileName;
  cout<<"Please enter a .txt file containing the deck."<<endl;
  cin>>fileName;
  deckFile.open(fileName.c_str());
  if(deckFile.fail()){
    cout<<"Whoops! The program could not open that file."<<endl;
    exit(-1);}
  for(int index = 0; index < 52; index++){
    deckFile>>deck[index];}
  deckFile.close();
  return;}

void separateSuits(int deck[52], int spades[13], int hearts[13],
int clubs[13], int diamonds[13]){
  int spadeCount = 0;
  int heartCount = 0;
  int clubCount = 0;
  int diamondCount = 0;
  for(int index = 0; index < 52; index++){
    if((0 < deck[index]) && (deck[index] < 14)){
    spades[spadeCount] = deck[index];
    spadeCount++;}
else if((13 < deck[index]) && (deck[index] < 27)){
    clubs[clubCount] = deck[index];
    clubCount++;}
else if((26 < deck[index]) && (deck[index] < 40)){
    hearts[heartCount] = deck[index];
    heartCount++;}
else{
    diamonds[diamondCount] = deck[index];
    diamondCount++;}}
  return;}

void displayDeckOrder(int deck[52]){
  cout<<endl<<"deck order:"<<endl;
  for(int test = 0; test<52; test++){
    cout<<deck[test]<<" ";}
  cout<<endl<<endl;
  return;}

void displaySeparatedSuits(int spades[13], int hearts[13],
int clubs[13], int diamonds[13]){
  cout<<"Each suit:"<<endl;
  cout<<"  "<<"Spades:   ";
  for(int test = 0; test < 13; test++){
    cout<<spades[test]<<" ";}
  cout<<endl;
  cout<<"  "<<"Clubs:    ";
  for(int test = 0; test < 13; test++){
    cout<<clubs[test]<<" ";}
  cout<<endl;
  cout<<"  "<<"Hearts:   ";
  for(int test = 0; test < 13; test++){
    cout<<hearts[test]<<" ";}
  cout<<endl;
  cout<<"  "<<"Diamonds: ";
  for(int test = 0; test < 13; test++){
    cout<<diamonds[test]<<" ";}
  cout<<endl<<endl;
  return;}

void addJunk(string& plaintext){
  int numJunk;
  string newPlaintext = "";
  cout<<"How many junk letter would you like to appear at ";
  cout<<"the end of your message?"<<endl;
  cin>>numJunk;
  if(numJunk < 0){
    numJunk = 0;}
  for(int index = 0; index < numJunk; index++){
    plaintext+=(convertCardNumberToLetter((rand()%52)+1));}
  cout<<"New plaintext: "<<plaintext<<endl<<endl;
  cout<<"How many junk letter would you like to appear at ";
  cout<<"the beginning of your message?"<<endl;
  cin>>numJunk;
  if(numJunk < 0){
    numJunk = 0;}
  for(int index = 0; index < numJunk; index++){
    newPlaintext+=(convertCardNumberToLetter((rand()%52)+1));}
  newPlaintext+=plaintext;
  plaintext = newPlaintext;
  cout<<"New plaintext: "<<plaintext<<endl<<endl;
  return;}

void cutPlaintext(string& plaintext){
  int cutHere = 0;
  string newPlaintext = "";
  cout<<"If you would like me to cut the plaintext, type in the location ";
  cout<<"of the letter that you would like to appear at the beginning.";
  cout<<endl;
  cout<<"If you would not like to cut the plaintext, just type in 0.";
  cout<<endl;
  cin>>cutHere;
  if(cutHere < 0){
    cutHere = 0;}
  if(cutHere > 1){
    for(int index = cutHere-1; index < plaintext.length(); index++){
      newPlaintext+=plaintext.at(index);}
    for(int index = 0; index < cutHere-1; index++){
      newPlaintext+=plaintext.at(index);}
    plaintext = newPlaintext;}
  cout<<"New plaintext: "<<plaintext<<endl<<endl;
  return;}

void firstPolybius(const int deck[52], int numDeck[52], string plaintext,
int firstPoly[2][MAX], const string bitString, const char chaoStep){
  //first, put the deck into the checkerboard
  int letterDeck[52];
  int locationOfCard = 0;
  for(int index = 0; index < 52; index++){
    letterDeck[index] = deck[index];}
  //display the checkerboard
  cout<<endl<<"Letter Deck: ";
  for(int index = 0; index < 52; index++){
  cout<<left<<setw(2)<<letterDeck[index]<<" ";}
  cout<<endl<<endl;
```

