Statistical Tests for Randomness on a Typewritten Key Stream Extracted With Computer Vision and Classified With a Convolutional Neural Network

Floe Foxon
University of Texas at Austin / USA
University of Leeds / UK
ff5384@eid.utexas.edu

Abstract

For a key stream to be cryptographically secure, it must be sufficiently random (i.e., unpredictable). This study tested the randomness of a set of typewritten, WW2-era German diplomatic key stream tables. Character objects were extracted from images of the tables using computer vision, and a bespoke convolutional neural network (convnet) was trained to classify these objects as digits (from 0–9). The convnet had a mean cross-validated testing balanced accuracy of 93.7% (standard deviation: 0.7%). $N = 74,979$ digits were extracted and classified from the images. Randomness was tested with the arithmetic mean, chi-squared, runs, and Monte Carlo pi tests; the key stream failed all four tests with 95% confidence. One digit appeared to be over-represented, and two others under-represented in the tables. Analysis suggests that the under-represented digits may be a simple artefact of computer vision error/bias, but the over-represented digit did not appear to have resulted from computer vision and/or classification error/bias. Reference streams generated with the Mersenne Twister and Linux OS entropy passed all four tests. WW2-era German diplomatic key stream tables may have lacked randomness. The extent to which this could potentially be exploited by cryptanalysts is unknown.

1 Introduction

Random numbers are used extensively in cryptography, for example in one-time pads (OTPs). Under the usual OTP assumptions including 'true' key randomness and no key re-use, text encrypted with the OTP cannot, in theory, be decrypted by any means except by using the very same key stream that encrypted it. Modern computer systems can, in theory, provide strong (or at least, sufficient) randomness quickly and efficiently. For example, the Linux operating system generates pseudo-random numbers by collecting data in the form of noise from the computer's environment/hardware ('entropy'), such as time between keystrokes (Nakov, 2019), and transforming these data into seeds for cryptographic random number generators. Historically however, the process of generating random numbers (without computers) was more time-consuming, less efficient, and potentially less cryptographically secure. As a purely demonstrative example, one could continuously re-roll a 10-sided die labelled 0–9, but even with many dice this process would be extremely time-consuming, and the dice could be manufactured or rolled in such a way that is biased, and the resulting key stream may contain patterns that could be exploited by cryptanalysts (at least in theory).

Nevertheless, key streams *were* generated manually and used extensively in the 20th century. From a cryptographic perspective, it is interesting to test the extent to which these key streams were random. If tables of historical key streams are found to lack randomness, then this could imply a degree of insecurity in texts encrypted with the tables.

Because historical key stream tables were typewritten on paper, the first task in analysing these tables for randomness is to read the key streams into a computer. Since the tables consist of thousands of digits, digitizing them by hand would be extremely slow and prone to human error, but tools exist to automate the process, namely computer vision (to detect and segment characters on the page) and classifiers (to determine which digit the character represents). Once digitized, the second task is to test the key stream for randomness, for which many methods have been developed (Foley, 2001;

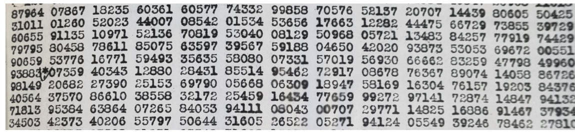
Kenny, 2005).

The aim of this study was to digitise historical key stream tables and test them for randomness, investigating bias both in the physical tables and in their digitization.

2 Methods

2.1 Data Source

The historical key stream tables used in this study come from German diplomatic tables from around the period of the Second World War, which were preserved in the United Kingdom's National Archives. As shown in Figure 1, the particular set of tables used in this study consist of thousands of digits prepared in groups of five, with 13 columns per page, dozens of rows per page, and multiple pages. The method used to generate these particular tables is unknown to the author, as is the purpose of the tables and their possible relation to the broken German OTP system described by Filby (1995).



87964	07867	18235	60361	60577	74332	99858	70576	52137	20707	14439	80605	50425
31011	01260	52023	44007	08542	01534	53656	17663	12282	44475	66729	73855	39729
60655	91135	10971	52136	70819	53040	08129	50968	05721	13483	84257	77919	74429
79795	80458	78611	85075	63597	39567	59188	04650	42020	93873	53053	69672	00551
90659	53776	16771	59493	35635	58080	07331	57019	56930	66662	83259	47798	49960
93883	07359	40343	12880	28431	85514	95462	72917	08678	76367	89074	14058	86726
98149	20682	27390	25153	69790	05668	06309	18947	58169	16304	76157	19203	84376
40564	37570	86610	38558	32172	25459	16434	77659	99272	97141	72874	14847	94132
71813	95384	63864	07265	84033	94111	08043	00707	29771	14825	16886	91467	37934
34503	42373	40206	55797	50644	31605	26522	05271	94124	05549	39246	78462	27810

Figure 1: Example image of the historical key stream tables used in this work. Image courtesy of Sir Dermot Turing and the National Archives.

