Enigma 2000: An 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. Enigma 2000 (E2K) is a cipher that updates the World War II-era Enigma Machine for the twenty-first century. Like the original Enigma, E2K is intended to be computed by an offline device; this prevents side channel attacks and eavesdropping by malware. Unlike the original Enigma, E2K uses modern cryptographic algorithms; this provides secure encryption. E2K 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. E2K uses a nonce in addition to the secret key, and requires that different messages use unique nonces. E2K performs authenticated encryption, and optional header data can be included in the authentication. This paper defines the E2K encryption and decryption algorithms, analyzes E2K's security, and describes an encryption appliance based on the Raspberry Pi computer for doing E2K encryptions and decryptions offline.

1 Introduction

Modern ciphers are vastly superior to the ciphers of the pre-computer era, such as the Enigma Machine used during World War II and similar machines used in subsequent years. These days we have powerful computers at our disposal, both to perform intricate encryption algorithms and to carry out cryptanalytic attacks. We know numerous attack techniques—differential attacks, linear attacks, algebraic attacks, and the rest—and we design our cipher algorithms to resist them all. Communications encrypted with modern ciphers are surely much more secure now than in the twentieth century.

However, the very fact that modern ciphers are executed by computer hardware or software proves to be these ciphers' Achilles heel. As I pointed out previously [15], no one breaks a modern cipher in the real world by attacking the algorithm itself. Rather, intruders do side channel attacks that exploit weaknesses in the cipher's hardware or software implementation while monitoring the cipher in operation. And even if the implementation includes countermeasures against side channel attacks, an intruder can install malware, such as a keystroke logger, on the computer doing the encryption to capture the plaintexts or the secret keys without needing to observe the cipher's operation at all. The intruder can then exfiltrate this sensitive information across the Internet and, once in possession of the keys, can decrypt all subsequent messages.

If one used an Enigma Machine to encrypt messages today, side channel attacks and malware attacks would not succeed—not just because the cipher algorithm is not being executed on a

computer, but also because the encryption device is offline. With no connection from the device to the Internet or any other communication channel, an external intruder is unable to monitor the cipher's execution, eavesdrop on keystrokes, or exfiltrate encryption keys. The intruder can see only the ciphertexts. Of course, given the Enigma Machine's weak algorithm, the ciphertexts are all the intruder needs to break the cipher and recover the plaintexts—as indeed the Allies did to Enigma-encoded messages during World War II.

It would be interesting to meld a *modern* cipher algorithm with an *offline* encryption device, like the Enigma Machine. That way, we get the best of both worlds: secure encryption plus side channel and malware attack prevention. Alex* enters a key and a plaintext on the offline device, which performs the encryption and displays the ciphertext. Alex then types the ciphertext on an online computer for transmission to Blake, via email or instant message, say. Blake receives the ciphertext and enters it along with the key on the offline device, which performs the decryption and displays the plaintext.

Enigma 2000 (abbreviated E2K) is an attempt at such a cipher. E2K 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. E2K uses a nonce in addition to the secret key, and requires that different messages use unique nonces. E2K performs authenticated encryption, and optional header data can be included in the authentication.

Enigma 2000 updates the Enigma algorithm for the twenty-first century. Like the original Enigma, E2K is a polyalphabetic cipher: each plaintext letter is mapped to a ciphertext letter by means of a permuted alphabet. Unlike the original Enigma, the permuted alphabet at each plaintext position is not determined by a series of rotors. Rather, E2K derives the permuted alphabet from the SHA-256 digest of the key, the nonce, and the index of the plaintext position. Thus, unlike historical polyalphabetic ciphers which typically repeat the permuted alphabets every so many positions, E2K's plaintext positions' permuted alphabets are all different from each other (because a different index is hashed at each position). Also, while the original Enigma could not generate certain permuted alphabets due to the design of its rotors (a weakness that helped the Allies break Enigma), E2K can generate any possible permuted alphabet.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the E2K encryption and decryption algorithms. Section 4 analyzes the statistical characteristics of the E2K ciphertexts. Section 5 analyzes the security of E2K. Section 6 describes an encryption appliance, based on the Raspberry Pi computer, that calculates E2K encryptions and decryptions offline.

2 Related Work

Enigma-inspired modern ciphers. Several recent ciphers that are similar to or inspired by the Enigma Machine have been published. Like the original Enigma, these are all polyalphabetic ciphers that use permuted alphabets to map plaintext characters to ciphertext characters.

In 1993 Anderson [1] described "a modern rotor machine" combining Enigma's notion of rotors with a linear feedback shift register (LFSR). The rotor machine used an alphabet of size 256; there were three permutations initialized at random based on the encryption key; each permutation was rotated based on the value of the LFSR, thus acting as LFSR-controlled rotors; a plaintext byte was encrypted by mapping it through the rotated permutations in series; and the LFSR was clocked, thus changing the rotors for the next plaintext byte. Anderson asserted that "it ap-

* I prefer to name my cryptography users Alex and Blake, which are more gender-neutral than the usual names.

pears to resist all known attacks" (in 1993). However, in 2015 Kepley *et al.* [17] published an attack, employing a standard computer algebra system, using chosen plaintexts and chosen ciphertexts, that recovers a key equivalent to the original secret key in just minutes of computation. In contrast, E2K uses completely different permutations at each plaintext position, rather than rotations of permutations that are fixed after initialization.

Billings [3] described Enigmatique, which changes the permuted alphabet after encrypting each character by doing a series of swaps. Billings did not specify a particular algorithm for determining these swaps; rather, any of a very large set of possible swap algorithms can be used, and the swap algorithms can even be selected on the fly during an encryption. Billings did not evaluate the security of Enigmatique. In contrast, E2K uses the SHA-256 hash function to determine the permuted alphabet for each plaintext position; this lets me argue the security of E2K from the security of SHA-256 (see Section 5).