```
//initialize numDeck
for(int index = 0; index < 13; index++){
  numDeck[index] = index + 14; //clubs
  numDeck[index+13] = index + 27; //hearts
  numDeck[index+26] = index + 1; //spades
  numDeck[index+39] = index + 40; //diamonds
  }
cout<<"Number Deck: ";
for(int index = 0; index < 52; index++){
cout<<left<<setw(2)<<numDeck[index]<<" ";}
cout<<endl<<endl;
//use the checkerboard to convert the plaintext into cards,
//which are then placed in firstPoly[2][MAX]
for(int index = 0; index < plaintext.length(); index++){
  int cardNumber = convertLetterToCardNumber(plaintext.at(index));
  //the randomizer is only executed during the first round
  if(bitString != "empty"){
    cardNumber = randomizer(cardNumber, index, bitString);}
//first, we must find the cardNumber in the grid
//int rowNum = 0;
//int colNum = 0;
searchDeck(letterDeck, cardNumber, locationOfCard);
firstPoly[0][index] = getFaceValue(numDeck[locationOfCard]);
firstPoly[1][index] = getSuit(numDeck[locationOfCard]);
  //perform Chaocipher-like permutation
  if(chaoStep == 'y'){
    chaoPerm(letterDeck, numDeck, locationOfCard);}}
  //display the results of the first Polybius
  cout<<"Results of Polybius Square (a face value of 0 corresponds ";
  cout<<"to the thirteenth row):"<<endl;
  for(int a=0; a < plaintext.length(); a++){
    cout<<left<<setw(3)<<firstPoly[0][a]<<" ";}
  cout<<endl;
  for(int a=0; a < plaintext.length(); a++){
    cout<<left<<setw(3)<<firstPoly[1][a]<<" ";}
  cout<<endl<<endl;
  return;}

int randomizer(int& cardNumber, const int index, const string bitString){
  //0 means black
  //1 means red
  char currentBit = bitString.at(index);
  if(currentBit == '0'){
    if(cardNumber > 26){
      cardNumber = cardNumber - 26;}}
  else{
    if(cardNumber < 27){
      cardNumber = cardNumber + 26;}}
  return cardNumber;}

void doubleColumnarTranspo(int firstPoly[2][MAX], int& inUse,
const int clubs[13], const int hearts[13], const int rounds){
  int inUseFaceValue = inUse;
  int inUseSuit = inUse;
```

```
firstFaceValueTranspo(firstPoly, inUseFaceValue, clubs, rounds);
cout<<"After first face value transpo:"<<endl;
for(int a=0; a < inUseFaceValue; a++){
  cout<<left<<setw(3)<<firstPoly[0][a]<<" ";}
cout<<endl;
for(int a=0; a < inUseSuit; a++){
  cout<<left<<setw(3)<<firstPoly[1][a]<<" ";}
cout<<endl<<endl;
secondFaceValueTranspo(firstPoly, inUseFaceValue, hearts, rounds);
cout<<"After second face value transpo:"<<endl;
for(int a=0; a < inUseFaceValue; a++){
  cout<<left<<setw(3)<<firstPoly[0][a]<<" ";}
cout<<endl;
for(int a=0; a < inUseSuit; a++){
  cout<<left<<setw(3)<<firstPoly[1][a]<<" ";}
cout<<endl<<endl;
firstSuitTranspo(firstPoly, inUseSuit, hearts, rounds);
cout<<"After first suit transpo:"<<endl;
for(int a=0; a < inUseFaceValue; a++){
  cout<<left<<setw(3)<<firstPoly[0][a]<<" ";}
cout<<endl;
for(int a=0; a < inUseSuit; a++){
  cout<<left<<setw(3)<<firstPoly[1][a]<<" ";}
cout<<endl<<endl;
secondSuitTranspo(firstPoly, inUseSuit, clubs, rounds);
cout<<"After second suit transpo:"<<endl;
for(int a=0; a < inUseFaceValue; a++){
    cout<<left<<setw(3)<<firstPoly[0][a]<<" ";}
  cout<<endl;
  for(int a=0; a < inUseSuit; a++){
    cout<<left<<setw(3)<<firstPoly[1][a]<<" ";}
  cout<<endl<<endl;
  inUse = inUseFaceValue;
  return;}

void firstFaceValueTranspo(int firstPoly[2][MAX], int& inUse,
const int clubs[13], const int rounds){
  //this columnar tranposition uses 7 columns
  int transpoGrid[MAX][7];
  int sevenClubs[7];
  int numClubs = 0;
  int numPlaced = 0;
  int row = 0;
  int column = 0;
  int traverseDownCol = 0;
  //the grid used for the transposition is filled
  while(numPlaced < inUse){
    transpoGrid[row][column] = firstPoly[0][numPlaced];
numPlaced++;
column++;
if(column == 7){
  column = 0;
  row++;}}
  //fill with junk on the first round
  if(rounds == 0){
    //fill with junk letters
while(column < 7){
      transpoGrid[row][column] = rand() % 13;
      column++;
      inUse++;}
    row++;}
  //on other rounds, always add 7 random numbers
  //for the seven-column transpo and 6 random numbers
  //for the six-column transpo
  else{
    //fill with seven random face values
    for(int index = 0; index < 7; index++){
      transpoGrid[row][column] = rand() % 13;
      column++;
      inUse++;
      if(column == 7){
      column = 0;
      row++;}}
    //if a row is partially filled,
while(column < 7){
      transpoGrid[row][column] = -1;
      column++; }
    row++;}
  //the next row is filled with -1 markers
  for(int index = 0; index < 7; index++){
    transpoGrid[row][index] = -1;}
```

