ElsieFour: A Low-Tech Authenticated Encryption Algorithm For Human-to-Human Communication

Alan Kaminsky
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
ark@cs.rit.edu

Abstract. ElsieFour (LC4) is a low-tech cipher that can be computed by hand; but unlike many historical ciphers, LC4 is designed to be hard to break. LC4 is intended for encrypted communication between humans only, and therefore it encrypts and decrypts plaintexts and ciphertexts consisting only of the English letters A through Z plus a few other characters. LC4 uses a nonce in addition to the secret key, and requires that different messages use unique nonces. LC4 performs authenticated encryption, and optional header data can be included in the authentication. This paper defines the LC4 encryption and decryption algorithms, analyzes LC4's security, and describes a simple appliance for computing LC4 by hand.

1 Introduction

Before there were computers, there were ciphers. In historical times, monoalphabetic substitution ciphers, polyalphabetic substitution ciphers like the Vigenère cipher, and other simple ciphers like the Playfair cipher were computed by hand. In the mid-twentieth century cipher machines came into use, such as the World War II-era Enigma machine featured in the 2014 thriller *The Imitation Game*. By the end of the twentieth century, cipher algorithms that could be effectively computed only by digital hardware or computer software had become the norm, from DES to AES and beyond.

While modern ciphers are much more difficult to attack than historical ciphers, historical ciphers do have one distinct advantage over modern ciphers. No one tries to break a modern cipher deployed in the real world by attacking the cipher algorithm itself. Rather, attackers do side channel attacks, exploiting weaknesses of the cipher implementations in hardware or software. Alternatively, attackers install malware (such as a keyboard logger) into the computers running the cipher algorithms in order to capture the secret keys or plaintexts. In contrast, when a message is encrypted offline, by hand or with the aid of a simple (non-computerized) device, as is possible with a historical cipher, the encryption is not susceptible to a malware attack; and a "side channel" attack would require installing something like a spy camera to watch the human perform the encryption.

Of course, historical ciphers are typically easy to break. It would be interesting to devise a cipher that could be computed by hand or with the aid of a simple device, and that would be *hard* to break.

ElsieFour (abbreviated LC4) is an attempt at such a cipher.* LC4 is intended for encrypted communication between humans only, and therefore it encrypts and decrypts plaintexts and ciphertexts consisting only of the English letters A through Z plus a few other characters. LC4 uses a nonce in addition to the secret key, and requires that different messages use unique nonces. LC4 performs authenticated encryption, and optional header data can be included in the authentication.

LC4 is an amalgam of ideas from the classic RC4 stream cipher, the historical Playfair cipher, and the notion of plaintext-dependent keystreams. Like RC4, LC4 is based on a state that is continually updated as an encryption progresses. The chief differences between RC4 and LC4 are these:

- RC4's state is a permutation of the integers 0 through 255. LC4's state is a permutation of the integers 0 through 35.
- RC4's state is a one-dimensional array. LC4's state is a two-dimensional matrix, somewhat like the matrix Playfair uses.
- RC4's keystream is not plaintext-dependent, and so RC4 (by itself) does not do authenticated encryption. LC4 can be viewed as having a plaintext-dependent keystream, thus enabling it to do authenticated encryption.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the LC4 encryption and decryption algorithms. Section 4 analyzes the statistical characteristics of the LC4 ciphertexts and discusses how LC4 overcomes published weaknesses of RC4. Section 5 describes key recovery attacks on LC4 and estimates the work required to mount the attacks. Section 6 describes a simple appliance that aids in calculating LC4 encryptions and decryptions by hand.

2 Related Work

RC4. The well-known RC4 algorithm's state S consists of a permutation of the values 0 through $N - 1$, where $N = 256$, yielding an 8-bit word size. The RC4 key scheduling algorithm (KSA) initializes the state to a key-dependent pseudorandom permutation. The RC4 pseudorandom generation algorithm (PRGA) generates a pseudorandom keystream, a sequence of words. When encrypting, the keystream words are added to the plaintext words to yield the ciphertext words. When decrypting, the keystream words are subtracted from the ciphertext words to yield the plaintext words. The arithmetic is done in GF(2^8), so that addition and subtraction are just bitwise exclusive-or. However, a variant of RC4 could use a different word size, or could do the arithmetic in any finite group appropriate to the word size.

At each step of the RC4 KSA and PRGA, the state is altered by swapping the state words at indexes i and j. Index i increases by 1 at each step. Index j increases by $S[i]$ at each step; that is, pseudorandomly. The sequence of i and j values does not depend on the plaintext word at each step. Thus, RC4's keystream depends solely on the key and is independent of the plaintext (or ciphertext).

RC4 variations. Numerous variations of RC4 have been proposed to mitigate weaknesses of RC4, including Bowline [17], Py [2], Py6 [2], Pypy [3], RC4(n,m) [10,18], RC4A [21], RC4B [17], RC4+ [15], Sheet Bend [17], TPy [4], TPy6 [4], TPypy [4], and VMPC [28]. Each of these variations changes the KSA or the PRGA or both; some increase the word size from 8 bits to 32

* LC1, LC2, and LC3 were earlier unsuccessful attempts.

or 64 bits. None of these variations generates a plaintext-dependent keystream, and so none of these (by itself) does authenticated encryption.

Plaintext-dependent stream ciphers. Golić [9] first described how to combine a stream cipher algorithm with a message authentication code (MAC) algorithm by making the keystream depend on the secret key and on the plaintext, producing both a ciphertext and a MAC tag. Helix [8] and Phelix [27] are early examples of stream ciphers that generate plaintext-dependent keystreams. More recently, the duplex sponge construction [1] provides a general framework for combined encryption and authentication using plaintext-dependent keystreams.

Plaintext-dependent RC4-like stream ciphers have been proposed. Chameleon [17] treats the state as a substitution box (S-box) and encrypts each plaintext word p as $S[S[p]]$; then the state word at index $S[p]$ is swapped with the state word at index j, where j cycles through a fixed sequence of values chosen in a key-dependent fashion at initialization. Because the S-box mapping changes at each step, it is called a "mutating S-box." Chameleon needs to store both the length-N state S and the length-N cycle of values for index j; twice as much storage as RC4.

Spritz [23], another plaintext-dependent RC4-like cipher, is a variant of RC4 published in 2014 by Ronald Rivest, RC4's inventor, and Jacob Schuldt. According to them, "Spritz attempts to repair weak design decisions in RC4, while remaining true to its general design principles." The Spritz algorithm melds elements of the duplex sponge construction with elements of the original RC4 algorithm. Spritz is, however, considerably more complex than RC4.

Low-tech ciphers. Several ciphers that can be computed by hand or with the aid of a simple device have been published. These typically confine themselves to encrypting the English alphabet rather than arbitrary binary data.

Pocket-RC4 [17] and Solitaire [26] each use a deck of playing cards to generate a keystream, which is then added to the plaintext to get the ciphertext. Fortuitously, a standard card deck has exactly twice as many cards as there are English letters, so representing letters with cards is especially easy. However, neither Pocket-RC4 nor Solitaire generates a plaintext-dependent keystream.

Card-Chameleon [17] is similar to Chameleon, but it is confined to English letters, and it uses a deck of playing cards to implement the mutating S-box. Mirdek [7] also uses a deck of playing cards and uses a process reminiscent of Chameleon's mutating S-boxes to convert the plaintext to the ciphertext.

Chaocipher [6] was invented by John F. Byrne in 1918, but its algorithm was not revealed until 2010 [25]. It uses two permuted English alphabets, one for plaintext letters, the other for ciphertext letters. The encryption key is the initial permutation of both alphabets. To encrypt, a plaintext letter in the one alphabet is converted to the ciphertext letter at the same position in the other alphabet (or vice versa to decrypt), then both alphabets are shuffled. The procedure repeats for each letter. Chaocipher can be viewed as a stream cipher with a plaintext-dependent keystream. Byrne described a simple device, consisting of two interlocking alphabet wheels, for computing Chaocipher encryptions and decryptions.

