Jericho Comms™

Version 1.5
12 September 2015

Joshua M. David
joshua.m.david [at] tutanota.de

Abstract

In the wake of the NSA spying revelations it is clear that none of our communications are secure any longer. The NSA is actively monitoring, collecting, decrypting and indefinitely storing the whole world's communications as it transits their networks. Any traffic passing through any of the Five Eyes countries (US, UK, Canada, Australia and NZ) is monitored, collected and stored indefinitely. Any and all of your internet traffic, phone calls, email, chat messages and anything else are vulnerable to the Five Eyes spy agencies. This is no longer conspiracy theory, it is actually happening.

This is a massive breach of international human rights and it is an attack on the liberty, freedom and privacy of every person on the planet. It has huge implications for our democracy, freedom of the press, attorney-client privilege and freedom of speech. The Universal Declaration of Human Rights states clearly:

- **Article 3:** Everyone has the right to life, *liberty* and security of person.
- **Article 12:** No one shall be subjected to arbitrary interference with his *privacy*, family, home or *correspondence*, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks.

Jericho Comms is an encrypted group communications program built on the principles of information-theoretic security using true random number generation, one-time message authentication codes and mathematically proven security of the one-time pad cipher. The goal is to deliver a free, open source, plausibly deniable, encrypted communications program for journalists, lawyers, activists and citizens of the world that need *high* assurances that their communications are free of censorship, control, oppression, totalitarian governments and eavesdropping from the world's most powerful intelligence agencies. To defeat the world's top intelligence agencies, you need to lift your game to their level. That means using encryption that they can *never* break, regardless of advances in mathematics, quantum physics, cryptanalysis or computing power.

The software makes the whole process very simple from generating truly random one-time pads, key management and chatting securely. After a message is sent, the one-time pad used is automatically deleted from the computer. Once received it is also deleted from the server and receiving computers. This prevents the message being decrypted in the future if one of the computers is compromised. If a user thinks their computer is about to become compromised they can also initiate the Auto Nuke feature which will delete the one-time pads from the local database, encrypted messages from the server and the one-time pads from the chat partner's local database. These features combined provide a form of Off The Record encrypted chat. The full source code is available on GitHub at https://github.com/joshua-m-david/jerichoencryption.

The main reason why one-time pads are infrequently used outside of government and military networks is that you need truly random data, the key must be at least as long as the message and the one-time pads must be sent through a secure channel e.g. delivered in person. This makes them somewhat inconvenient to use but these are *not* insurmountable problems. This software aims to solve most of the issues that make using one-time pads difficult. The only minor issue would be delivering the one-time pads to the other user, but there are ways to do that easily enough. In today's world of complete surveillance it is necessary to have a shared secret to authenticate the correspondents and have complete assurance you are actually communicating with the right people and not an active Man in the Middle (MITM) attacker. Obviously this software is not a solution for all cryptography problems. It does however solve a specific problem and allows you to communicate securely with your family, friends, colleagues or associates in the future if you have met them at least once and exchanged one-time pads with them.

Contents

3. How it works
4. Server configuration
5. Server authentication protocol
6. Preventing information leakage and traffic analysis
7. Using TLS/HTTPS
8. True Random Number Generator (TRNG)
9. Testing the TRNG random output
10. TRNG tests and analysis
11. Custom random data loading
12. Pad storage and exporting data
13. One-time pad database encryption and authentication
14. Protection of database encryption and authentication keys
15. Using HTML5
16. REST API using JSON
17. Message encoding
18. Encryption process
19. Decryption process
20. Message authentication code (MAC)
21. Auto nuke process