```
cout<<"Face Values Before Cipher Block Chaining: ";
for(int rowIndex = 0; rowIndex < row; rowIndex++){
  for(int colIndex = 0; colIndex < 7; colIndex++){
    if(transpoGrid[rowIndex][colIndex] != -1){
      cout<<transpoGrid[rowIndex][colIndex]<<" ";}}}
cout<<endl<<endl;
//here, we add the output feedback mode / cipher block chaining
for(int rowIndex = 1; rowIndex < row; rowIndex++){
  for(int colIndex = 0; colIndex < 7; colIndex++){
    if(transpoGrid[rowIndex][colIndex] != -1){
transpoGrid[rowIndex][colIndex] =
(transpoGrid[rowIndex-1][colIndex]+transpoGrid[rowIndex][colIndex])%13;}}
}
cout<<"Face Values After Cipher Block Chaining: ";
for(int rowIndex = 0; rowIndex < row; rowIndex++){
  for(int colIndex = 0; colIndex < 7; colIndex++){
    if(transpoGrid[rowIndex][colIndex] != -1){
      cout<<transpoGrid[rowIndex][colIndex]<<" ";}}}
cout<<endl<<endl;
for(int index = 0; index < 13; index++){
  if(clubs[index] < 21){
    sevenClubs[numClubs] = clubs[index];
    numClubs++;}}
cout<<"First Columnar Transposition Grid for Face Values:"<<endl;
//display the columnar transposition grid without the -1's
for(int index = 0; index < 7; index++){
  cout<<right<<setw(2)<<sevenClubs[index]-13<<"  ";}
  cout<<endl;
  for(int rowIndex = 0; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 7; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        cout<<right<<setw(2)<<transpoGrid[rowIndex][colIndex]<<"  ";}}
    cout<<endl;}
  cout<<endl;
  //sevenClubs[7] is used to perform the transposition
  numPlaced = 0;
  for(int nth = 14; nth < 21; nth++){
int locationOfNth = 0;
    while(sevenClubs[locationOfNth] != nth){
  locationOfNth++;}
while(transpoGrid[traverseDownCol][locationOfNth] != -1){
  firstPoly[0][numPlaced] = transpoGrid[traverseDownCol][locationOfNth];
      numPlaced++;
      traverseDownCol++;}
    traverseDownCol = 0;}
  return;}

void secondFaceValueTranspo(int firstPoly[2][MAX], int& inUse,
const int hearts[13], const int rounds){
  int transpoGrid[MAX][6];
  int sixHearts[6];
  int numHearts = 0;
  int numPlaced = 0;
  int row = 0;
  int column = 0;

  int traverseDownCol = 0;

  int leftTriangles = 0;

  int junkToAdd = 0;

  for(int index = 0; index < 13; index++){

    if(hearts[index] < 33){

      sixHearts[numHearts] = hearts[index];

      numHearts++;}}

  //fill with -2 to prepare for the VIC-like transposition

  for(int nth = 27; nth < 33; nth++){

int locationOfNth = 0;

    while(sixHearts[locationOfNth] != nth){

  locationOfNth++;}

  for(int index = locationOfNth; index < 6; index++){

    for(int secondIndex = 0; secondIndex < index; secondIndex++){

      transpoGrid[row][secondIndex] = -3;}

    column = index;

    while(column < 6){

      transpoGrid[row][column] = -2;

      column++;}

    row++;}}

  cout<<"Triangular Sections for the Second Columnar Transposition ";

  cout<<"Grid for Face Values:"<<endl;