Lloyd *et al.* [19][10][21][20] proposed "quantum enigma machines" that rely for their security on the phenomenon of quantum data locking. E2K needs no quantum computer to operate.

Offline-computed ciphers. Several "low-tech" ciphers that can be computed offline 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 [22] 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. Card-Chameleon [22] and Mirdek [6] also use a deck of playing cards, but use substitution tables to convert the plaintext to the ciphertext, rather than adding a keystream. ElsieFour [15] uses a 6-by-6 matrix of tiles to encrypt and authenticate plaintexts. Chaocipher [5][25] uses two permuted English alphabets, one for plaintext letters, the other for ciphertext letters, that are shuffled as the encryption proceeds; the author described a simple device, consisting of two interlocking alphabet wheels, for computing Chaocipher encryptions and decryptions. Handycipher [14] is a homophonic monoalphabetic substitution cipher computed with pen and paper using a matrix of letters.

Like these low-tech ciphers, E2K encryptions and decryptions are intended to be performed offline. However, in contrast to these low-tech ciphers which can be executed without involving a computer, E2K is a "high-tech" cipher that requires a computer to run the SHA-256 hash function at the algorithm's heart.

3 E2K Encryption and Decryption Algorithms

Alphabet. E2K encrypts and decrypts plaintexts and ciphertexts consisting of strings of five-bit integers. For human input and output, the integers 0 through 31 are mapped to these 32 characters (case insensitive):

```
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 _ @ # & < >
```

To allow the plaintexts and keys to include digits and other punctuation within the five-bit encoding, E2K borrows an idea from the old teletypewriter codes, like the Baudot code. Along with the normal character mapping above, E2K also uses this alternate character mapping:

```
0 1 2 3 4 5 6 7 8 9 ? ! $ % + - * / ^ = . , : ; ( ) _ @ # & < >
```

When converting a plaintext or a key from a string of characters to a string of integers or vice versa, the conversion starts with the normal mapping, shifts to the alternate mapping if a `'<'` character (value 30) is encountered, and shifts to the normal mapping if a `'>'` character (value

31) is encountered. For example, the character string `"ABC_<123>_DEF"` maps to the integer string 0, 1, 2, 26, 30, 1, 2, 3, 31, 26, 3, 4, 5. However, when converting a *ciphertext* from a string of characters to a string of integers or vice versa, the conversion uses only the normal mapping.

Henceforth, I will use the term "character" to refer interchangeably to a five-bit integer or to the character corresponding to a five-bit integer. I will use the term "string" to refer to a string of five-bit integers (characters).

Key. The E2K key is a string of length 16 or greater (80 or more bits). There is no upper bound on the length of the key. The key is intended to be an easy-to-remember passphrase. However, the key must also have enough entropy to avert brute force key guessing attacks.

Diceware [24] is one way to generate such a key: pick k words at random from a list of n words, yielding a key with an entropy of $k \log_2 n$ bits. Here is an example using the Electronic Frontier Foundation's long word list [4] with $n = 7776$ and $k = 6$, for an entropy of 77.6 bits:

`SLIDESHOW_OVERPAY_CONFESS_CARDSTOCK_TWELVE_FINALLY`

Nonce. The E2K nonce is a string of length 8 (40 bits). Every message encrypted with a particular key must use a different nonce. As is true of many ciphers, encrypting more than one message with the same key and the same nonce leaks information that can be used to attack the cipher. The nonce could be an (encoded) sequence number. The nonce could be chosen at random; but in that case a nonce collision is expected to occur after about 2^{20}, or about one million, messages. After encoding that many messages, Alex and Blake should pick a different key.

Header. E2K supports including an optional header as additional authenticated data in a message. The header is a string of any length; if not needed, the header is an empty (zero-length) string.

Permuted alphabet generation. Let $S_c[x]$ be the permuted alphabet that maps plaintext character x ($0 \le x \le 31$) at index c ($c \ge 0$) to ciphertext character y ($0 \le y \le 31$). The index denotes the character's position in the plaintext. The permuted alphabet is determined as follows:

$D \leftarrow$ SHA-256 (key || 00001 || nonce || 00010 || (32-bit value $c + 1$))
For $x = 0$ to 31:
    $S_c[x] \leftarrow x$
For $x = 0$ to 30:
    $j \leftarrow D \bmod (32 - x)$
    Swap $S_c[x]$ with $S_c[x+j]$
    $D \leftarrow D / (32 - x)$

where "||" is concatenation of bit strings, "mod" is the nonnegative integer remainder operation, and "/" is the truncating integer division operation.

Here is an example. The key is `"KAMINSKYPASSWORD"`, which is encoded as this bit string:

01010 00000 01100 01000 01101 10010 01010 11000
01111 00000 10010 10010 10110 01110 10001 00011

After the key comes the bit string 00001, for domain separation. The randomly chosen nonce is `"I&VPSWYT"`, which is encoded as this bit string:

01000 11101 10101 01111 10010 10110 11000 10011

After the nonce comes the bit string 00010, again for domain separation. The index c is 0, so $c + 1$ is represented as this 32-bit string:

00000000 00000000 00000000 00000001

D, a 256-bit integer, is the SHA-256 digest of the concatenation of the above five bit strings. Note that the length of the concatenated bit strings is not necessarily a multiple of 8; thus, the SHA-256 implementation must be able to handle inputs of arbitrary bit length, not just byte sequences. In this example, *D* is (decimal)

18155582907448927521984985698441208202874806821993959926919987808752615671964

or (hexadecimal)

2823b21de570a045c71c807d8d0f48fb505b1c63f674ef45ced2c1462369c89c

Then the permuted alphabet is

x	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	_	@	#	&	<	>
$S_c[x]$	#	I	X	R	H	U	G	Z	Y	O	Q	>	S	W	K	&	J	L	C	T	N	V	@	M	P	B	_	<	E	F	D	A

If *c* is changed to 1 while the key and the nonce stay the same, the permuted alphabet becomes

x	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	_	@	#	&	<	>
$S_c[x]$	Q	X	G	B	P	>	D	E	Z	@	J	N	M	T	Y	_	#	H	F	R	C	S	W	L	V	O	K	A	I	U	<	&

There are $32! \approx 2^{117.7}$ possible permutations of a 32-character alphabet. The E2K permuted alphabet generation procedure in effect uses the 117.7 least significant bits of the SHA-256 digest of the key, nonce, and index to select one of these permutations. Assuming that SHA-256 behaves as a random oracle, the (truncated) digest selects one of the 32! possible permutations at random; and changing any or all of the key, the nonce, or the index yields a different permutation almost certainly. Thus, every plaintext character position in every E2K encryption uses a different permuted alphabet, provided each encryption with a particular key uses a different nonce.