2.2 Pre-Processing

Colour images of the tables were taken with a smartphone and saved in PNG format. The images were cropped as best as possible to contain only digits. These cropped image files were read into a Jupyter Notebook using the Open Source Computer Vision (cv2) Python package (2025). The imported images were converted to black-and-white, with adaptive Gaussian thresholding used to remove noise that might impact subsequent steps.

2.3 Computer Vision for Detection and Segmentation

Characters (digits) were identified by their coordinates in each pre-processed image using the Python-tesseract package; a python wrapper for Google's Tesseract optical character recognition engine (2024).

2.4 Digit Classification

Although Tesseract itself provides a classification (i.e., digit label) for each identified character, it was quickly realised that this pre-trained classifier performed poorly on the tables used in this study, e.g. by confusing the digits '3' and '5' (see Section 3). Therefore, it was necessary to build a bespoke classifier trained on these specific tables (as opposed to other texts).

For this purpose, a convolutional neural network (convnet) was built in Python using the TensorFlow (2025) and Keras (2015) packages. This convnet used a simple, feed-forward (sequential) architecture with three hidden convolution layers (each with a convolution window size of 3; and with 32, 64, and 128 respective filters; and each followed by pooling layers), as well as a dropout layer to prevent over-fitting. Batch sizes from 16 to 128, and epoch numbers from 50 to 300 were tested. The model was evaluated with three-fold cross-validation to estimate the generalisation performance. Balanced accuracy (as opposed to standard accuracy) was used as the evaluation metric to account for class imbalance (some digits appeared more frequently than others).

To train and evaluate the convnet, a dataset of $N = 651$ manually-labelled characters was created. This dataset included noise 'characters' (i.e., blobs detected by Tesseract representing ink/paper artefacts rather than real digits) which were treated as a separate, 11th class/category. After evaluation, the convnet was fitted to this entire dataset, and the final trained/fitted convnet was applied to all available images of the tables to create a final key stream dataset to be tested for randomness.

2.5 Testing for Randomness

Four tests for randomness were used, testing different aspects of the key stream:

1. **Arithmetic Mean ($\bar{x}$) Test:** In the limiting case, the frequencies of truly randomly-generated digits from 0–9 should be equal. The theoretical arithmetic mean of a truly random key stream is then $\bar{x} = \frac{0+1+2+3+4+5+6+7+8+9}{10} = 4.5$, where each digit has equal weight due to equal frequencies. To test whether the centrality of a key stream of length n is consistent with this theoretical value with 95% confidence, the observed arithmetic mean $\hat{\bar{x}}$, standard deviation s, and 95% confidence interval

$\hat{\bar{x}} \pm t_c \frac{s}{\sqrt{n}}$ of the key stream can be calculated, where t_c is the t-value corresponding to the significance level $\alpha = 0.05$ and degrees of freedom $\mathrm{df} = n-1$. If the 95% confidence interval contains the theoretical value, then the arithmetic mean test is passed with 95% confidence, in support of the randomness of the key stream; otherwise, it is failed.

2. **Chi-Squared (χ^2) Test:** The expected (theoretical) frequencies for n total digits 0–9 generated randomly are all $f_e = \frac{n}{10}$ (a discrete uniform distribution). Pearson's chi-squared test can be used to compare the distribution of observed frequencies f_O in a key stream of length n to the expected frequencies f_e. The test statistic is given by $D = \sum_i \frac{(f_{\mathrm{O},i} - f_{\mathrm{e},i})^2}{f_{\mathrm{e},i}}$, where the sum is over the $k = 10$ digits 0–9. This test statistic is compared to a chi-squared distribution with $\mathrm{df} = k-1$. If the p-value of the test is $\geq \alpha = 0.05$, then the chi-squared test is passed with 95% confidence, in support of the randomness of the key stream; otherwise, it is failed. Whereas the arithmetic mean test can be passed without equal frequencies (e.g. if the digits 0 and 9 are equally inflated), the chi-squared test detects unevenness in the frequencies. This test used the SciPy Python package (Virtanen et al., 2020). It can be thought of as an extension to decimal sequences of the Frequency (Monobit) Test for binary sequences in the NIST Statistical Test Suite (Bassham et al., 2010).[1]