Handycipher [12] is a homophonic monoalphabetic substitution cipher. It is computed with pen and paper using a matrix of letters in a manner vaguely reminiscent of the historical Playfair cipher. However, Handycipher is considerably more complex than Playfair.

Comparison with LC4. LC4 is a low-tech cipher intended to be computed by hand. Although the LC4 encryption and decryption algorithms described below do not generate a keystream per se, LC4 can be viewed as a stream cipher with a plaintext-dependent keystream. Of the ciphers

mentioned above, LC4 is most similar to Card-Chameleon and Chaocipher. However, LC4's state size is smaller than that of Card-Chameleon or Chaocipher; this makes LC4 easier to compute by hand, but somewhat more susceptible to the attacks described in Section 5.

3 LC4 Encryption and Decryption Algorithms

Alphabet. LC4 encrypts and decrypts plaintexts and ciphertexts consisting of these 36 characters (case insensitive):

`# _ 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z`

Internally, the encryption and decryption algorithms treat the characters `#` through `z` as integers 0 through 35, respectively. The character `#` is used, rather than the digit `0`, to avoid confusion with the uppercase letter `O`. The character _ is used, rather than the digit `1`, to avoid confusion with the uppercase letter `I` and the lowercase letter `l`. The `#` and _ characters can be used as separators in the plaintexts.

Key. The LC4 key is a permutation of the integers 0 through 35 chosen uniformly at random from the set of all such permutations. The size of the key space is 36!, equivalent to a 138.1-bit key. The key can be represented as a string, such as:

`_ie497g5oyhwqnazb3xsltcvfprd82u#6jmk`

The key is required to be a *random* permutation. If the key is based on a keyword, such as `goldameir...`, the key recovery attacks described in Section 5 can be sped up considerably by guessing dictionary words for the initial portion of the key.

State. The LC4 state consists of a 6×6 matrix *S* containing a permutation of the integers 0 through 35, as well as two indexes *i* and *j*. These indexes constitute a "marker" that designates a certain matrix element *S*[*i*][*j*] (the first index is the row, the second index is the column).

The state initialization algorithm is as follows. The "/" operation is truncating integer division. The "mod 6" operation returns a remainder in the range 0 to 5.

Input: Key array *K*
Output: State matrix *S;* indexes *i* and *j*
Algorithm:
    For *k* = 0 to 35:
        S[*k* / 6][*k* mod 6] ← *K*[*k*]
    i ← 0
    j ← 0

The state size (36 elements) is larger than the size of the English alphabet (26 letters) to make the key recovery attacks described in Section 5 more difficult. Consequently, the LC4 alphabet has to be augmented with 10 non-letter characters.

Basic encryption. The basic encryption algorithm is as follows. It assumes the state has been initialized. The plaintext characters are represented as integers in the range 0 to 35.

Inputs: State matrix *S;* indexes *i, j;* a sequence of plaintext characters, *P*
Output: A sequence of ciphertext characters, *C*
Algorithm:
    For each plaintext character *P:*
        r ← row of *S* in which *P* appears (0 ≤ *r* ≤ 5)
        c ← column of *S* in which *P* appears (0 ≤ *c* ≤ 5)

$x \leftarrow (r + (S[i][j] / 6)) \bmod 6$
$y \leftarrow (c + (S[i][j] \bmod 6)) \bmod 6$
$C \leftarrow S[x][y]$; output C
Right-rotate row r of S
Down-rotate column y of S
$i \leftarrow (i + (C / 6)) \bmod 6$
$j \leftarrow (j + (C \bmod 6)) \bmod 6$

The "Right-rotate row *r*" subroutine is:

$(S[r][0], S[r][1], S[r][2], S[r][3], S[r][4], S[r][5]) \leftarrow$
$(S[r][5], S[r][0], S[r][1], S[r][2], S[r][3], S[r][4])$
$c \leftarrow (c + 1) \bmod 6$
If $x = r$: $y \leftarrow (y + 1) \bmod 6$
If $i = r$: $j \leftarrow (j + 1) \bmod 6$

The "Down-rotate column *y*" subroutine is:

$(S[0][y], S[1][y], S[2][y], S[3][y], S[4][y], S[5][y]) \leftarrow$
$(S[5][y], S[0][y], S[1][y], S[2][y], S[3][y], S[4][y])$
$x \leftarrow (x + 1) \bmod 6$
If $c = y$: $r \leftarrow (r + 1) \bmod 6$
If $j = y$: $i \leftarrow (i + 1) \bmod 6$

Refer to Section 6 for the basic encryption algorithm expressed as a series of steps a human would perform.

At each step, the LC4 encryption algorithm goes to the position of a plaintext character *P* in the state matrix *S;* from there, it moves to the right and down by an amount dictated by the element of *S* under the marker; and it uses the resulting element of *S* as the ciphertext character *C*. The matrix row that contains *P* is rotated, and the matrix column that contains *C* is rotated; this plaintext-dependent (and ciphertext-dependent) state updating is what makes authenticated encryption possible. If the element under the marker is rotated, the marker is rotated along with that element. Finally, the marker moves to the right and down by an amount dictated by *C*. The state starts out as a random permutation; the elements under the marker change in a pseudorandom fashion as the encryption progresses; so the plaintext characters *P* are substituted with pseudorandomly chosen ciphertext characters *C*.

Basic decryption. The basic decryption algorithm is as follows. It assumes the state has been initialized. The ciphertext characters are represented as integers in the range 0 to 35.

Inputs: State matrix *S;* indexes *i, j;* a sequence of ciphertext characters, *C*
Output: A sequence of plaintext characters, *P*
Algorithm:
For each ciphertext character *C:*
$x \leftarrow$ row of S in which C appears $(0 \le x \le 5)$
$y \leftarrow$ column of S in which C appears $(0 \le y \le 5)$
$r \leftarrow (x - (S[i][j] / 6)) \bmod 6$
$c \leftarrow (y - (S[i][j] \bmod 6)) \bmod 6$
$P \leftarrow S[r][c]$; output P
Right-rotate row r of S
Down-rotate column y of S

$i \leftarrow (i + (C / 6)) \bmod 6$
$j \leftarrow (j + (C \bmod 6)) \bmod 6$

Refer to Section 6 for the basic decryption algorithm expressed as a series of steps a human would perform.

The LC4 decryption algorithm works the same way as the encryption algorithm, except it moves backwards in the state matrix *S* from a ciphertext character *C* to a plaintext character *P*. The decryption algorithm updates *S* in the same way as the encryption algorithm, so the sequence of values for *S* is the same for both encryption and decryption.

LC4 can be viewed as a stream cipher where the keystream is the sequence of differences $(C - P) \bmod 36$, although the differences are not calculated explicitly.

Message encryption and authentication. To send an encrypted and authenticated message using LC4, first assemble the following items:

- Key as described above.
- Nonce; a sequence of 6 or more characters, each chosen uniformly at random from the alphabet. The nonce is required to be different for each message.
- Header data to be included in the authentication; a sequence of zero or more characters.
- Plaintext of the message.
- Signature; a sequence of 10 or more characters that uniquely identifies the sender.

Then perform the following steps. Note that the state is not altered from one step to the next.

- Initialize the state using the key.
- Perform the basic encryption algorithm on the nonce; discard the ciphertext.
- Perform the basic encryption algorithm on the header data if any; discard the ciphertext.
- Perform the basic encryption algorithm on the concatenation of the plaintext plus the signature.

Send the unencrypted nonce followed by the ciphertext from the last step. See Appendix A for an example.

To decrypt and authenticate a received message, perform the following steps. Note that the state is not altered from one step to the next.

- Initialize the state using the key.
- Perform the basic encryption algorithm on the nonce; discard the ciphertext.
- Perform the basic encryption algorithm on the header data if any; discard the ciphertext. (The recipient is assumed to know what the header data is.)
- Perform the basic decryption algorithm on the ciphertext, yielding the concatenation of the plaintext plus the signature.
- Verify that the decrypted signature is that of the purported sender; if not, reject the message.