In the original Enigma, because of the design of the rotors, each position's permuted alphabet never mapped a character to itself. Also, the permutation was always self-reciprocal: if *x* mapped to *y*, then *y* mapped to *x*. Thus, Enigma could generate only a small fraction of the possible permutations. In contrast, E2K can generate any possible permutation, including those where a character maps to itself (as can be seen in the above examples) and those that are not self-reciprocal (likewise).

Encryption. To encrypt a plaintext *X*, a string of length *L*, consisting of characters *X*[0] through *X*[*L*–1], along with a (possibly empty) header *H*, a string of length *M*, consisting of characters *H*[0] through *H*[*M*–1], yielding a ciphertext *Y*, a string of length *L*+16, consisting of characters *Y*[0] through *Y*[*L*+15]:

D ← SHA-256 (*H*[0] .. *H*[*M*–1] || 00001 || *X*[0] .. *X*[*L*–1])
X[*L*] .. *X*[*L*+15] ← the 80 most significant bits of *D* as 16 characters
For *c* = 0 to *L*+15:
    Y[*c*] ← S_c[*X*[*c*]]

The first *L* characters of the ciphertext are the encrypted plaintext. The last 16 characters of the ciphertext are the 80-bit message authentication tag. The tag is the encrypted SHA-256 digest of the header, a domain separation bit string of 00001, and the plaintext, truncated to 80 bits. Alex chooses a unique nonce, encrypts the plaintext and the digest with the secret key and the nonce, and sends the nonce and the ciphertext to Blake. Alex assumes Blake knows what the header is supposed to be, so Alex need not encrypt or send the header.

Decryption. To decrypt a ciphertext *Y*, a string of length *L*+16, consisting of characters *Y*[0] through *Y*[*L*+15], along with a (possibly empty) header *H*, a string of length *M*, consisting of characters *H*[0] through *H*[*M*–1], yielding a plaintext *X*, a string of length *L*, consisting of characters *X*[0] through *X*[*L*–1]:

For c = 0 to L+15:
    $X[c] \leftarrow S_c^{-1}[Y[c]]$
$D \leftarrow$ SHA-256 (H[0] .. $H[M–1]$ || 00001 || X[0] .. $X[L–1]$)
If $X[L]$ .. $X[L+15]$ = the 80 most significant bits of D as 16 characters:
    Authentication succeeds
Else:
    Authentication fails

$S_c^{-1}[y]$ is the inverse of the permuted alphabet at index c. Blake receives the nonce and the ciphertext from Alex, decrypts the ciphertext with the secret key and the nonce, and accepts the plaintext if the authentication succeeds or rejects the plaintext if the authentication fails.

Replays. The E2K encryption and decryption algorithms do not themselves detect a replay of a ciphertext. Alex and Blake can detect a replay by including unique identifying information in the plaintext, such as a serial number or a date/time stamp.

Key establishment. Before sending encrypted messages to each other, Alex and Blake must establish the secret key they will use. Because E2K is intended for human-to-human communication, a key exchange protocol is unnecessary. Alex and Blake can meet in person in a secure location and agree on a secret key. Rather than meeting in person, Alex and Blake could use a key exchange protocol based on a public key infrastructure (PKI) over the Internet; but then the computers executing the key exchange protocol would be susceptible to side channel attacks, malware attacks, or attacks against the PKI that could recover the secret key.

Design rationale. I intended E2K for human-to-human secure messaging. I also intended E2K to be computed by a small offline device with a limited screen area, like the Raspberry Pi computer described in Section 6. This motivated several choices in the design of E2K.

The human-to-human messaging means that E2K needs to encrypt only English letters, digits, and punctuation, not arbitrary binary data. The limited screen area means the alphabet size has to be limited, so that the buttons for the alphabet's characters can all fit on the screen. I therefore decided to use a 32-character alphabet and to define the algorithms as operating on five-bit integers. The normal and alternate character mappings permit the plaintexts and keys to use an expanded set of 58 characters. (Although E2K could be used to encrypt arbitrary binary data by dividing the data into a series of 5-bit characters, this was not my primary intent.)

I decided to base E2K on the existing SHA-256 hash function, both for encryption and for authentication, rather than trying to define a new core cryptographic algorithm. SHA-256's security is well understood, and this translates to the security of E2K as I will argue in Section 5.

Much recent cryptographic research has been focused on fast, lightweight ciphers, to achieve high-throughput encryption while minimizing gate area in hardware or CPU cycles in software. The tradeoff is that the faster and lighter a cipher becomes, the less secure it becomes. In contrast, E2K is a slow, heavyweight cipher: encrypting or decrypting each character involves a full SHA-256 hash computation plus generation of a permuted alphabet. However, E2K only needs to encrypt or decrypt as fast as a human can type characters on the offline encryption device.

Given that E2K need not have high throughput, there's no reason to avoid basing E2K on a secure but heavy primitive like SHA-256.