  //display the columnar transposition grid

  for(int index = 0; index < 6; index++){

    cout<<right<<setw(2)<<sixHearts[index]-26<<"  ";}

  cout<<endl;
```

```
for(int rowIndex = 0; rowIndex < row; rowIndex++){
  for(int colIndex = 0; colIndex < 6; colIndex++){
    if(transpoGrid[rowIndex][colIndex] != -1){
      if(transpoGrid[rowIndex][colIndex] == -3){
        cout<<right<<setw(2)<<" "<<"  ";}
      else{
        cout<<right<<setw(2)<<"T"<<"  ";}}}
  cout<<endl;}
cout<<endl;
//add junk face values
if(rounds == 0){
  junkToAdd = 6 - (inUse % 6);
  for(int index = 0; index < junkToAdd; index++){
    firstPoly[0][inUse + index] = rand() % 13;}
  inUse+=junkToAdd;}
else{
  for(int index = 0; index < 6; index++){
    firstPoly[0][inUse + index] = rand() % 13;}
  inUse+=6;}
cout<<"Face Values Before Cipher Block Chaining: ";
for(int index = 0; index < inUse; index++){
  cout<<firstPoly[0][index]<<" ";}
cout<<endl<<endl;
for(int index = 6; index < inUse; index++){
  firstPoly[0][index]=(firstPoly[0][index]+firstPoly[0][index-6])%13;}
cout<<"Face Values After Cipher Block Chaining: ";
for(int index = 0; index < inUse; index++){
    cout<<firstPoly[0][index]<<" ";}
  cout<<endl<<endl;
  row = 0;
  column = 0;
  //the grid used for the transposition is filled
  while(numPlaced < inUse){
    if(transpoGrid[row][column] != -2){
      transpoGrid[row][column] = firstPoly[0][leftTriangles];
      leftTriangles++;}
numPlaced++;
column++;
if(column == 6){
  column = 0;
  row++;}}
  row = 0;
  column = 0;
  numPlaced = leftTriangles;
  while(numPlaced < inUse){
    if(transpoGrid[row][column] == -2){
      transpoGrid[row][column] = firstPoly[0][numPlaced];
      numPlaced++;}
  column++;
  if(column == 6){
    column = 0;
    row++;}}
  //find where to begin placing the -1's
  row = (numPlaced - (numPlaced % 6)) / 6;
  column = numPlaced % 6;
  while((column > 0) && (column < 6)){
    transpoGrid[row][column] = -1;
    column++;
    if(column == 6){
      row++;}}
  //the next row is filled with -1 markers
  for(int index = 0; index < 6; index++){
    transpoGrid[row][index] = -1;}
  cout<<"Second Columnar Transposition Grid for Face Values:"<<endl;
  //display the columnar transposition grid
  for(int index = 0; index < 6; index++){
    cout<<right<<setw(2)<<sixHearts[index]-26<<"  ";}
  cout<<endl;
  for(int rowIndex = 0; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 6; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        cout<<right<<setw(2)<<transpoGrid[rowIndex][colIndex]<<"  ";}}
    cout<<endl;}
  cout<<endl;
  //the first six hearts are used to perform the transposition
  numPlaced = 0;
  for(int nth = 27; nth < 33; nth++){
int locationOfNth = 0;
    while(sixHearts[locationOfNth] != nth){
  locationOfNth++;}
  while(transpoGrid[traverseDownCol][locationOfNth] != -1){
    firstPoly[0][numPlaced] = transpoGrid[traverseDownCol][locationOfNth];
        numPlaced++;
        traverseDownCol++;}
    traverseDownCol = 0;}
  return;}

void firstSuitTranspo(int firstPoly[2][MAX], int& inUse,
const int hearts[13], const int rounds){
  //this columnar tranposition uses 7 columns
  int transpoGrid[MAX][7];
  int sevenHearts[7];
  int numHearts = 0;
  int numPlaced = 0;
  int row = 0;
  int column = 0;
  int traverseDownCol = 0;
  //the grid used for the transposition is filled
  while(numPlaced < inUse){
    transpoGrid[row][column] = firstPoly[1][numPlaced];
numPlaced++;
column++;
if(column == 7){
  column = 0;
  row++;}}
  //only fill with junk on the first round
  if(rounds == 0){
    //fill with junk letters
```

```
while(column < 7){
      transpoGrid[row][column] = rand() % 4;
      column++;
      inUse++;}
    row++;}
  else{
    //fill with seven random face values
    for(int index = 0; index < 7; index++){
      transpoGrid[row][column] = rand() % 4;
      column++;
      inUse++;
      if(column == 7){
    column = 0;
    row++;}}
    //if a row is partially filled,
while(column < 7){
      transpoGrid[row][column] = -1;
      column++; }
    row++; }
  //the next row is filled with -1 markers
  for(int index = 0; index < 7; index++){
    transpoGrid[row][index] = -1;}
  cout<<"Suits Before Cipher Block Chaining: ";
  for(int rowIndex = 0; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 7; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        cout<<transpoGrid[rowIndex][colIndex]<<" ";}}}
  cout<<endl<<endl;
  //here, we add the output feedback mode / cipher block chaining
  for(int rowIndex = 1; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 7; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        transpoGrid[rowIndex][colIndex] =
(transpoGrid[rowIndex-1][colIndex]+transpoGrid[rowIndex][colIndex])%4;
        }}}
  cout<<"Suits After Cipher Block Chaining: ";
  for(int rowIndex = 0; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 7; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        cout<<transpoGrid[rowIndex][colIndex]<<" ";}}}
  cout<<endl<<endl;
  for(int index = 0; index < 13; index++){
    if(hearts[index] > 32){
      sevenHearts[numHearts] = hearts[index];
      numHearts++;}}
  cout<<"First Columnar Transposition Grid for Suits:"<<endl;
  //display the columnar transposition grid
  for(int index = 0; index < 7; index++){
    cout<<right<<setw(2)<<sevenHearts[index]-26<<"  ";}
  cout<<endl;
  for(int rowIndex = 0; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 7; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        cout<<right<<setw(2)<<transpoGrid[rowIndex][colIndex]<<"  ";}}
    cout<<endl;}
  cout<<endl;
  //sevenHearts[7] is used to perform the transposition
  numPlaced = 0;
  for(int nth = 33; nth < 40; nth++){
int locationOfNth = 0;
    while(sevenHearts[locationOfNth] != nth){
  locationOfNth++;}
  while(transpoGrid[traverseDownCol][locationOfNth] != -1){
  firstPoly[1][numPlaced] = transpoGrid[traverseDownCol][locationOfNth];
      numPlaced++;
      traverseDownCol++;}
    traverseDownCol = 0;}
  return;}