The nonce serves the usual purpose of generating a unique keystream for each message while using the same secret key for multiple messages. The minimum nonce length of 6 is chosen to compensate for nonrandom biases in the initial ciphertext positions, as discussed in Section 4. Nonce collisions must also be considered. With 36^6 possible nonce values, a collision between randomly chosen nonces is expected to occur after about 36^3, or about 46 thousand, encryptions. Before that, the two communicating parties should switch to a different key. A longer nonce would allow more messages to be encrypted before encountering a nonce collision, but would require more time to perform an encryption or decryption.

Here the signature is not a "digital signature;" rather, it is simply a string uniquely associated with the sender. The signature is assumed to be secret, known only to the communicating parties, as is the key. Because the LC4 keystream is plaintext-dependent, if the nonce or the ciphertext is altered during transmission, with high probability the decrypted signature will not match the sender's secret signature, and the authentication will fail. The same is true if an attacker tries to forge a message. Section 4 analyzes the probability of a false positive authentication, where the received transmission is incorrect due to alteration or forgery but the decrypted signature is correct. The minimum signature length of 10 is chosen to make the false positive authentication probability acceptably small, as discussed in Section 4.

4 Statistical Analysis of LC4

Numerous weaknesses have been identified in RC4. Because LC4 is similar to RC4 at a high level—both are based on a permutation of a limited set of values, both update the permutation as the algorithm progresses—LC4 might have weaknesses similar to RC4's. This section analyzes the statistical properties of LC4 and shows how LC4 overcomes various weaknesses of RC4.

Initial state weaknesses. After executing the RC4 key scheduling algorithm, ideally the initial RC4 state should be a permutation of the integers 0 through 255 chosen uniformly at random from the set of all such permutations, and different keys should result in different initial states. However, several investigators showed that certain initial states (permutations) occur with higher probabilities than others [20,24]. Other investigators identified "key collisions," where different keys result in the same initial state [5].

LC4 avoids these weaknesses by equating the initial state to the key and requiring the key to be a random permuted alphabet. Such keys are easily generated—for example, by drawing tiles from a bag, or by going to the Random.org web site (with precautions to prevent eavesdropping).

Keystream biases. Ideally, the RC4 pseudorandom generation algorithm should generate keystreams such that for each keystream word, every possible value from 0 to 255 occurs with an equal probability of 1/256. However, several investigators showed that the keystream is biased. In the first few keystream words, the values do not all appear with equal probabilities [16,19,24]. The same is true of certain later keystream words [16]. Others identified correlations between the keystream bytes and the initial RC4 state after the PRGA [11,19].

A nonce would mitigate these initial keystream biases in RC4. By encrypting a nonce of sufficient length before the actual plaintext and discarding the encrypted nonce, initial keystream biases would not affect the encryption of the plaintext. LC4 might exhibit initial keystream biases too; if so, LC4 would require a nonce to mitigate these biases.

In addition, LC4 requires a nonce to yield a different ciphertext if the same plaintext is encrypted with the same key. Ideally, the ciphertexts produced by encrypting a certain plaintext with the same key but different nonces should be random strings, where each ciphertext character obeys a discrete uniform distribution (all values 0 through 35 equally likely).

I wrote a program to test the uniformity of each position in LC4's ciphertexts. The program did R repetitions. For each repetition, the program chose a key and chose a plaintext of length P. The key was a permutation of the alphabet chosen uniformly at random from the set of all such permutations. The letters of the plaintext were chosen at random using the digram frequencies of English text. I obtained these by measuring the digram frequencies in the 25 most popular books from Project Gutenberg (www.gutenberg.org). For each repetition, the program did T trials. For each trial, the program chose a nonce of length N and encrypted the plaintext using the key and

the nonce. Each character of the nonce was chosen uniformly at random from the alphabet. For each position in the ciphertext, the program counted occurrences of each character and did an *odds ratio uniformity test* for that position. (See Appendix B for a description of the odds ratio uniformity test.) The test yielded the *log odds ratio* for the hypothesis that the characters were uniformly distributed; the test *passed* if the log odds ratio was greater than 0; the test *failed* if the log odds ratio was less than 0. The program did the same for the digrams starting at each ciphertext position. Finally, the program aggregated the log odds ratios for the ciphertext characters and digrams across the repetitions.

I ran the program with repetitions $R = 100$, plaintext length $P = 100$, trials $T = 1000$, and various nonce lengths $N = 1$ to 12. Table 1 shows the uniformity test's aggregate log odds ratios for characters and digrams for the first ten ciphertext positions for $N = 1$ to 4.* For nonce lengths of 1 or 2, the test detected nonrandom behavior (negative log odds ratios) in the initial ciphertext characters, digrams, or both. For nonce lengths of 3 or more, no nonrandom behavior was detected in the ciphertext characters and digrams at any positions (all log odds ratios were positive). Although a minimum nonce length of 3 would suffice to avert nonrandom ciphertexts, I recommend a minimum nonce length of 6 when using LC4, as discussed in Section 3.

Table 1. Ciphertext uniformity test log odds ratios versus nonce length

	$N = 1$			$N = 2$			$N = 3$			$N = 4$	
i	*Chars.*	*Digrams*	*i*	*Chars.*	*Digrams*	*i*	*Chars.*	*Digrams*	*i*	*Chars.*	*Digrams*
0	38.929	6.8460	0	22.237	–54.596	0	169.84	128.60	0	170.69	141.27
1	5.8638	–28.552	1	8.1955	–61.897	1	134.03	99.798	1	156.14	119.78
2	12.589	–13.048	2	–9.2257	–63.556	2	159.44	123.34	2	174.50	141.11
3	–7.4612	–39.577	3	24.633	–51.255	3	150.90	115.66	3	163.51	125.93
4	14.561	–10.308	4	65.473	12.352	4	153.11	127.38	4	158.55	121.62
5	2.4371	–29.043	5	34.299	–14.196	5	168.86	134.70	5	182.67	151.89
6	7.2546	–20.661	6	79.740	21.658	6	141.78	109.68	6	177.70	144.24
7	13.891	–11.948	7	37.900	–31.183	7	135.02	91.997	7	188.79	159.74
8	13.157	–15.809	8	30.617	–34.970	8	189.35	158.93	8	158.08	124.93
9	5.1913	–29.011	9	3.1972	–51.218	9	155.72	111.56	9	146.49	104.72

Ciphertext randomness. Historical ciphers encrypting English plaintexts, like the monoalphabetic ciphers used in newspaper cryptograms, merely substitute letters or digrams with other letters or digrams but do not alter the nonuniform distribution of the letters and digrams in English text. Consequently, such ciphers are easily broken by a ciphertext-only attack based on those nonuniform distributions: the most frequent ciphertext letter is probably the encryption of E; the next most frequent, of T; and so on. To avert such attacks, a strong cipher should "flatten" these nonuniform letter and digram distributions, yielding ciphertexts whose letters and digrams are uniformly distributed.

I wrote a program to test the uniformity of the characters and digrams in LC4's ciphertexts. The program was similar to the preceding program, except the program tested the uniformity of the characters and digrams over the entire ciphertext rather than separately for each position. The program did R repetitions. For each repetition, the program chose a random key and did T trials. For each trial, the program chose a random nonce of length N; chose a random plaintext of length P using the digram frequencies of English text; and encrypted the plaintext using the key

* Complete results, and the programs' Java source code, for all the test programs in this paper are available at (https://www.cs.rit.edu/~ark/parallelcrypto/elsiefour/).

and the nonce. (This simulates what a pair of humans would do to communicate using LC4: establish a key, then encrypt a series of plaintexts with the same key and different nonces.) The program counted occurrences of each character in the ciphertext and did an odds ratio uniformity test. The program did the same for the ciphertext digrams. Finally, the program aggregated the log odds ratios for the ciphertext characters and digrams across the trials for each repetition, and printed all these aggregate log odds ratios.

I ran the program with repetitions R = 100, plaintext length P = 100, trials T = 1000, and nonce lengths N = 6 through 12. The ciphertext character aggregate log odds ratios ranged from 419.29 to 629.09 for the different nonce lengths. The ciphertext digram aggregate log odds ratios ranged from 24.536 to 280.65 for the different nonce lengths. Because all the aggregate log odds ratios were greater than 0, I conclude that LC4's ciphertext characters and digrams are indeed uniformly distributed—despite the nonuniform distribution of the plaintext letters and digrams.