4 Statistical Analysis of E2K

The security of E2K relies on the properties of SHA-256. While I believe that SHA-256 behaves as a random oracle, and therefore that the permuted alphabets for different keys, nonces, and plaintext positions are chosen uniformly at random, and therefore that the ciphertexts look like uniformly chosen random strings, it is as well to verify this with statistical tests of randomness. The statistical tests are *odds ratio uniformity tests.* (See Appendix B for a description of the odds ratio uniformity test.) Each test yields a *log odds ratio* for the hypothesis that the items being tested are uniformly distributed. If the log odds ratio is greater than 0, the test *passes* and detects random behavior. If the log odds ratio is less than 0, the test *fails* and detects nonrandom behavior.

Permuted alphabet randomness. I wrote a program* to test whether the permuted alphabets for different keys, nonces, and plaintext positions are chosen uniformly at random. Given a key, a nonce, and a number of plaintext positions N, the program examines every combination of a one-character change to the key, a one-character change to the nonce, and indexes c from 0 to N–1. For each such combination, the program generates and examines the permuted alphabet $S_c[x]$. For each plaintext character x from 0 to 31, the program counts the number of times the permuted alphabets map x to A, to B, to C, and so on. The hypothesis is that the permuted alphabets map each plaintext character to uniformly chosen random ciphertext characters. The program reports the log odds ratio for this hypothesis for each plaintext character, as well as the aggregate log odds ratio over all plaintext characters.

Table 1 lists the results of four program runs as well as the key, nonce, and N value used for each run. I deliberately chose the keys and nonces to be nonrandom, to ensure that any random behavior detected in the permuted alphabets comes from the permuted alphabet generation procedure itself, even when that procedure is presented with nonrandom inputs. Despite a few isolated occurrences of negative log odds ratios, the aggregate log odds ratios are all positive, confirming that overall, E2K's permuted alphabets look as though chosen at random.

Ciphertext character position randomness. I wrote a program to test whether the ciphertext characters at each position look as though chosen at random for various choices of keys and nonces, even when the keys, nonces, and plaintexts are nonrandom. The program examines K keys; the keys are sequential values starting at all zeroes. For each key, the program examines N nonces; the nonces are sequential values starting at all zeroes. For each key and nonce, the program encrypts an all-zeroes plaintext, yielding a ciphertext and a tag. For each key, across all the nonces, the program tests whether the ciphertext and tag characters and digrams at each position obey a uniform distribution, using an odds ratio uniformity test. The program prints the tests' log Bayes factors, aggregated across the keys, for each ciphertext and tag character position and each ciphertext and tag digram position.

I ran the program with K = 100 keys of length 16, N = 10000 nonces, and plaintexts of length 40. Table 2 lists the aggregate log Bayes factors. All are positive, confirming that the individual ciphertext and tag characters and digrams look as though chosen at random.

* The programs mentioned in this paper, as well as an implementation of E2K itself, written in Java, are available at https://www.cs.rit.edu/~ark/parallelcrypto/enigma2000/.

Table 1. Permuted alphabet odds ratio uniformity test results

Plaintext character	*Run 1 l.b.f.*	*Run 2 l.b.f.*	*Run 3 l.b.f.*	*Run 4 l.b.f.*
A	8.8486	8.3473	5.8497	7.2179
B	8.4988	5.6485	9.7812	8.3043
C	2.9865	8.3238	3.1714	8.1156
D	8.5920	6.1802	9.1810	6.4515
E	–1.7835	1.9124	6.4786	8.6657
F	9.6818	8.8081	8.9685	9.1618
G	6.0247	7.0002	6.0189	8.5544
H	8.2816	4.4427	9.4570	8.8821
I	5.0501	7.7733	5.6587	6.3724
J	8.2596	5.6430	7.1279	10.025
K	9.0388	6.3160	4.0541	3.6609
L	8.5361	8.7208	8.9874	8.3825
M	7.4261	8.9490	8.1917	6.9338
N	7.8034	8.4181	6.1423	9.3365
O	6.2377	8.2693	6.5946	9.4561
P	8.8760	7.2300	8.6733	8.4728
Q	8.4980	7.5606	9.5318	8.8780
R	9.3530	5.9571	6.6512	5.7301
S	8.1311	8.4693	8.2971	9.0716
T	1.9190	8.3200	5.2694	8.9979
U	7.5828	2.1175	9.4036	8.0408
V	8.9070	6.6093	9.3800	9.5697
W	8.3718	6.1519	2.0915	7.5539
X	8.5052	–0.97331	9.1102	4.8211
Y	3.9664	8.5324	4.4547	6.9697
Z	7.9588	8.6013	8.9566	3.1934
_	8.8069	7.8961	9.4029	5.3771
@	7.4030	7.6939	5.5668	8.9370
#	7.1479	9.2448	9.3975	9.5960
&	5.7482	4.3946	9.3548	9.3104
<	6.1203	8.7554	6.0575	8.6622
>	6.3913	8.4304	7.5712	8.1226
Aggregate	227.17	219.74	234.83	250.82

Run 1: Key = "AAAAAAAAAAAAAAAA", nonce = "AAAAAAAA", N = 10000
Run 2: Key = "ABCDEFGHIJKLMNOP", nonce = "ABCDEFGH", N = 10000
Run 3: Key = "ZYXWVUTSRQPONMLK", nonce = "ZYXWVUTS", N = 10000
Run 4: Key = "ZZZZZZZZZZZZZZZZ", nonce = "ZZZZZZZZ", N = 10000