void secondSuitTranspo(int firstPoly[2][MAX], int& inUse,
const int clubs[13], const int rounds){
  int transpoGrid[MAX][6];
  int sixClubs[6];
  int numClubs = 0;
  int numPlaced = 0;
  int row = 0;
  int column = 0;
  int traverseDownCol = 0;
  int leftTriangles = 0;
  int junkToAdd = 0;
  for(int index = 0; index < 13; index++){
    if(clubs[index] > 20){
      sixClubs[numClubs] = clubs[index];
      numClubs++;}}
  //fill with -2 to prepare for the VIC-like transposition
  for(int nth = 21; nth < 27; nth++){
int locationOfNth = 0;
    while(sixClubs[locationOfNth] != nth){
  locationOfNth++;}
    for(int index = locationOfNth; index < 6; index++){
      for(int secondIndex = 0; secondIndex < index; secondIndex++){
        transpoGrid[row][secondIndex] = -3;}
      column = index;
      while(column < 6){
        transpoGrid[row][column] = -2;
        column++;}
      row++;}}
  cout<<"Triangular Sections for the Second Columnar Transposition Grid ";
  cout<<"for Suits:"<<endl;
  //display the columnar transposition grid
  for(int index = 0; index < 6; index++){
    cout<<left<<setw(3)<<sixClubs[index]-13;}
  cout<<endl;
  for(int rowIndex = 0; rowIndex < row; rowIndex++){
    for(int colIndex = 0; colIndex < 6; colIndex++){
      if(transpoGrid[rowIndex][colIndex] != -1){
        if(transpoGrid[rowIndex][colIndex] == -3){
          cout<<left<<setw(3)<<" ";}
        else{
          cout<<left<<setw(3)<<"T";}}}
    cout<<endl;}
  cout<<endl;
  //add junk face values
  if(rounds == 0){
    junkToAdd = 6 - (inUse % 6);
    for(int index = 0; index < junkToAdd; index++){
      firstPoly[1][inUse + index] = rand() % 4;}
    inUse+=junkToAdd;}
  else{
    for(int index = 0; index < 6; index++){
      firstPoly[1][inUse + index] = rand() % 4;}
    inUse+=6;}
  cout<<"Suits Before Cipher Block Chaining: ";
  for(int index = 0; index < inUse; index++){
    cout<<firstPoly[1][index]<<" ";}
  cout<<endl<<endl;
  for(int index = 6; index < inUse; index++){
    firstPoly[1][index]=(firstPoly[1][index]+firstPoly[1][index-6])%4;}
  cout<<"Suits After Cipher Block Chaining: ";
  for(int index = 0; index < inUse; index++){
    cout<<firstPoly[1][index]<<" ";}
  cout<<endl<<endl;
  row = 0;
  column = 0;
  //the grid used for the transposition is filled
```

```
while(numPlaced < inUse){
  if(transpoGrid[row][column] != -2){
    transpoGrid[row][column] = firstPoly[1][leftTriangles];
    leftTriangles++;}
numPlaced++;
column++;
if(column == 6){
  column = 0;
  row++;}}
row = 0;
column = 0;
numPlaced = leftTriangles;
while(numPlaced < inUse){
  if(transpoGrid[row][column] == -2){
    transpoGrid[row][column] = firstPoly[1][numPlaced];
    numPlaced++;}
column++;
if(column == 6){
  column = 0;
  row++;}}
//find where to begin placing the -1's
row = (numPlaced - (numPlaced % 6)) / 6;
column = numPlaced % 6;
while((column > 0) && (column < 6)){
  transpoGrid[row][column] = -1;
  column++;
  if(column == 6){
      row++;}}
    //the next row is filled with -1 markers
    for(int index = 0; index < 6; index++){
      transpoGrid[row][index] = -1;}
    cout<<"Second Columnar Transposition Grid for Suits:"<<endl;
    //display the columnar transposition grid
    for(int index = 0; index < 6; index++){
      cout<<right<<setw(2)<<sixClubs[index]-13<<"  ";}
    cout<<endl;
    for(int rowIndex = 0; rowIndex < row; rowIndex++){
      for(int colIndex = 0; colIndex < 6; colIndex++){
        if(transpoGrid[rowIndex][colIndex] != -1){
          cout<<right<<setw(2)<<transpoGrid[rowIndex][colIndex]<<"  ";}}
      cout<<endl;}
    cout<<endl;
    //the last six clubs are used to perform the transposition
    numPlaced = 0;
    for(int nth = 21; nth < 27; nth++){
  int locationOfNth = 0;
      while(sixClubs[locationOfNth] != nth){
    locationOfNth++;}
  while(transpoGrid[traverseDownCol][locationOfNth] != -1){
    firstPoly[1][numPlaced] = transpoGrid[traverseDownCol][locationOfNth];
        numPlaced++;
        traverseDownCol++;}
      traverseDownCol = 0;}
    return;}
```