3. **Runs Test:** A random key stream should not contain trends, i.e. long runs of values all above/below the median; or cyclic effects, i.e. regular oscillations about the median. A runs test can be used to test for the presence of these effects in a key stream. If the p-value of the test is $\geq \alpha = 0.05$, then the runs test is passed with 95% confidence, in support of the randomness of the key stream; otherwise, it is failed (Bradley, 1968, Chapter 11). This test used the statsmodels Python package (Seabold and Perktold, 2010). It can be thought of as an extension to decimal sequences of the Runs Test for binary sequences in the NIST Statistical Test Suite (Bassham et al., 2010).

4. **Monte Carlo Pi Test:** A circle of radius $r = 1$ and area $A_\circ = \pi r^2 = \pi$, and a square of side length $a = 2$ and area $A_\square = a^2 = 4$ are centered on the origin of a Cartesian coordinate system. The ratio between these shapes' areas is $\frac{A_\circ}{A_\square} = \frac{\pi}{4} \leftrightarrow \pi = \frac{4A_\circ}{A_\square}$. n points are placed randomly in the two-dimensional space. The number of points that fall inside the circle $n_\circ \propto A_\circ$. The number of points that fall inside the square $n_\circ + n_{\circ'} \propto A_\square$, where $n_{\circ'}$ denotes the number of points that fall inside the square but *not* the circle. In the limiting case, $\pi = \lim_{n\to\infty} \frac{4n_\circ}{n_\circ + n_{\circ'}}$, and an estimate for π for some finite n is given by $\hat{\pi} = \frac{4n_\circ}{n_\circ + n_{\circ'}}$ with approximate 95% confidence interval $\hat{\pi} \pm z_c \sqrt{\frac{\pi(4-\pi)}{n}}$, where z_c is the z-score corresponding to the significance level $\alpha = 0.05$. A key stream can be divided into successive strings of six digits (e.g. 160930, 717903, ...), which are then divided by $1{,}000{,}000$ to produce random numbers from 0.00000 to 0.999999 (e.g. 0.160930, 0.717903, ...). Successive pairs of these numbers are treated as coordinates, e.g. $(x = 0.160930, y = 0.717903)$. If the distance between this coordinate and the origin $d = \sqrt{x^2 + y^2} <= r$, then $n_\circ$ is incremented by one, else if $d > r$ then $n_{\circ'}$ is incremented by one. $\hat{\pi}$ and its 95% confidence interval are then calculated with the equations above. If the 95% confidence interval contains the true value of $\pi = 3.14159...$, then the Monte Carlo pi test is passed with 95% confidence, in support of the randomness of the key stream; otherwise, it is failed (Li and Nakano, 2022; Nakade, 2025).

2.6 Reference Streams

Two reference key streams with lengths equal to the historical tables were produced with the following methods:

1. Mersenne Twister (MT): the reproducible random number generator used in the 'random' Python package (Matsumoto and Nishimura, 1998; Python Software Foundation, 2025a).
2. Cryptographically Strong Pseudo-Random Number Generator (CSPRNG): the non-reproducible random number generator used

[1] For a similar example using the normal distribution, see Tomášek et al. (2021).

in the ‘secrets’ Python package, which “provides access to the most secure source of randomness that your operating system provides” (Python Software Foundation, 2025b).

All four tests for randomness were applied separately to the historical key stream and reference streams.

3 Results

For the convnet, 100 epochs and a batch size of 32 resulted in the optimal mean cross-validated testing balanced accuracy (93.7%, with a standard deviation of 0.7%) without evidence of overfitting (perfect mean cross-validated training balanced accuracy, which was observed for more epochs). By way of comparison, the pre-trained Tesseract classifier (configured for single digits) achieved a balanced accuracy of just 21.7% on the objects it extracted from the training image. This demonstrates the superiority of a bespoke convnet for classification of obscure text images whose properties are unlike those Tesseract was trained on.

The tables contained $N = 83,390$ digits total. Tesseract extracted $N = 81,252$ objects from all historical key stream images combined, of which the final convnet classified $n = 74,979$ as digits from 0–9, and $n = 6,273$ as noise which were removed for subsequent analyses.