Table 2. Ciphertext and tag position odds ratio uniformity test results

	Ciphertexts			*Tags*	
Posn.	*Character l.b.f.*	*Digram l.b.f.*	*Posn.*	*Character l.b.f.*	*Digram l.b.f.*
0	282.31	245.88	0	287.87	255.89
1	283.85	246.60	1	286.25	250.92
2	276.00	240.42	2	283.93	244.14
3	278.96	241.99	3	300.86	268.05
4	290.48	258.89	4	254.48	212.66
5	274.06	235.31	5	258.53	223.76
6	283.11	246.17	6	287.50	250.38
7	260.34	216.67	7	267.46	227.65
8	293.24	253.53	8	301.19	263.71
9	270.04	237.40	9	309.00	273.92
10	282.55	254.82	10	285.28	251.45
11	282.20	247.69	11	284.53	246.93
12	293.57	254.65	12	274.08	243.41
13	287.55	248.99	13	290.66	256.96
14	280.91	243.17	14	295.55	259.51
15	300.29	259.79	15	289.29	
16	278.60	244.72			
17	294.30	260.91			
18	295.81	257.95			
19	284.32	251.33			
20	299.26	262.21			
21	292.15	256.51			
22	288.95	259.77			
23	276.69	241.10			
24	285.33	249.64			
25	281.67	249.76			
26	276.45	241.73			
27	285.13	249.06			
28	300.84	268.18			
29	288.33	255.93			
30	273.31	234.29			
31	260.67	224.63			
32	272.78	234.98			
33	282.88	251.89			
34	253.09	216.44			
35	283.42	242.53			
36	293.31	252.69			
37	268.55	229.83			
38	287.13	245.65			
39	288.69				

Ciphertext overall 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 E2K'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 *K* repetitions. For each repetition, the program chose a random key and did *N* trials. For each trial, the program chose the next sequential nonce starting from all 0s*;* chose a random plaintext; and encrypted the plaintext using the key and the nonce. 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). This simulates what a pair of humans would do to communicate using E2K: 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 tag and did an odds ratio uniformity test. The program did the same for the ciphertext and tag digrams. Finally, the program aggregated the log odds ratios for the ciphertext and tag characters and digrams across the trials for each repetition, and printed all these aggregate log odds ratios.

I ran the program with $K = 100$ keys of length 16, $N = 10000$ nonces, and plaintexts of length 40. The ciphertext character and tag character aggregate log odds ratios were 479.74 and 409.47, respectively. The ciphertext digram and tag digram aggregate log odds ratios were 444.31 and 367.40, respectively. This confirms that the ciphertext and tag characters and digrams look as though chosen at random—despite the nonuniform distribution of the plaintext letters and digrams.

5 Security Analysis of E2K

In this section I argue that E2K is immune to key recovery attacks by analyzing several attack avenues: ciphertext-only attacks, known plaintext attacks, SHA-256 preimage attacks, and SHA-256 algebraic attacks. I also argue that E2K is immune to message forgery attacks.

Ciphertext-only attacks. 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. Corey the cryptanalyst, possessing only the ciphertext of a message, can take advantage of these nonuniform distributions to deduce the plaintext. As shown in Section 4, however, E2K yields ciphertexts with a uniformly random distribution of character and digram frequencies, so this kind of attack will not work on E2K.

Consequently, to carry out a ciphertext-only attack, Corey has to do a generic brute force key search: Decrypt the ciphertext with every possible key until it yields a plaintext for which the authentication succeeds. A brute force search of 2^{80} keys would be expected to find one key on average that yields a plaintext whose 80-bit truncated digest equals the message's decrypted tag. However, the brute force search would require an impractical amount of work. (This assumes the key's entropy is at least 80 bits, as mentioned previously.)

One avenue of ciphertext-only attack on the original Enigma Machine relied on the fact that Enigma never encrypted a letter to itself. Corey would postulate that an Enigma ciphertext was the encryption of a plaintext containing a certain string (a "crib", as this string was called). Corey aligned the first letter of the crib with the first letter of the ciphertext and checked whether any letter in the crib equaled any letter in the ciphertext; if so, the plaintext did not contain the crib at that position. Corey then shifted the crib one place forward, compared it with the ciphertext, and repeated. Any position where none of the crib letters matched the corresponding ciphertext letters indicated that the plaintext potentially contained the crib at that position. Corey could then use this assumed partial plaintext in a search for the encryption key.

E2K overcomes this weakness of Enigma. Because E2K's permuted alphabets can map a plaintext character to itself, any possible crib can appear at any position in any ciphertext, and Corey has no way to narrow down the existence or position of a crib in the plaintext.

Known plaintext attacks. If Corey knows both the plaintext and the ciphertext of an E2K encrypted message, Corey can do a brute force key search with less computation than a ciphertext-only attack. Testing a potential key, Corey encrypts the known plaintext characters one at a time. As soon as the resulting ciphertext character does not match the corresponding character in the known ciphertext, Corey knows the key is incorrect and can go on to try the next key. With a ciphertext-only attack, Corey must decrypt and authenticate the full ciphertext, including the tag, to test a potential key. However, the known plaintext brute force key search still requires an impractical 2^{80} amount of work.

SHA-256 preimage attacks. Given enough information from known plaintexts and ciphertexts encrypted with a particular key, Corey can deduce the permuted alphabets used to encrypt each plaintext character, and therefore can deduce the SHA-256 truncated output digest values that generated the permuted alphabets. Corey also knows a portion of the SHA-256 input bit strings, namely the nonce and the index. Corey wants to deduce the remainder of the SHA-256 input bit strings, namely the key. This is a preimage attack on SHA-256.

A number of structural preimage attacks on SHA-256, which require less work than a 2^{256} generic preimage attack, have been published. Isobe and Shibutani [13] described a preimage attack on SHA-256 reduced to 24 rounds (of 64 rounds) that requires 2^{240} work. Aoki *et al.* [2] described a preimage attack on 43 rounds with $2^{254.9}$ work. Guo *et al.* [11] described a preimage attack on 42 rounds with $2^{248.4}$ work. Khovratovich *et al.* [18] described a preimage attack on 45 rounds with $2^{255.5}$ work. None of these attacks succeed against full-round SHA-256, and all of these attacks are theoretical only, requiring an impractical amount of work.