```
void laggedFibonacciGenerators(int firstPoly[2][MAX], const int inUse,
const int spades[13], const int diamonds[13]){
  int Mod13LFG[MAX];
  int Mod4LFG[MAX];
  int modifiedDiamonds[13];
  //convert diamonds to face value
  for(int index = 0; index < 13; index++){
    modifiedDiamonds[index] = diamonds[index] % 13;
    if(modifiedDiamonds[index] == 0){
      modifiedDiamonds[index] = 13;}}
  //convert face values to modulo 4
  for(int index = 0; index < 13; index++){
    modifiedDiamonds[index] = modifiedDiamonds[index] % 4;}
  createMod13LFG(Mod13LFG, spades, inUse);
  createMod4LFG(Mod4LFG, modifiedDiamonds, inUse);
  useLFG(firstPoly, Mod4LFG, Mod13LFG, inUse);
  cout<<"Lagged Fibonacci generators:"<<endl;
  for(int index = 0; index < inUse; index++){
    cout<<left<<setw(3)<<Mod13LFG[index]<<" ";}
  cout<<endl;
  for(int index = 0; index < inUse; index++){
    cout<<left<<setw(3)<<Mod4LFG[index]<<" ";}
  cout<<endl<<endl;
  cout<<"After lagged Fibonacci generators:"<<endl;
  for(int a=0; a < inUse; a++){
    cout<<left<<setw(3)<<firstPoly[0][a]<<" ";}
  cout<<endl;
  for(int a=0; a < inUse; a++){
    cout<<left<<setw(3)<<firstPoly[1][a]<<" ";}
  cout<<endl<<endl;
  return;}

void createMod13LFG(int Mod13LFG[MAX], const int spades[13],
const int inUse){
  for(int index = 0; index < 13; index++){
    Mod13LFG[index] = spades[index];}
  for(int index = 13; index < inUse; index++){
    Mod13LFG[index] = (Mod13LFG[index-13] + Mod13LFG[index-12]) % 13;}
  return;}

void createMod4LFG(int Mod4LFG[MAX], const int diamonds[13],
const int inUse){
  for(int index = 0; index < 13; index++){
    Mod4LFG[index] = diamonds[index];}
  for(int index = 13; index < inUse; index++){
    Mod4LFG[index] = (Mod4LFG[index-13] + Mod4LFG[index-12]) % 4;}
  return;}

void useLFG(int firstPoly[2][MAX], int Mod4LFG[MAX], int Mod13LFG[MAX],
const int inUse){
  for(int index = 0; index < inUse; index++){
    firstPoly[0][index] = (firstPoly[0][index] + Mod13LFG[index]) % 13;
    firstPoly[1][index] = (firstPoly[1][index] + Mod4LFG[index]) % 4;}
  return;}


void secondPolybius(const int deck[52], int numDeck[52], const int inUse,
string& ciphertext, int firstPoly[2][MAX], const char chaoStep){
  int letterDeck[52];
  int locationOfCard = 0;
  int cipherCardNum;
  for(int index = 0; index < 52; index++){
    letterDeck[index] = deck[index];}
  for(int index = 0; index < 13; index++){
    numDeck[index] = index + 14; //clubs
    numDeck[index+13] = index + 27; //hearts
    numDeck[index+26] = index + 1; //spades
    numDeck[index+39] = index + 40; //diamonds
    }
  for(int index = 0; index < inUse; index++){
    cipherCardNum = getCardNum(firstPoly[0][index], firstPoly[1][index]);
    searchDeck(numDeck, cipherCardNum, locationOfCard);
ciphertext += convertCardNumberToLetter(letterDeck[locationOfCard]);
    //perform Chaocipher-like permutation
    if(chaoStep == 'y'){
      chaoPerm(letterDeck, numDeck, locationOfCard);}}
  //display ciphertext
  cout<<"Ciphertext:"<<endl;
  cout<<ciphertext<<endl<<endl;
  return;}
```

```
void chaoPerm(int letterDeck[52],int numDeck[52],const int locationOfCard){
  permPTDeck(letterDeck, locationOfCard);
  permCTDeck(numDeck, locationOfCard);
  return;}

void permPTDeck(int letterDeck[52], const int locationOfCard){
  int placeHolder[52];
  if(locationOfCard != 0){
    int numInTopHalf = 52 - locationOfCard;
for(int index = locationOfCard; index < 52; index++){
  placeHolder[index-locationOfCard] = letterDeck[index];}
for(int index = (locationOfCard-1); index > -1 ; index--){
  letterDeck[index + numInTopHalf] = letterDeck[index];}
for(int index = 0; index < numInTopHalf; index++){
  letterDeck[index] = placeHolder[index];}}
  //the zenith+2 card
  int placeHolderSingle = letterDeck[49];
  for(int index = 49; index > 25; index--){
letterDeck[index] = letterDeck[index - 1];}
letterDeck[25] = placeHolderSingle;
  return;}