False positive authentication. Ideally, if an attacker alters an LC4-encrypted message during transit by changing one or more characters in the nonce or the ciphertext, the decrypted signature at the end of the message should not match the sender's signature, thus alerting the recipient to the alteration. However, there is a chance that an altered message might still yield the correct signature after decryption—a *false positive authentication.*

I wrote a program to estimate the probability of a false positive authentication in LC4. The program did R repetitions. For each repetition, the program chose a random key; chose a random nonce of length N; chose a random plaintext of length P using the digram frequencies of English text, the signature being the final S characters of the plaintext; and encrypted the message yielding a ciphertext. The program then made every possible alteration of one character in either the nonce or the ciphertext; decrypted the result; checked whether the decrypted signature matched the original signature; and counted the number of such false positive authentications. This number divided by the total number of alterations is an estimate of the false positive authentication probability. The total number of alterations was $R \cdot 35 \cdot (N + P)$—in each repetition, 35 alterations in each nonce and ciphertext position.

I ran the program with repetitions R = 1000000, nonce length N = 10, various signature lengths S from 4 to 30, and plaintext length $P = 100 + S$. Table 2 shows the observed number of false positive authentications and the probability thereof versus the signature length S. As expected, the false positive authentication probability decreases as the signature length increases. The data shows that for signatures of length 10 or more under these conditions, the probability of a false positive authentication is less than one in 100 million. I therefore recommend, more or less arbitrarily, a minimum signature length of 10 characters when using LC4.

Table 2. False positive authentication probability versus signature length

S	*Number*	*Probability*	*S*	*Number*	*Probability*	*S*	*Number*	*Probability*
4	4461	1.1587×10^{-6}	13	9	2.1609×10^{-9}	22	0	0.0
5	868	2.2342×10^{-7}	14	2	4.7619×10^{-10}	23	0	0.0
6	306	7.8061×10^{-8}	15	1	2.3613×10^{-10}	24	0	0.0
7	147	3.7168×10^{-8}	16	1	2.3419×10^{-10}	25	0	0.0
8	65	1.6291×10^{-8}	17	0	0.0	26	0	0.0
9	42	1.0435×10^{-8}	18	1	2.3041×10^{-10}	27	0	0.0
10	23	5.6650×10^{-9}	19	1	2.2857×10^{-10}	28	0	0.0
11	14	3.4188×10^{-9}	20	0	0.0	29	0	0.0
12	6	1.4528×10^{-9}	21	0	0.0	30	0	0.0

However, because LC4 is intended for human-to-human communication, there is another way to detect a message altered during transit, without even needing to check a signature. Humans would use LC4 to send natural language messages to each other, not arbitrary data. If a ciphertext is altered, it is likely that the resulting decrypted plaintext will contain misspelled words, ungrammatical phrases, or random gibberish, which a human could spot. For example:

Key:	`7ehtkb59cmvxy4zf2jd83rug_np6#owqilsa`
Original ciphertext:	`t4ui8b_9dpv6xzgat6hh2oy3nbq5q6wr7wfa`
Altered ciphertext:	`t4ui8b_9dpv6xzgbt6hh2oy3nbq5q6wr7wfa`
Decryption thereof:	`in_the_beginninnuhix67rzb7#xdyo5ssvu`

Without even knowing the original plaintext or the correct signature, it's obvious that the ciphertext message must have been altered.

Message forgery. An attacker wishing to forge an LC4-encrypted message, without knowing the (secret) key and signature of the legitimate sender, is forced to guess the key and the signature. When the recipient decrypts the forged message using the legitimate key rather than the key the attacker guessed, the decrypted signature ends up as a random string. The probability that this random string matches the legitimate signature is $1/36^S$, where S is the signature length. For a signature of length 10, the probability of a successful forgery is about $2^{-51.7}$.

5 Key Recovery Attacks on LC4

Ciphertext-only attack. Many historical ciphers, including monoalphabetic and polyalphabetic substitution ciphers, when encrypting natural language messages that have nonuniform distributions of letter frequencies, yield ciphertexts with likewise nonuniform distributions of letter frequencies. An attacker, possessing only the ciphertext of a message, can take advantage of these nonuniform distributions to deduce the plaintext. As shown in Section 4, however, LC4 yields ciphertexts with a uniform random distribution of character frequencies, so this kind of attack will not work on LC4.

Consequently, to carry out a ciphertext-only attack, the attacker has to do a generic brute force key search: Decrypt the ciphertext with every possible key until it yields an intelligible plaintext. This would require 36!, or $2^{138.1}$, decryptions in the worst case, an impractical effort.

Known plaintext attack. If the attacker knows both the plaintext and the ciphertext of an LC4 encrypted message, but not the key, the attacker can do a simple backtracking search to recover the key. Starting from an initial LC4 state where every state element is unknown, the search guesses the value of the first state element (the one under the marker); uses the first plaintext character and the first ciphertext character to guess additional state elements; updates the marker; and repeats the process for subsequent plaintext and ciphertext characters. If the search encounters a conflict, where the actual known ciphertext character differs from the one appearing in the state, the search backtracks to the most recent guess and tries a different guess. Once the search has reached the end of the plaintext and ciphertext, the search takes the final state and runs the encryption process backwards to recover the initial state, that is, the key.

Whenever the search encounters a conflict and backtracks, the search in effect eliminates an entire subset of the key space. The earlier the conflict, the larger the subset eliminated. Thus, the backtracking key search should take less time than a brute force key search, which looks at the entire key space.

I wrote a program to explore the effort required to do a backtracking key search on LC4. The program did a number of repetitions. For each repetition, the program generated a random key; generated a random nonce of length 6; generated a random plaintext of length *P* using the digram frequencies of English text; and encrypted the nonce and plaintext to get a ciphertext. The program then "forgot" the final *X* characters of the key and did a backtracking search to recover the missing key characters. The program reported the number of conflicts encountered, which indicates the amount of effort needed for the attack.

I ran the program with 11 repetitions, plaintext length $P = 200$, and $X = 1$ to 28. Figure 1 shows the results. The black dots plot *Y*, the number of conflicts, versus *X*, the number of missing key characters, for each repetition. The vertical axis is logarithmic, so that an exponential function of *X* would show up as a straight line. The blue line plots the median number of conflicts versus *X*. The trend of the median line shows that the key recovery attack's effort is a superexponential function of the number of missing key characters: the median line is not straight, but curves upwards. This is as expected; in the worst case, the attack would have to search all possible permutations of the *X* missing key characters, that is, *X*! permutations, a superexponential function of *X*.

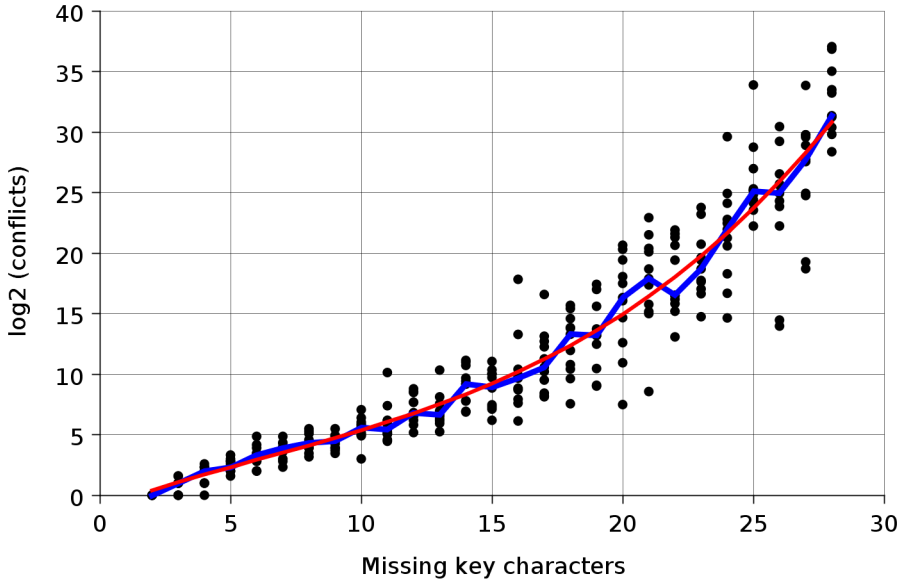


Figure 1. Key recovery attack effort versus number of missing key characters

I did not have the patience to run the key recovery attack program for $X = 36$ to recover a completely unknown key. Instead, I did a least squares fit of the median data to this model:

$$\log_2(\text{median}(Y)) = a + bX + cX^2 + dX^3$$

The fitted model parameters were $a = -1.0548$, $b = 0.77999$, $c = -0.028372$, and $d = 0.0014715$. The red line in Figure 1 plots the fitted model. This model is not based on theoretical considerations; out of several simple ad hoc models, this was the one that yielded the closest match to the median data. I then extrapolated the model to $X = 36$, yielding $\log_2(\text{median}(Y)) = 58.9$. Assuming the attack's actual effort continues to match the model as *X* increases, a full key recovery attack on LC4 would take about $2^{58.9}$ work. This is probably more effort than anyone would want to spend to recover the key for two parties communicating with LC4-encrypted messages.

It should be emphasized, though, that the key recovery attack requires knowing both the plaintext and the ciphertext of an encrypted message. The attacker can easily retrieve the ciphertext if it is sent through the Internet. The LC4 encryption, however, is computed offline by hand, and the plaintext is presumably destroyed after sending the ciphertext. Under these circumstances, it is difficult to imagine how the attacker could retrieve the plaintext and carry out the attack.

6 Computing LC4 By Hand

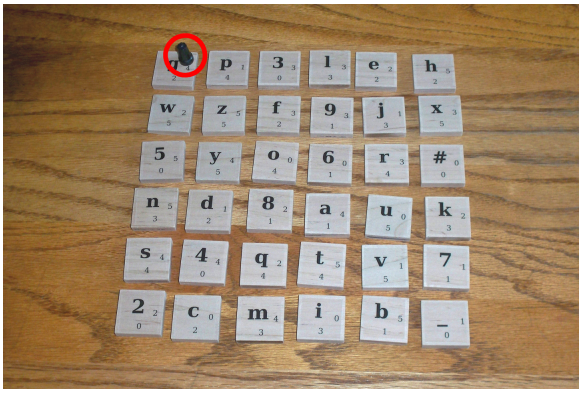
To prevent computer-based surveillance of humans exchanging LC4-encrypted messages, the LC4 encryption and decryption algorithms are intended to be computed offline by hand, not online by software. After encrypting a message offline, the sender enters the nonce and the ciphertext into a computer for transmission. Upon receiving these, the recipient decrypts the message offline. The keys, plaintexts, and signatures never appear in any computer or network message.

I made a simple appliance to help a human perform the LC4 encryption and decryption algorithms by hand. The appliance consists of 36 wooden tiles plus a small plastic game marker, which I carry around in a bag (Figure 2). Each tile is marked at the top with one of the characters of the alphabet; at the right, the value (c mod 6); and at the bottom, the value (c / 6), where c is the character's integer value. The bag also is used to generate a random key: put all the tiles in the bag, shake it well, and draw tiles one by one.

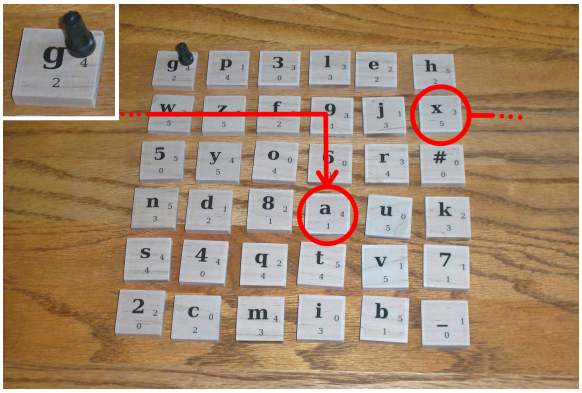
Figure 2. The LC4 appliance; closeup of tiles

Starting on the next page are the steps a human would perform to encrypt a message using the LC4 appliance.

Arrange the tiles in a 6×6 matrix according to the key. Put the marker on the upper left tile.

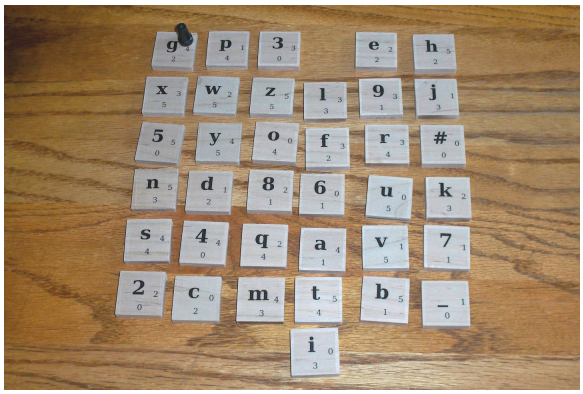


Find the plaintext character in the matrix (in this example, x). In that row, count to the right the number of tiles shown at the right side of the tile under the marker (4), wrapping around to the beginning of the row if necessary. In that column, count down the number of tiles shown at the bottom of the tile under the marker (2), wrapping around to the beginning of the column if necessary. The resulting tile is the ciphertext character (a).

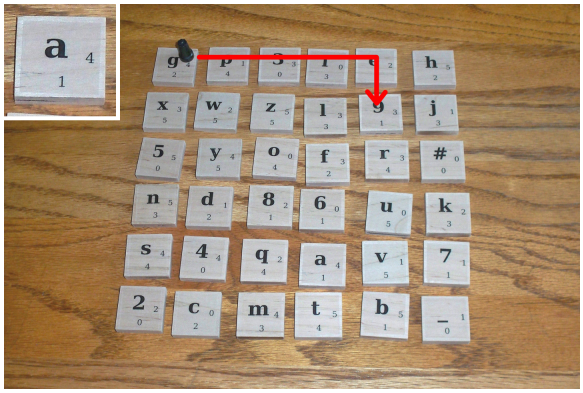


In the row with the plaintext character, shift the tiles one position right, and put the rightmost tile at the beginning of the row. If the marker's tile moves, the marker stays on that tile.

In the column with the ciphertext character, shift the tiles one position down, and put the bottom-most tile at the beginning of the column. If the marker's tile moves, the marker stays on that tile.



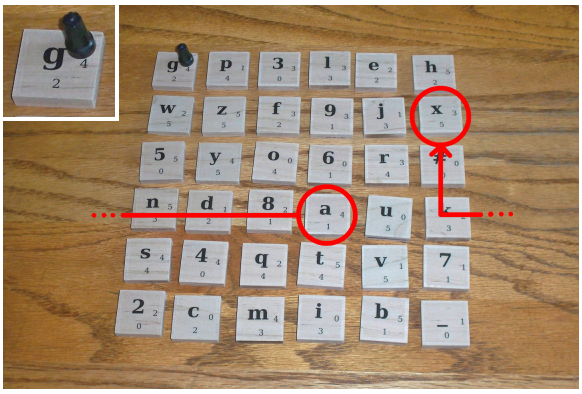
Move the marker to the right the number of tiles shown at the right side of the ciphertext tile (4), wrapping around to the beginning of the row if necessary. Move the marker down the number of tiles shown at the bottom of the ciphertext tile (1), wrapping around to the beginning of the column if necessary.



Repeat these steps to encrypt additional plaintext characters.

Decryption is the same as encryption, except the plaintext character is derived from the ciphertext character as follows:

Find the ciphertext character in the matrix (in this example, `a`). In that row, count to the left the number of tiles shown at the right side of the tile under the marker (4), wrapping around to the end of the row if necessary. In that column, count up the number of tiles shown at the bottom of the tile under the marker (2), wrapping around to the end of the column if necessary. The resulting tile is the plaintext character (`x`).