Furthermore, while the preceding generic and structural attacks find *a* preimage that produces a given digest, they do not necessarily find *the specific* preimage (key, nonce, index) E2K used to encrypt a certain message. Because the key length and therefore the hash input length is effectively unlimited, there could be many different hash inputs that produce the same truncated 117.7-bit digest output (by the pigeonhole principle). Finding one such input for a message with a given nonce does not necessarily yield the correct key; and attempting to decrypt ciphertexts with that incorrect key and other nonces almost certainly will not succeed.

SHA-256 algebraic attacks. Algebraic attacks represent a cipher as a collection of Boolean expressions mapping input variables, such as E2K's key, nonce, and index bits, to output variables, such as E2K's permuted alphabets. Corey observes the output bit values and tries to deduce the input bit values by solving the Boolean equations.

Dinur's and Shamir's cube attack [7] has achieved success at recovering the key from reduced-round versions of various stream ciphers. The cube attack requires 2^d work, where d is the algebraic degree of the cipher's Boolean expressions. For the cube attack to be practical, d cannot be too large. One round of SHA-256 involves seven 32-bit additions as well as other operations; each addition has an algebraic degree of 32 due to the carry propagation; the algebraic degree therefore increases by a minimum of 32 as each round follows another; and there are 64 rounds. While I have not determined, nor am I aware that anyone has published, the actual algebraic degree of SHA-256, it seems clear that d would be too large for a cube attack on the full-round SHA-256 to succeed.

Dinur and Shamir also published a modified cube attack [8] that uses side channel information to reduce the amount of work. This attack is not applicable to E2K in the real world, however, as Corey has no way to obtain side channel information from the offline encryption device.

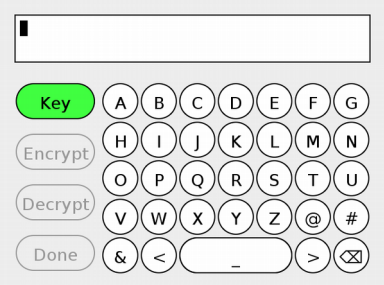
Hao *et al.* [12] and Nejati *et al.* [23] published algebraic fault attacks on SHA-256. "Fault attacks" like these work by altering particular bits or words of the SHA-256 state at particular steps during the hash computation. The attacks are "algebraic" because they represent SHA-256 as algebraic normal form (ANF) expressions, then apply a Boolean satisfiability (SAT) solver to the ANF expressions to determine the faults to inject. Such attacks are not applicable to E2K in the real world, however, as Corey has no way to inject the required very specific faults into the offline encryption device and then exfiltrate the faulty hash computation's results.

Message forgery. For authentication, E2K uses a technique similar to the CCM mode of operation of a block cipher [9]: hash the plaintext; append the hash digest to the plaintext; and encrypt the whole thing using counter mode. The encrypted digest becomes the message authentication tag. A successful forgery of a new message or an altered message is therefore impossible, as it would require knowledge of the secret encryption key.

6 An E2K Encryption Appliance

Figure 1 depicts what I like to call an "encryption appliance" for performing E2K encryptions and decryptions offline. The E2K encryption appliance is based on the credit-card-sized Raspberry Pi computer, coupled with a touch-sensitive 320-by-240-pixel color display. The Pi runs the Linux operating system, and the E2K application is written in Java.* Note that the only wire entering the Pi is the power cable. The Pi does not have a wireless Ethernet interface. While the Pi does have a wired Ethernet interface, this is not connected to anything.

Here are screenshots of the E2K application in operation as Alex encrypts a message for Blake and Blake decrypts the message:

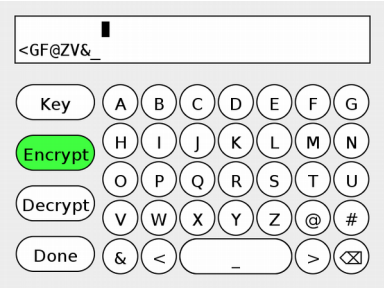


When Alex starts the E2K application, this screen appears, inviting Alex to enter the secret key.

* The E2K application's Java source code is available at https://www.cs.rit.edu/~ark/parallelcrypto/enigma2000/.

Figure 1. E2K encryption appliance

Alex types the secret key on the keyboard on the touch screen. (The key at the lower right is a Delete key.) Once Alex has entered at least 16 characters, the “Encrypt” and “Decrypt” buttons become active.



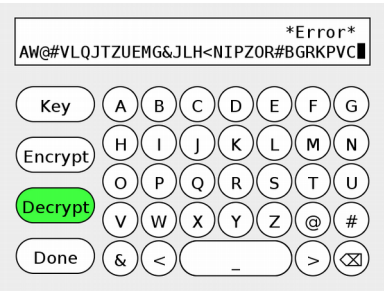
Alex taps the “Encrypt” button. The display changes to show the plaintext (initially empty) on the first line and the ciphertext on the second line. The ciphertext begins with the randomly chosen nonce, obtained from the system entropy source.

Alex enters the plaintext. As Alex types each plaintext character, the corresponding ciphertext character appears. If Alex types the alternate-map character (`'<'`), the keyboard changes to show the alternate character set. If Alex types the normal-map character (`'>'`), the keyboard changes back to the normal character set.

Once the plaintext is complete, Alex taps the “Done” button. The appliance computes the message authentication tag, which appears at the end of the ciphertext. (The E2K application does not support additional authenticated data; the header is always empty.) Alex can scroll the display by swiping it left or right. Alex types the ciphertext message into an online computer for transmission to Blake.

Upon receiving the message, Blake starts up the encryption appliance; enters the secret key previously agreed with Alex; taps the “Decrypt” button; and enters the ciphertext, beginning with the nonce. As Blake types each ciphertext character, the corresponding plaintext character appears.

After typing the entire ciphertext including the tag, Blake taps the “Done” button. The appliance authenticates the message. If the authentication succeeds, the display shows “OK” along with the plaintext.