void permCTDeck(int numDeck[52], const int locationOfCard){
  int placeHolder[52];
  if(locationOfCard != 51){
    int numInTopHalf = 52 - (locationOfCard + 1);
for(int index = locationOfCard+1; index < 52; index++){
  placeHolder[index-(locationOfCard+1)] = numDeck[index];}
for(int index = locationOfCard; index > -1 ; index--){
  numDeck[index + numInTopHalf] = numDeck[index];}
for(int index = 0; index < numInTopHalf; index++){
  numDeck[index] = placeHolder[index];}}
  //the zenith+1 card
  int placeHolderSingle = numDeck[50];
  for(int index = 50; index > 25; index--){
    numDeck[index] = numDeck[index - 1];}
  numDeck[25] = placeHolderSingle;
  return;}

void searchDeck(const int deck[52], const int cardNumber,
int& locationOfCard){
  locationOfCard = 0;
  while(deck[locationOfCard] != cardNumber){
    locationOfCard++;}
  return;}

int convertLetterToCardNumber(const char plainTextLetter){
  string alphabet = "#abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
  return alphabet.find(plainTextLetter);}

char convertCardNumberToLetter(const int cardNumber){
  string alphabet = "aabcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
  return alphabet.at(cardNumber);}
```

```
int getFaceValue(const int cardNumber){
  int faceValue = cardNumber % 13;
  return faceValue;}

int getSuit(const int cardNumber){
  int suit;
  //club
  if((13 < cardNumber) && (cardNumber < 27)){
    suit = 0;}
  //heart
  else if((26 < cardNumber) && (cardNumber < 40)){
    suit = 1;}
  //spade
  else if((0 < cardNumber) && (cardNumber < 14)){
    suit = 2;}
  //diamondCount
  else{
    suit = 3;}
  return suit;}

int getCardNum(const int faceValue, const int suit){
  int cardNum;
  switch(suit){
    case 0:
    cardNum = 13;
    break;
    case 1:
```

```
  cardNum = 26;
  break;
  case 2:
  cardNum = 0;
  break;
  case 3:
  cardNum = 39;
  break;}
if(faceValue == 0){
  cardNum+=13;}
else{
  cardNum+=faceValue;}
return cardNum;}
```

C C++ Code Used To Generate Mod 4 LFG Seeds

I used the following code to generate all 1,201,200 possible seeds for the modulo 4 lagged Fibonacci generator:

```
//this program generates all 1,201,200 possible seeds
//for the modulo 4 LFG
#include<iostream>
#include<fstream>
#include<string>
#include<iomanip>    // This is needed for "setw"
#include<cstdlib>
#define FILENAME1 "FirstHalfOfSeeds.txt"
#define FILENAME2 "SecondHalfOfSeeds.txt"
using namespace std;
void iterateAllZerosTwosThreesAndOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2);
void iterateAllTwosThreesAndOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2);
void iterateAllThreesAndOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2);
void iterateAllOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2);
void outputSeed(int seedArray[13]);
void moveToEmpty(const int seedArray[13], int &indexToMove);


int main(){
  int seedArray[13];
  int numKeys = 0;
  ofstream seedFile1;
  ofstream seedFile2;
  seedFile1.open(FILENAME1);
  if(seedFile1.fail()){
    cout<<"Error opening file"<<endl;}
  seedFile2.open(FILENAME2);
  if(seedFile2.fail()){
    cout<<"Error opening file"<<endl;}
  //fill with -1
  for(int index = 0; index < 13; index++){
    seedArray[index] = -1;}
  iterateAllZerosTwosThreesAndOnes(seedArray,numKeys,seedFile1,seedFile2);
  cout<<"Number of keys: " <<numKeys<<endl;
  seedFile1.close();
  seedFile2.close();
  return 0;}

void iterateAllZerosTwosThreesAndOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2){
  int first0, second0, third0;
  for(first0 = 0; first0 < 11; first0++){
    seedArray[first0] = 0;
    for(second0 = first0 + 1; second0 < 12; second0++){
      seedArray[second0] = 0;
      for(third0 = second0 + 1; third0 < 13; third0++){
        seedArray[third0] = 0;
        iterateAllTwosThreesAndOnes(seedArray,numKeys,seedFile1,seedFile2);
        seedArray[third0] = -1;}
      seedArray[second0] = -1;}
    seedArray[first0] = -1;}
  return;}

void iterateAllTwosThreesAndOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2){
  int first2, second2, third2;
  for(first2 = 0; first2 < 11; first2++){
    moveToEmpty(seedArray, first2);
    seedArray[first2] = 2;
    for(second2 = first2 + 1; second2 < 12; second2++){
      moveToEmpty(seedArray, second2);
      seedArray[second2] = 2;
      for(third2 = second2 + 1; third2 < 13; third2++){
    moveToEmpty(seedArray, third2);
        if(third2 < 13){
          seedArray[third2] = 2;
          iterateAllThreesAndOnes(seedArray, numKeys, seedFile1, seedFile2);
          seedArray[third2] = -1;}}
      seedArray[second2] = -1;}
    seedArray[first2] = -1;}
  return;}

void iterateAllThreesAndOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2){
  int first3, second3, third3;
  for(first3 = 0; first3 < 11; first3++){
    moveToEmpty(seedArray, first3);
    seedArray[first3] = 3;
    for(second3 = first3 + 1; second3 < 12; second3++){
      moveToEmpty(seedArray, second3);
      seedArray[second3] = 3;
      for(third3 = second3 + 1; third3 < 13; third3++){
        moveToEmpty(seedArray, third3);
        if(third3 < 13){
          seedArray[third3] = 3;
          iterateAllOnes(seedArray, numKeys, seedFile1, seedFile2);
          seedArray[third3] = -1;}}
      seedArray[second3] = -1;}
    seedArray[first3] = -1;}
  return;}

void iterateAllOnes(int seedArray[13], int &numKeys,
ofstream &seedFile1, ofstream &seedFile2){
  int first1 = 0, second1, third1, fourth1;
  moveToEmpty(seedArray, first1);
  seedArray[first1] = 1;
  second1 = first1;
  moveToEmpty(seedArray, second1);
  seedArray[second1] = 1;
  third1 = second1;
  moveToEmpty(seedArray, third1);
  seedArray[third1] = 1;

  fourth1 = third1;

  moveToEmpty(seedArray, fourth1);

  seedArray[fourth1] = 1;

  outputSeed(seedArray);

  numKeys++;

  if(numKeys < 500001){

    for(int index = 0; index < 13; index++){

      seedFile1<<seedArray[index]<<" ";}

    seedFile1<<"\n";}

  else{

    for(int index = 0; index < 13; index++){

      seedFile2<<seedArray[index]<<" ";}

    seedFile2<<"\n";}

  //empty the filled elements

  seedArray[first1] = -1;

  seedArray[second1] = -1;

  seedArray[third1] = -1;

  seedArray[fourth1] = -1;

  return;}


void outputSeed(int seedArray[13]){

  for(int index = 0; index < 13; index++){

    cout<<seedArray[index]<<" ";}

  cout<<endl;

  return;}
```

```
void moveToEmpty(const int seedArray[13], int &indexToMove){
  int initial = indexToMove;
  if(indexToMove > 12){
    cout<<"An index was out of range"<<endl;
    exit(-1);}
  while((indexToMove < 13) && (seedArray[indexToMove] != -1)){
    indexToMove++;}
  return;}
```

D C++ Code Used To Generate Mod 13 LFG Seeds

I used the following code to generate select seeds for the modulo 13 lagged Fibonacci generator:

```
//this program generates selective seeds for the Mod 13 LFG
#include<iostream>
#include<fstream>
#include<string>
#include<iomanip>    // This is needed for "setw"
#define PERM 13
#define SELECTION 10
#define SOMESEEDS "EightFourTwelveFive.txt"
using namespace std;
void moveToEmpty(int permutation[PERM], int& locationOfNum);
void placeNum(int permutation[PERM], int theNumsToBePlaced[SELECTION],
int numToPlace, int& numPerm, ofstream& seedFile);


int main(){
  int permutation[PERM];
  int theNumsToBePlaced[SELECTION] = {0,1,2,3,6,7,9,10,11,0};
  int numToPlace = 1;
  int numPerm = 0;
  ofstream seedFile;
  for(int index = 0; index < PERM; index++){
    permutation[index] = -1;}
  seedFile.open(SOMESEEDS);
  if(seedFile.fail()){
    cout<<"Error opening file"<<endl;}
  permutation[0] = 8;
  permutation[1] = 4;
  permutation[2] = 12;
  permutation[3] = 5;
  placeNum(permutation, theNumsToBePlaced, numToPlace, numPerm, seedFile);
  cout<<"Number of permutations: "<<numPerm<<endl;
  seedFile.close();
  return 0;}

void placeNum(int permutation[PERM], int theNumsToBePlaced[SELECTION],
int numToPlace, int& numPerm, ofstream& seedFile){
  for(int locationOfNum = 0; locationOfNum < PERM; locationOfNum++){
    moveToEmpty(permutation, locationOfNum);
    if(locationOfNum < PERM){
      permutation[locationOfNum] = theNumsToBePlaced[numToPlace];
      if(numToPlace == (SELECTION-1)){
        numPerm++;
        for(int index = 0; index < PERM; index++){
          cout<<permutation[index]<<" ";}
        cout<<endl;
        for(int index = 0; index < PERM; index++){
          seedFile<<permutation[index]<<" ";}
        seedFile<<"\n";}
      else{
        placeNum(permutation, theNumsToBePlaced,
        (numToPlace+1), numPerm, seedFile);}
      permutation[locationOfNum] = -1;}}
  return;}

void moveToEmpty(int permutation[PERM], int& locationOfNum){
  while((locationOfNum < PERM) && (permutation[locationOfNum] != -1)){
    locationOfNum++;}
  return;}
```