Table 1 shows the results of the randomness tests for each key stream. Both reference streams (MT and CSPRNG) passed all four tests, while the historical key stream failed all tests.

Test	H	MT	CSPRNG
$\bar{x}$	Failed	Passed	Passed
$\hat{\bar{x}}$ (95% CI)	4.228 (4.207, 4.248)	4.511 (4.490, 4.532)	4.492 (4.472, 4.513)
χ^2	Failed	Passed	Passed
p-value	< 0.001	0.307	0.415
Runs	Failed	Passed	Passed
p-value	< 0.001	0.148	0.152
π	Failed	Passed	Passed
$\hat{\pi}$ (95% CI)	3.319 (3.279, 3.360)	3.133 (3.092, 3.174)	3.149 (3.108, 3.189)

Table 1: Randomness test results with 95% confidence. H means historical key stream tables. MT means Mersenne Twister. CSPRNG means cryptographically strong pseudo-random number generator.

Figure 2 visualises the Monte Carlo π test results. There appears to be a small cluster of co-ordinates from the historical key stream tables around $x \approx 0.3$, $y \approx 0.3$ (see Section 4).

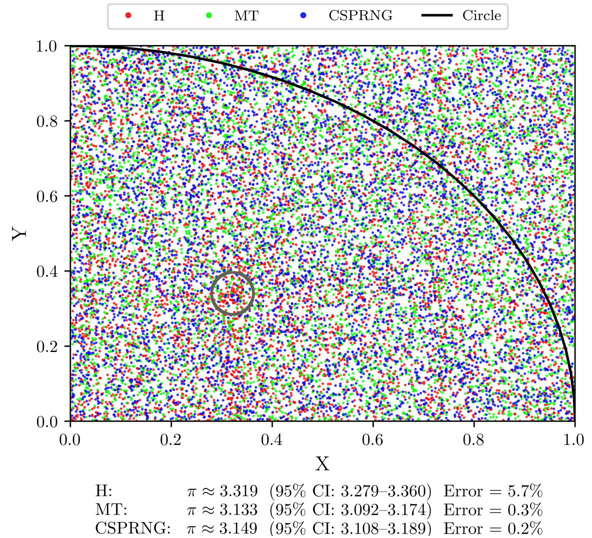


Figure 2: Monte Carlo estimates for the constant π. $N = 6,248$ points derived from $N = 74,976$ digits per method (six digits per co-ordinate, two co-ordinates per point). The dark grey ring around ($x \approx 0.3$, $y \approx 0.3$) is a visual indicator of the cluster of co-ordinates from the historical key stream tables.

4 Discussion

This study found statistical evidence for lack of randomness in a digitization of historical key stream tables used by diplomats in WW2-era Germany. The digitized key stream’s arithmetic mean was too low. Inspection of the raw frequencies reveals that this is due to over-representation/inflation of the digit 3, and under-representation of the digits 8 and 9. The over-representation of the digit 3 in particular ($n = 9,894$ compared to the theoretical $n = 7,498$) explains the cluster observed in Figure 2, as well as the failings of both the χ^2 and π tests. The failing of the runs test also implies that the digitized key stream contains trends, but the extent to which these trends could be exploited by cryptanalysts is unknown.

The major limitation of this study is that bias could have been introduced in the digitization process for the historical key stream tables; if the computer vision and/or the convnet digit classification under/over-extracted/classified one or more digits, this in turn could have biased the results of the randomness tests. E.g., if the computer vision systematically failed to recognise the digit ‘3’

(which had the highest measured frequency), or the classifier systematically mistook the digit '8' (which had the lowest measured frequency) for the similar-looking digit '3', then it may be erroneous to conclude that the historical key stream lacked randomness.