If there is an error in the message, either due to malicious activity or due to Blake’s mistyping, the authentication fails. The plaintext disappears and the display shows “Error”.

To encrypt another message with the same key, Alex or Blake taps the “Encrypt” key; the appliance chooses a new random nonce for the new message. To decrypt another message with the same key, Alex or Blake taps the “Decrypt” button. To start over with a different key, Alex or Blake taps the “Key” button.

References

[1] R. Anderson. A modern rotor machine. *Fast Software Encryption (FSE 1993)*, pages 47–50, 1993.

[2] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang. Preimages for step-reduced SHA-2. *Advances in Cryptology—ASIACRYPT 2009,* pages 578–597, 2009.

[3] C. Billings. The Enigmatique toolkit. Cryptology ePrint Archive, Report 2008/408, September 24, 2008.

[4] J. Bonneau. Deep Dive: EFF's New Wordlists for Random Passphrases. https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases, retrieved April 17, 2019.

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

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

[7] I. Dinur and A. Shamir. Cube attacks on tweakable black box polynomials. Cryptology ePrint Archive, Report 2008/385, January 26, 2009.

[8] I. Dinur and A. Shamir. Side channel cube attacks on block ciphers. Cryptology ePrint Archive, Report 2009/127, March 18, 2009.

[9] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality.* NIST Special Publication 800-38C, May 2004.

[10] S. Guha, P. Hayden, H. Krovi, S. Lloyd, C. Lupo, J. Shapiro, M. Takeoka, and M. Wilde. Quantum enigma machines and the locking capacity of a quantum channel. arXiv:1307.5368 [quant-ph], November 9, 2013.

[11] J. Guo, S. Ling, C. Rechberger, and H. Wang. Advanced meet-in-the-middle preimage attacks: first results on full Tiger, and improved results on MD4 and SHA-2. Cryptology ePrint Archive, Report 2010/016, September 3, 2010.

[12] R. Hao, B. Li, B. Ma, and L. Song. Algebraic fault attack on the SHA-256 compression function. *International Journal of Research in Computer Science* 4(2):1–9, 2014.

[13] T. Isobe and K. Shibutani. Preimage attacks on reduced Tiger and SHA-2. *Fast Software Encryption (FSE 2009),* pages 139–155, 2009.

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

[15] A. Kaminsky. ElsieFour: a low-tech authenticated encryption algorithm for human-to-human communication. Cryptology ePrint Archive, Report 2017/339, April 12, 2017.

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

[17] S. Kepley, D. Russo, and R. Steinwandt. Cryptanalysis of a modern rotor machine in a multicast setting. Cryptology ePrint Archive, Report 2015/661, July 1, 2015.

[18] D. Khovratovich, C. Rechberger, and A. Savelieva. Bicliques for preimages: attacks on Skein-512 and the SHA-2 family. Cryptology ePrint Archive, Report 2011/286, February 7, 2012.

[19] S. Lloyd. Quantum enigma machines. arXiv:1307.0380 [quant-ph], July 1, 2013.

[20] D. Lum, M. Allman, T. Gerrits, C. Lupo, V. Verma, S. Lloyd, S. Nam, and J. Howell. A quantum enigma machine: experimentally demonstrating quantum data locking. arXiv:1605.06556 [quant-ph], July 21, 2016.

[21] C. Lupo and S. Lloyd. Continuous-variable quantum enigma machines for long-distance key distribution. arXiv:1501.07212 [quant-ph], October 12, 2015.

[22] 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.

[23] S. Nejati, J. Horáček, C. Gebotys, and V. Ganesh. Algebraic fault attack on SHA hash functions using programmatic SAT solvers. *Principles and Practice of Constraint Programming (CP 2018)*, pages 737–754, 2018.

[24] A. Reinhold. The Diceware Passphrase Home Page. http://world.std.com/~reinhold/diceware.html, retrieved April 17, 2019.

[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.