References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. *18th International Workshop on Selected Areas in Cryptography (SAC 2011)*, pages 320–337, 2011.

[2] E. Biham and J. Seberry. Py: a fast and secure stream cipher using rolling arrays. April 29, 2005. http://www.ecrypt.eu.org/stream/p2ciphers/py/py_p2.ps, retrieved February 2, 2015.

[3] E. Biham and J. Seberry. Pypy: another version of Py. June 27, 2006. http://www.ecrypt.eu.org/stream/p2ciphers/py/pypy_p2.ps, retrieved February 2, 2015.

[4] E. Biham and J. Seberry. Tweaking the IV setup of the Py family of stream ciphers—the ciphers TPy, TPypy, and TPy6. January 27, 2007. http://www.ecrypt.eu.org/stream /papersdir/2007/038.ps, retrieved February 2, 2015.

[5] E. Biham and O. Dunkelman. Differential cryptanalysis in stream ciphers. Cryptology ePrint Archive, Report 2007/218, June 6, 2007.

[6] J. Byrne. *Silent Years: An Autobiography with Memoirs of James Joyce and Our Ireland*. Farrar, Straus, and Young, 1953.

[7] P. Crowley. Mirdek: a card cipher inspired by “Solitaire.” January 13, 2000. http://www.ciphergoth.org/crypto/mirdek/, retrieved February 2, 2015.

[8] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno. Helix: fast encryption and authentication in a single cryptographic primitive. *10th International Workshop on Fast Software Encryption (FSE 2003)*, pages 330–346, 2003.

[9] J. Golić. Modes of operation of stream ciphers. *7th International Workshop on Selected Areas in Cryptography (SAC 2000)*, pages 233–247, 2000.

[10] G. Gong, K. Gupta, M. Hell, and Y. Nawaz. Towards a general RC4-like keystream generator. *First SKLOIS Conference on Information Security and Cryptology (CISC 2005)*, pages 162–174, 2005.

[11] R. Jenkins. ISAAC and RC4. 1996. http://burtleburtle.net/bob/rand/isaac.html, retrieved February 16, 2015.

[12] B. Kallick. Handycipher: a low-tech, randomized, symmetric-key cryptosystem. Cryptology ePrint Archive, Report 2014/257, January 19, 2015.

[13] R. Kass and A. Raftery. Bayes factors. *Journal of the American Statistical Association* 90:773–795, 1995.

[14] A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography,* 48(3):269–286, September 2008.

[15] S. Maitra and G. Paul. Analysis of RC4 and proposal of additional layers for better security margin. *9th International Conference on Cryptology in India (INDOCRYPT 2008),* pages 27–39, 2008.

[16] S. Maitra and G. Paul. New form of permutation bias and secret key leakage in keystream bytes of RC4. *15th International Workshop on Fast Software Encryption (FSE 2008),* pages 253–269, 2008.

[17] M. McKague. Design and analysis of RC4-like stream ciphers. University of Waterloo M.S. thesis, 2005. https://uwspace.uwaterloo.ca/bitstream/handle/10012/1141 /memckagu2005.pdf, retrieved February 2, 2015.

[18] Y. Nawaz, K. Gupta, and G. Gong. A 32-bit RC4-like keystream generator. Cryptology ePrint Archive, Report 2005/175, June 12, 2005.

[19] G. Paul, S. Rathi, and S. Maitra. On non-negligible bias of the first output byte of RC4 towards the first three bytes of the secret key. *International Workshop on Coding and Cryptography (WCC),* pages 285–294, 2007.

[20] G. Paul and S. Maitra. *RC4 Stream Cipher and Its Variants.* CRC Press, 2012.

[21] S. Paul and B. Preneel. A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. *11th International Workshop on Fast Software Encryption (FSE 2004),* pages 245–259, 2004.

[22] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition.* Cambridge University Press, 2007.

[23] R. Rivest and J. Schuldt. Spritz—a spongy RC4-link stream cipher and hash function. October 27, 2014. http://people.csail.mit.edu/rivest/pubs/RS14.pdf, retrieved February 2, 2015.

[24] A. Roos. A class of weak keys in the RC4 stream cipher. Posted in sci.crypt.research newsgroup, September 22, 1995. https://groups.google.com/forum /#!forum/sci.crypt.research, retrieved February 11, 2015.

[25] M. Rubin. Chaocipher revealed: the algorithm. July 2, 2010. http://www.mountainvista soft.com/chaocipher/ActualChaocipher/Chaocipher-Revealed-Algorithm.pdf, retrieved April 7, 2015.

[26] B. Schneier. The Solitaire encryption algorithm. May 26, 1999. https://www.schneier.com/solitaire.html, retrieved February 2, 2015.

[27] D. Whiting, B. Schneier, S. Lucks, and F. Muller. Phelix—fast encryption and authentication in a single cryptographic primitive. April 29, 2005. http://www.ecrypt.eu.org /stream/ciphers/phelix/phelix.pdf, retrieved February 2, 2015.

[28] B. Zoltak. VMPC one-way function and stream cipher. *11th International Workshop on Fast Software Encryption (FSE 2004),* pages 210–225, 2004.

A Encryption Example

Here is a trace of LC4 encrypting the message “I’m about to put the hammer down.” The trace shows the state array *S*, the indexes *i* and *j*, the plaintext character, and the ciphertext character at the start of the encryption and at the end of each loop iteration. To save space, the rows of the state are printed on one line.

Key: `xv7ydq#opaj_39rzut8b45wcsgehmiknf26l`
Nonce: `solwbf`
Header: (none)
Message: `im_about_to_put_the_hammer_down`
Signature: `#rubberduck`

```
State                                    i  j  pt ct
xv7ydq #opaj_ 39rzut 8b45wc sgehmi knf26l  0  0
xv72dq #opyj_ 39raut 8b4zwc isg5hm knfe6l  0  5  s  5
xv7edq _#o2yj 39rput 8b4awc isgzhm knf56l  2  1  o  e
xvnedq _#72yj 39oput 8brawc is4zhm lkgf56  3  2  l  7
xknedq _v72yj 3#oput c9braw i84zhm lsgf56  3  3  w  #
xkned6 _v72yq 3#opuj wc9brt i84zha lsgf5m  0  5  b  j
xkngd6 _v7eyq 3#o2uj wc9prt i84bha mlszf5  2  1  f  e
xlngd6 _k7eyq 3vo2uj w#9prt ac84bh miszf5  0  1  i  i
xlnsd6 _k7gyq 3voeuj w#92rt ac8pbh 5mi4zf  0  3  m  2
xlnsz6 q_k7dy 3voegj w#92ut ac8prh 5mi4bf  5  2  _  z
5lnsz6 x_k7dy qvoegj 3#92ut wac8pr hmi4bf  3  4  a  q
5lns46 x_k7zy qvoedj 3#92gt wac8ur fhmipb  2  5  b  p
5lni46 x_kszy jqv7ed 3#9ogt wac2ur fhm8pb  5  0  o  i
5hni46 xlkszy j_v7ed 3q9ogt r#ac2u fwm8pb  2  3  u  l
fhni46 5lkszy x_v7ed j3q9og t#ac2u rwm8pb  0  0  t  r
fhnip6 5lks4y dx_vze j3q97g t#acou rwm82b  0  2  _  2
fhnipb 5lks46 dx_vzy j3q97e ut#acg rwm82o  5  0  t  y
fhwipb 5lns46 dxkvzy j3_97e utqacg or#m82  3  3  o  q
fhwip2 5lns4b dxkvz6 ej3_9y utqac7 or#m8g  5  2  _  g
2fhwig 5lns4p dxkvzb ej3_96 utqacy or#m87  3  3  p  p
2f#wig 5lhs4p dxnvzb ejk_96 yu3qac ortm87  1  2  u  t
2o#wig 5fhs4p dlnvzb exk_96 yj3qac 7urtm8  4  5  t  l
2o#tig 5fhw4p dlnszb 6exv_9 yj3kac 7urqm8  2  4  _  t
g2r#ti 5fow4p dlhszb 6env_9 yjxkac 7u3qm8  0  1  t  r
g2r#t8 5fow4i bdlhsp 6env_z yjxka9 7u3qmc  5  0  h  z
g23#t8 5frw4i bdohsp z6lnv_ yjeka9 7uxqmc  4  3  e  x
gu3#t8 52rw4i bfohsp _d6lnv yzeka9 7jxqmc  4  5  _  2
7u3#t8 g2rw4i 5bfohs pd6lnv _zeka9 yjxqmc  4  0  h  _
yu3#t8 72rw4i gbfohs 5d6lnv p_zeka 9jxqmc  5  4  a  9
yuj#t8 723w4i gbrohs 5dflnv p_6eka c9zxqm  1  2  m  f
yujzt8 723#4i gbrwhs 5dfonv p_6lka mc9exq  0  1  m  z
yuj9t8 723z4i gbr#hs 5dfwnv p_6oka qmclex  3  4  e  l
ymj9t8 7u3z4i s2br#h 5gfwnv pd6oka q_clex  0  2  r  m
ym_9t8 7ujz4i s23r#h 5gbwnv pdfoka xq6cle  2  1  _  b
ym_9l8 7ujzti s23r4h 5gbw#v apdfnk xq6coe  0  1  d  o
ymq9l8 7u_zti s2jr4h 5g3w#v apbfnk exd6co  0  4  o  3
emq9l8 yu_zti 72jr4h s5g3w# vpbfnk axd6co  5  2  w  y
emq9lo yu_zt8 72jr4i s5g3wh kvpbf# axd6cn  0  4  n  8
emd9lo yuqzt8 72_r4i s5j3wh #kgpbf axv6cn  0  5  #  _
emd6lo yuq9t8 i72zr4 s5j_wh #kg3bf axvpcn  1  2  r  9
```

```
emdplo 8yu69t i72qr4 s5jzwh #kg_bf axv3cn  0  4  u  p
exdplo 8mu69t iy2qr4 s7jzwh f5kg_b a#v3cn  5  2  b  y
axdplo emu69t 8y2qr4 i7jzwh sf5kg_ b#v3cn  3  0  b  s
bxdplo aemu69 ty2qr4 87jzwh if5kg_ s#v3cn  2  4  e  s
b#dplo axmu69 4ey2qr 8tjzwh i75kg_ sfv3cn  1  2  r  x
sb#dpl oxmu69 aey2qr 4tjzwh 875kg_ ifv3cn  2  4  d  8
sb#dpn 9oxmul aey2q6 4tjzwr 875kgh ifv3c_  5  3  u  n
sbfdpn 9o#mul aex2q6 4tyzwr 87jkgh _i5v3c  1  1  c  f
sbfvpn 9o#dul aexmq6 4ty2wr h87zkg _i5j3c  1  3  k  2
```