Bias was investigated by manually counting the frequency of the digits 3, 8, and 9 in the training image, and comparing these to (1) the number of Tesseract-extracted digits from this image; and (2) the number of convnet-classified digits in this image. There were $n = 64$ digit 3s, $n = 57$ digit 8s, and $n = 61$ digit 9s in the training set image. Tesseract extracted $n = 59$ digit 3s, $n = 44$ digit 8s, and $n = 58$ digit 9s intelligibly (remaining digits in the training set were extracted by Tesseract but with additional digits in the same extraction, e.g. in one case, '34' was extracted as a single object. These were labeled manually as noise). Thus, Tesseract under-represented the number of digit 3s, 8s, and 9s by five, thirteen, and three counts, respectively in the training set image. This likely explains the apparent under-representation of digit 8s (and possibly digit 9s, though to a lesser extent) in the historical tables (when computer vision and the convnet were applied to all available images), but does not explain the apparent *over*-representation of digit 3s in the historical tables, since the biasing effect from Tesseract would work in the opposite direction. The convnet classified $n = 60$ digit 3s, $n = 44$ digit 8s, and $n = 58$ digit 9s from the Tesseract extractions; thus, the convnet is unlikely to have introduced substantial bias as these are identical or very similar to the number of Tesseract extractions for each digit. Thus, the observed under-representation of digit 8 in the historical key stream may be a simple artefact of computer vision error, but the over-representation of digit 3 may not.

In conclusion, computer vision and convolutional neural networks can be used to digitize historical key streams. The particular digitized key stream in this work, the purpose of which is unknown, appeared to lack randomness in basic statistical tests. This finding may raise real doubts about the security of the key stream from a modern cryptographic perspective, but does not alone prove practical cryptanalytic exploitability. Bias introduced by computer vision and the convnet cannot be ruled out for some digits but appears less likely for others.

Acknowledgments

The author thanks Sir Dermot Turing for providing the images of the historical key stream tables and some references used in this work, the National Archives for preserving the original documents, and three anonymous reviewers for helpful comments and suggestions. Code is available in an online GitHub repository: `https://github.com/FloeFoxon/Statistical-Tests-for-Randomness-With-Computer-Vision-and-a-Convnet`

References

Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Stefan D. Leigh, M. Levenson, M. Vangel, Nathanael A. Heckert, and D. L. Banks. 2010. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute of Standards and Technology, Gaithersburg, MD. `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762`.

James V. Bradley. 1968. *Distribution-free statistical tests*. Prentice-Hall, Englewood Cliffs, NJ.

Percy William Filby. 1995. Floradora and a unique break into one-time pad ciphers. *Intelligence and National Security*, 10(3):408–422. `https://doi.org/10.1080/02684529508432310`.

Louise Foley. 2001. *Analysis of an On-line Random Number Generator*. Trinity College Dublin. `https://www.random.org/analysis/Analysis2001.pdf`.

Charmaine Kenny. 2005. *Random Number Generators: An Evaluation and Comparison of Random.org and Some Commonly Used Generators*. Trinity College Dublin. `https://www.random.org/analysis/Analysis2005.pdf`.

Keras Developers. 2015. *Keras*. `https://keras.io`.

Rongpeng Li and Aiichiro Nakano, 2022. *Calculating Pi with Monte Carlo Simulation*, pages 1–18. Apress, Berkeley, CA. `https://doi.org/10.1007/978-1-4842-8185-7_1`.

Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *CM Transactions on Modeling and Computer Simulation*, 8(1):3–30. `https://doi.org/10.1145/272991.272995`.

Apurva Nakade. 2025. Estimating π. In *Monte Carlo Methods*. `https://apurvanakade.github.io/Monte-Carlo-Methods/chapters/estimation/estimating_pi.html`.

Svetlin Nakov. 2019. Secure random generators (csprng). In *Practical Cryptography for Developers*. https://cryptobook.nakov.com/secure-random-generators/secure-random-generators-csprng.

opencv-python Developers. 2025. *opencv-python 4.12.0.88*. https://pypi.org/project/opencv-python/.

pytesseract Developers. 2024. *pytesseract 0.3.13*. https://pypi.org/project/pytesseract/.

Python Software Foundation. 2025a. *random — Generate pseudo-random numbers*. Python Software Foundation, Wilmington, DE. https://docs.python.org/3/library/random.html.

Python Software Foundation. 2025b. *secrets — Generate secure random numbers for managing secrets*. Python Software Foundation, Wilmington, DE. https://docs.python.org/3/library/secrets.html#module-secrets.

Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*, pages 92–96. https://doi.org/10.25080/Majora-92bf1922-011.

TensorFlow Developers. 2025. *TensorFlow 2.20.0*. Zenodo. https://doi.org/10.5281/zenodo.16852354.

Pavel Tomášek, Hana Tomášková, and Jakub Rak. 2021. Chi-square of Pseudorandom Number Generator of Normal Distribution in C++17. *TEM Journal*, 10(4):1495–1499. https://doi.org/10.18421/TEM104-01.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272. https://doi.org/10.1038/s41592-019-0686-2.