A E2K Encryption Example

Here is a trace of E2K encrypting the message “It’s 12 AM and I’m about to put the hammer down.” The trace shows the plaintext character position *c*, the permuted alphabet $S_c[x]$, the plaintext character *x*, and the ciphertext character *y* at each step.

```
Key:       RUBBER_DUCK_<142857>
Nonce:     LFTXKIDZ
Header:    V<1.0>
Plaintext: ITS_<12>_AM_AND_IM_ABOUT_TO_PUT_THE_HAMMER_DOWN
Digest:    >FNN@F>UHCDRUB>B

0   HCA><BRGDWXVOT&PMUJKYZNE_I@FQ#SL  I  D
1   HYVGNM>KCDQ&IESFA#<W@XPUTRBOJL_Z  T  W
2   TLZW@_JDEKRGNFCBOX<QVYIS&HAU#MP>  S  <
3   GNIOU_WFTZL#Y<PMEQ@&XHK>JACBVSRD  _  C
4   _MEFUP><XH#&KGOQSTACWNLYDR@ZVIBJ  <  B
5   RQGP>F&TKWNU@CEMZJ#HSDYL<VBIAXO_  1  Q
6   HEYDIZJV>KTSF_@AWUGN#<RBQMXLP&CO  2  Y
7   WDQRZK#HX<>UNISL@VJOCMPGB&EAF_YT  >  T
8   ZG@<TN_YISHKBO#WXRUP&>QVAFCEMDLJ  _  C
9   JEW_HB<FCKD&QLTUVYAI>NZPGR@MSXO#  A  J
10  @EBCDXHW#UQOTR&K_PVLSNMA<>JFYZIG  M  T
11  Z&HW<R#P_AE@KVFS>JMDTUXGOCNQYILB  _  N
12  HVBE>@NAFCK_WRGDTQXPUMYZIL<&#SOJ  A  H
13  NGD#WEAXBICJZRQV<MS>_HK&UOTLP@FY  N  R
14  JD#AWSHXELK&C@_MRUYOFP<TB>NVZQGI  D  A
15  _OXKAUBTJ<ZF>YQW&@SMCERIDGVHLNP#  _  V
16  BOW<TFENLHD@VMYS>RPZJU_AXICQG#K&  I  L
```

```
17  AL>HGNOWQV&DJ@MX#<RYIUTF_BCPZKES  M  J
18  >ZGK#_CQIJOUPNBMYSRVEFWDHXA&<@TL  _  A
19  QXS&W_<RZC>FDNMOV@TEPKHUG#BYAILJ  A  Q
20  HZVO@TKI>PL_BF&RU#AC<MYSDJWQENGX  B  Z
21  RXAKUVC>ZWGOJT#FQSHE@DPI<N_LYB&M  O  #
22  <BDLYXVG@MRHA&UCFQ#TEKOSI>PN_WZJ  U  E
23  T<XDIYBR&WSJ@HNMCAGZ_L#>PFEOUQVK  T  Z
24  #WS<OTFCR@IVPH_GAXB>QKYZLU&JDNEM  _  &
25  ERGNJPVI_XLF#ASHCBY&QMO<D>UW@TKZ  T  &
26  TCOB#M_GL@KZQASHJN<W>IRVPD&YFUEX  O  S
27  NEXKHJUIWYFD>#SPA&Z<L_GOCVRMQ@BT  _  R
28  XHYFEWCNOJ#KRIMLDUGP>SVQ@<ATB_&Z  P  L
29  IK<OHJGVE&UM@TNAXCBW#LD_>FQYPSZR  U  #
30  NAUYLPV&QOB@HDXCKJRI>#TMZS<F_WEG  T  I
31  YQRXCS_DLPO<JWIVNBAGKM&>#ZT@UEFH  _  T
32  &PFOWE<UXNM>VL_C#DYIKGTSR@HZJBAQ  T  I
33  UGEI<LSNZ_>AJKVRCPXFHYMT@W#DO&BQ  H  N
34  EHGQPNC@SADYFOKXJWT>Z_MVIL<R#U&B  E  P
35  &NQKESHTROZAU@VJDXYL><GIMBW#P_FC  _  W
36  VWTGQ#AJR>ZBSPFKUM&NY<_HILDX@OEC  H  J
37  LRQEMYUWJDKTZXPV<_S#GAIHFB&CNO@>  A  L
38  TJ<NI@ABYM&CVSRHPDEXFLKW#OZQ>_GU  M  V
39  Z<TPCOXQ@VS#NHUYKEGWD_IJ>B&MRLAF  M  N
40  GQDSKNTX#EAHML&B>WRVJYPZ<@_OCFUI  E  K
41  WPEVNUQC>_XF@#<AZ&JIHSTRDGLKYBOM  R  &
42  LFTVP&<SXUGEHY#JIZMDWA@C>RN_QKBO  _  N
43  LFV@RGDSHW#IUNOAK>Q&ZT_<YBCEJMPX  D  @
44  @&K_JFVGAHBI>CDEZLW<XO#UYNSPRTMQ  O  D
45  ZC#JQXSHEDVK>MAP@WFLBOUTRY_GN&<I  W  U
46  &QNHUZBE#@KJ>FDWTILC<YMPAGX_ORSV  N  F
47  &G<XLUM@PF_BI>ECA#YZQNOKWJSDTHRV  >  V
48  H@&ZMUCGJD>ERVYKOF_SINWBAXTPQ#L<  F  U
49  IXTHOLUN<CW&SFQJERVB@>AYPG_MK#ZD  N  F
50  AYVLGUKTE_#&IJ<DHXNF@QRCSP>MOBWZ  N  J
51  B>J&NY#G@FHWEVOIRMSAC<TKZQUPL_XD  @  P
52  UFHI_LDMXY#@Q&KGWTBCEARPS>JON<ZV  F  L
53  &LIXTFSCAGJQPBRW_DOEYK<V#U@MH>ZN  >  N
54  TQC#&>RDNP_YIJL@HSMZGOUAKEBF<WVX  U  G
55  AFRJP&EG#YDBQI@KN>TLW<UVHCOX_ZSM  H  G
56  FD@BZHTENW#AIPJMV>OUXSR<CLQ&GK_Y  C  @
57  &DZNHBOCVMGQJXLTEKPFIU>WSA@YR_#<  D  N
58  LSDI&P>GQX#ECOZBKU_FVJY@HMNRTWA<  R  U
59  UCSVMDI_YPB#KRNEFQW>TJHA<XZO@LG&  U  T
60  JGE@ACBSVQ&DKXPRTL>YFN<Z_IUWH#OM  B  G
61  EDRWBP>OGAYMSQKZJXTN@#CLV<FH&_UI  >  I
62  U<@RHDS_L>IN&TWCOPZ#MEFQYGXVKABJ  B  <

Nonce:      LFTXKIDZ
Ciphertext: DW<CBQYTCJTNHRAVLJAQZ#EZ&&SRL#ITINPWJLVNK&N@DUF
Tag:        VUFJPLNGG@NUTGI<
```

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 [16].

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

* To make this paper self-contained, this appendix is a copy of Appendix B of [15].

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 [16]. 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 [16]:

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

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

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

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

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

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

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:

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

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

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

Let y be the bin such that the absolute difference $|F_{\mathrm{obs}}(y) - F_{\mathrm{exp}}(y)|$ 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 $k = F_{\mathrm{obs}}(y)$, and the success probability is $p = F_{\mathrm{exp}}(y)/n = (y+1)/B$ 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

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

(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]	$F_{obs}(x)$	$F_{exp}(x)$	$\lvert F_{obs}(x) - F_{exp}(x) \rvert$
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]	$F_{obs}(x)$	$F_{exp}(x)$	$\lvert F_{obs}(x) - F_{exp}(x) \rvert$
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

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.