Ciphertext: `i2zqpilr2yqgptltrzx2_9fzlmbo3y8_9pyssx8nf2`

B Odds Ratio Uniformity Test

The odds ratio uniformity test is an alternative to frequentist statistical tests such as the chi-square test. A strong point of the odds ratio uniformity test is that the results of multiple independent tests can easily be *aggregated* to yield a single overall result. The odds ratio uniformity test uses the methodology of *Bayesian model selection* applied to binomial distributions. For more information about Bayesian model selection, see [13].

Bayes factors and odds ratios. Let H denote a *hypothesis*, or *model*, describing some process. Let D denote an experimental *data sample*, or just *sample*, observed by running the process. Let $\mathrm{pr}(H)$ be the probability of the model. Let $\mathrm{pr}(D|H)$ be the conditional probability of the sample given the model. Let $\mathrm{pr}(D)$ be the probability of the sample, apart from any particular model. Bayes's Theorem states that $\mathrm{pr}(H|D)$, the conditional probability of the model given the sample, is

$$\mathrm{pr}(H\,|\,D) \;=\; \frac{\mathrm{pr}(D\,|\,H)\;\mathrm{pr}(H)}{\mathrm{pr}(D)} \;. \tag{B.1}$$

Suppose there are two alternative models H_1 and H_2 that could describe a process. After observing sample D, the *posterior odds ratio* of the two models, $\mathrm{pr}(H_1|D)/\mathrm{pr}(H_2|D)$, is calculated from Equation (B.1) as

$$\frac{\mathrm{pr}(H_1|D)}{\mathrm{pr}(H_2|D)} \;=\; \frac{\mathrm{pr}(D|H_1)}{\mathrm{pr}(D|H_2)} \cdot \frac{\mathrm{pr}(H_1)}{\mathrm{pr}(H_2)} \;, \tag{B.2}$$

where the term $\mathrm{pr}(H_1)/\mathrm{pr}(H_2)$ is the *prior odds ratio* of the two models, and the term $\mathrm{pr}(D|H_1)/\mathrm{pr}(D|H_2)$ is the *Bayes factor*. The odds ratio represents one's belief about the relative probabilities of the two models. Given one's initial belief before observing any samples (the prior odds ratio), the Bayes factor is used to *update* one's belief after performing an experiment and observing a sample (the posterior odds ratio). Stated simply, posterior odds ratio = Bayes factor × prior odds ratio.

Suppose two experiments are performed and two samples, D_1 and D_2, are observed. Assuming the samples are independent, it is straightforward to calculate that the posterior odds ratio based on both samples is

$$\begin{aligned} \frac{\mathrm{pr}(H_1|D_2,D_1)}{\mathrm{pr}(H_2|D_2,D_1)} &= \frac{\mathrm{pr}(D_2|H_1)}{\mathrm{pr}(D_2|H_2)} \cdot \frac{\mathrm{pr}(H_1|D_1)}{\mathrm{pr}(H_2|D_1)} \\ &= \frac{\mathrm{pr}(D_2|H_1)}{\mathrm{pr}(D_2|H_2)} \cdot \frac{\mathrm{pr}(D_1|H_1)}{\mathrm{pr}(D_1|H_2)} \cdot \frac{\mathrm{pr}(H_1)}{\mathrm{pr}(H_2)} \;. \end{aligned} \tag{B.3}$$

In other words, the posterior odds ratio for the first experiment becomes the prior odds ratio for the second experiment. Equation (B.3) can be extended to any number of independent samples D_i; the final posterior odds ratio is just the initial prior odds ratio multiplied by all the samples' Bayes factors.

Model selection is the problem of deciding which model, H_1 or H_2, is better supported by a series of one or more samples D_i. In the Bayesian framework, this is determined by the posterior odds ratio (B.3). Henceforth, "odds ratio" will mean the posterior odds ratio. If the odds ratio is greater than 1, then H_1's probability is greater than H_2's probability, given the data; that is, the data supports H_1 better than it supports H_2. The larger the odds ratio, the higher the degree of support. An odds ratio of 100 or more is generally considered to indicate decisive support for H_1 [13]. If on the other hand the odds ratio is less than 1, then the data supports H_2 rather than H_1, and an odds ratio of 0.01 or less indicates decisive support for H_2.

Models with parameters. In the preceding formulas, the models had no free parameters. Now suppose that model H_1 has a parameter θ_1 and model H_2 has a parameter θ_2. Then the conditional probabilities of the samples given each of the models are obtained by integrating over the possible parameter values [13]:

$$\mathrm{pr}(D|H_1) \;=\; \int \mathrm{pr}(D|\theta_1,H_1)\;\pi(\theta_1|H_1)\;\mathrm{d}\,\theta_1 \;, \tag{B.4}$$

$$\mathrm{pr}(D|H_2) \;=\; \int \mathrm{pr}(D|\theta_2,H_2)\;\pi(\theta_2|H_2)\;\mathrm{d}\,\theta_2 \;, \tag{B.5}$$

where $\mathrm{pr}(D|\theta_1,H_1)$ is the probability of observing the sample under model H_1 with the parameter value θ_1, $\pi(\theta_1|H_1)$ is the prior probability density of θ_1 under model H_1, and likewise for H_2 and θ_2. The Bayes factor is then the ratio of these two integrals.

Odds ratio for binomial models. Suppose an experiment performs n Bernoulli trials, where the probability of success is θ, and counts the number of successes k, which obeys a binomial distribution. The values n and k constitute the sample D. With this as the model H, the probability of D given H with parameter θ is

$$\mathrm{pr}(D|H,\theta) \;=\; \binom{n}{k}\,\theta^{k}\,(1-\theta)^{n-k} \;=\; \frac{n!}{k!\,(n-k)!}\,\theta^{k}\,(1-\theta)^{n-k} \;. \tag{B.6}$$

Consider the odds ratio for two particular binomial models, H_1 and H_2. H_1 is that the Bernoulli success probability θ_1 is a certain value p, the value that the success probability is "supposed" to have. Then the prior probability density of θ_1 is a delta function, $\pi(\theta_1|H_1) = \delta(\theta_1 - p)$, and the Bayes factor numerator (B.4) becomes

$$\mathrm{pr}(D|H_1) \;=\; \frac{n!}{k!\,(n-k)!}\,p^{k}\,(1-p)^{n-k} \;. \tag{B.7}$$

H_2 is that the Bernoulli success probability θ_2 is some unknown value between 0 and 1, not necessarily the value it is "supposed" to have. The prior probability density of θ_2 is taken to be a uniform distribution: $\pi(\theta_2|H_2) = 1$ for $0 \le \theta_2 \le 1$ and $\pi(\theta_2|H_2) = 0$ otherwise. The Bayes factor denominator (B.5) becomes

$$\mathrm{pr}(D|H_2) \;=\; \int_0^1 \frac{n!}{k!\,(n-k)!}\,\theta_2^{k}\,(1-\theta_2)^{n-k}\,\mathrm{d}\,\theta_2 \;=\; \frac{1}{n+1} \;. \tag{B.8}$$

Putting everything together, the Bayes factor for the two binomial models is

$$\frac{\mathrm{pr}(D|H_1)}{\mathrm{pr}(D|H_2)} = \frac{(n+1)!}{k!\,(n-k)!}\, p^k\,(1-p)^{n-k} \quad . \tag{B.9}$$

Substituting the gamma function for the factorial, $n! = \Gamma(n+1)$, gives

$$\frac{\mathrm{pr}(D|H_1)}{\mathrm{pr}(D|H_2)} = \frac{\Gamma(n+2)}{\Gamma(k+1)\,\Gamma(n-k+1)}\, p^k\,(1-p)^{n-k} \quad . \tag{B.10}$$

Because the gamma function's value typically overflows the range of floating point values in a computer program, we compute the logarithm of the Bayes factor instead of the Bayes factor itself:

$$\log\frac{\mathrm{pr}(D|H_1)}{\mathrm{pr}(D|H_2)} = \begin{array}{l}\log\Gamma(n+2) - \log\Gamma(k+1) - \log\Gamma(n-k+1) + \\ k\log p + (n-k)\log(1-p)\end{array} \quad . \tag{B.11}$$

The log-gamma function can be computed efficiently (see [22] page 256), and mathematical software libraries usually include log-gamma.

Odds ratio test. The above experiment can be viewed as a *test* of whether H_1 is true, that is, whether the success probability is p. The log (posterior) odds ratio of the models H_1 and H_2 is the log prior odds ratio plus the log Bayes factor (B.11). Assuming that H_1 and H_2 are equally probable at the start, the log odds ratio is just the log Bayes factor. The test *passes* if the log odds ratio is greater than 0, otherwise the test *fails.*

When multiple independent runs of the above experiment are performed, the overall log odds ratio is the sum of all the log Bayes factors. In this way, one can *aggregate* the results of a series of individual tests, yielding an overall odds ratio test. Again, the aggregate test passes if the overall log odds ratio is greater than 0, otherwise the aggregate test fails.

Note that the odds ratio test is not a frequentist statistical test that is attempting to disprove some null hypothesis. The odds ratio test is just a particular way to decide how likely or unlikely it is that a series of observations came from a Bernoulli(p) distribution, by calculating a posterior odds ratio. While a frequentist statistical test could be defined based on odds ratios, I am not doing that here.

Odds ratio uniformity test. Consider a random variable X with a *discrete uniform distribution.* The variable has B different possible values ("bins"), $0 \le x \le B-1$. An experiment with n trials is performed. In each trial, the random variable's value is observed, and a counter for the corresponding bin is incremented. If the variable obeys a discrete uniform distribution, all the counters should end up the same.

The *odds ratio uniformity test* calculates the odds ratio of two hypotheses: H_1, that X obeys a discrete uniform distribution, and H_2, that X does not obey a discrete uniform distribution. To do so, first calculate the observed cumulative distribution of X and the expected cumulative distribution of X under model H_1. The observed cumulative distribution is

$$F_{\mathrm{obs}}(x) = \sum_{i=0}^{x} \mathrm{counter}[x] \;,\; 0 \le x \le B-1, \tag{B.12}$$

and the expected cumulative distribution is

$$F_{\text{exp}}(x) = \frac{(x+1)n}{B}\ ,\ 0 \le x \le B-1. \tag{B.13}$$

Let y be the bin such that the absolute difference $|F_{\text{obs}}(y) - F_{\text{exp}}(y)|$ is maximized.* The trials are now viewed as Bernoulli trials, where incrementing a bin less than or equal to y is a success, the observed number of successes in n trials is $k = F_{\text{obs}}(y)$, and the success probability is $p = F_{\text{exp}}(y)/n = (y + 1)/B$ if H_1 is true. An odds ratio test for a discrete uniform distribution (H_1 versus H_2) is therefore equivalent to an odds ratio test for this particular binomial distribution, with Equation (B.11) giving the log Bayes factor. If the log Bayes factor is greater than 0, then X obeys a discrete uniform distribution, otherwise X does not obey a discrete uniform distribution.

A couple of examples will illustrate the odds ratio uniformity test. I queried a pseudorandom number generator one million times; each value was uniformly distributed in the range 0.0 (inclusive) through 1.0 (exclusive); I multiplied the value by 10 and truncated to an integer, yielding a bin x in the range 0 through 9; and I accumulated the values into 10 bins, yielding this data:

x	counter[x]	$F_{\text{obs}}(x)$	$F_{\text{exp}}(x)$	$\lvert F_{\text{obs}}(x) - F_{\text{exp}}(x)\rvert$
0	99476	99476	100000	524
1	100498	199974	200000	26
2	99806	299780	300000	220
3	99881	399661	400000	339
4	99840	499501	500000	499
5	99999	599500	600000	500
6	99917	699417	700000	583
7	100165	799582	800000	418
8	100190	899772	900000	228
9	100228	1000000	1000000	0

The maximum absolute difference between the observed and expected cumulative distributions occurred at bin 6. With n = 1000000, k = 699417, and p = 0.7, the log Bayes factor is 5.9596. In other words, the odds are about exp(5.9596) = 387 to 1 that this data came from a discrete uniformly distributed random variable.

I queried a pseudorandom number generator one million times again, but this time I raised each value to the power 1.01 before converting it to a bin. This introduced a slight bias towards smaller bins. I got this data:

x	counter[x]	$F_{\text{obs}}(x)$	$F_{\text{exp}}(x)$	$\lvert F_{\text{obs}}(x) - F_{\text{exp}}(x)\rvert$
0	101675	101675	100000	1675
1	101555	203230	200000	3230
2	100130	303360	300000	3360
3	99948	403308	400000	3308
4	99754	503062	500000	3062
5	99467	602529	600000	2529
6	99355	701884	700000	1884
7	99504	801388	800000	1388
8	99306	900694	900000	694
9	99306	1000000	1000000	0

* This is similar to what is done in a Kolmogorov-Smirnov test for a *continuous* uniform distribution.

The maximum absolute difference between the observed and expected cumulative distributions occurred at bin 2. With n = 1000000, k = 303360, and p = 0.3, the log Bayes factor is –20.057. In other words, the odds are about exp(20.057) = 514 million to 1 that this data did not come from a discrete uniformly distributed random variable.

The odds ratio test can be applied to any discrete distribution, not just a discrete uniform distribution. Just substitute, in Equation (B.13), the cumulative distribution function of the expected distribution.