Formulas and notation

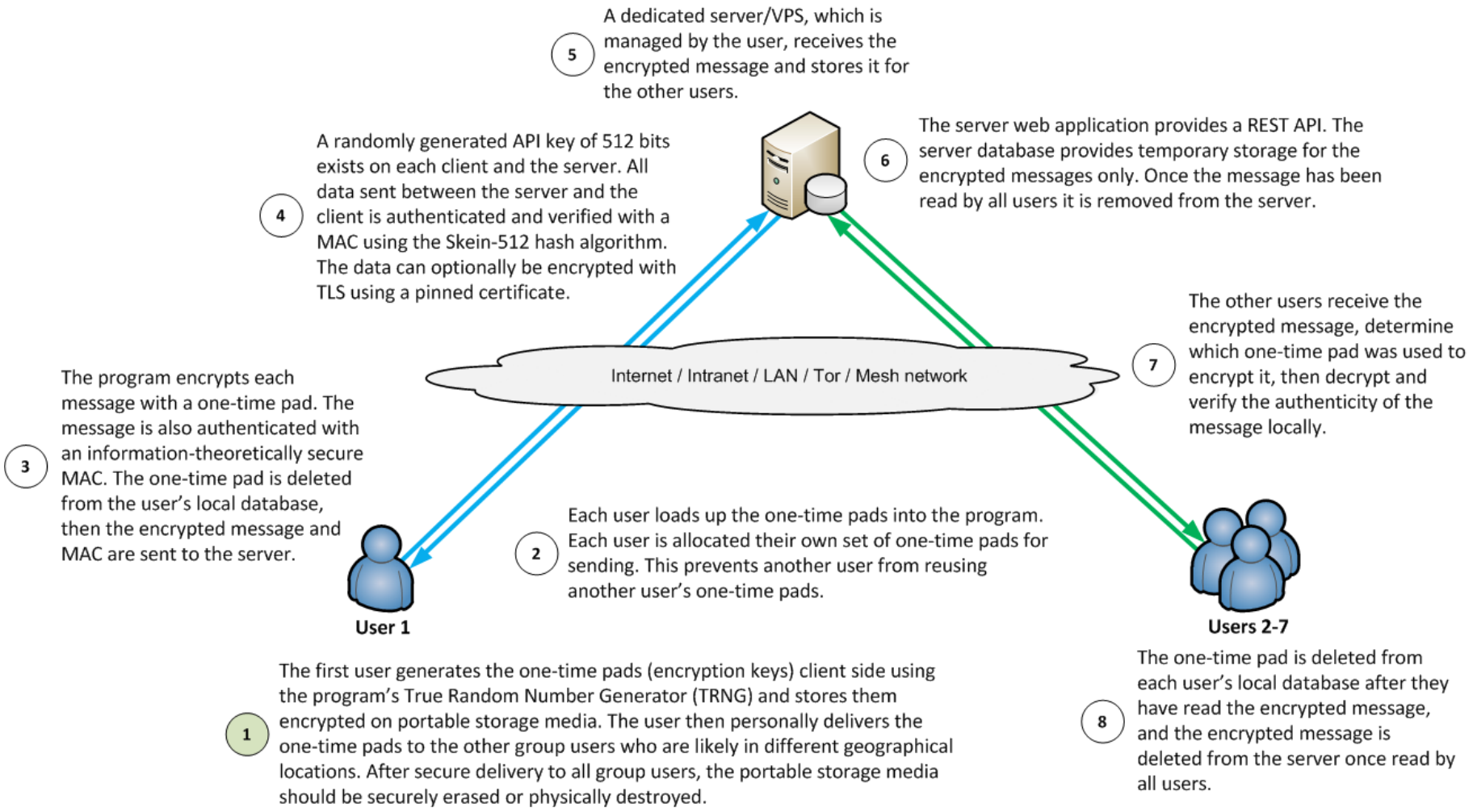
The notation used in this document will use very limited mathematical and cryptographic formulas and will tend towards explaining the design in plain English rather than mathematical formulas so that the content is more accessible and

understandable by a wider audience.

- $\oplus$ Bitwise Exclusive OR (XOR) operator e.g. $0 \oplus 1 = 1$.
- $||$ Concatenation. Usually variables will be converted to the same type and format before being concatenated together. For example, for input into a hash function all variables will be in hexadecimal format.
- $+$ Regular addition.
- $\times$ Regular multiplication.

Overall network architecture

The following will outline the entire design of the application and how it works. Exact implementation details can be found in the source code which is provided with every download.



- One particular point about this client-server architecture is that the clients are the only devices which have the one-time pads (encryption keys). The server effectively only contains encrypted data. The clients operate on a request, response basis. For this reason the clients which have the sensitive encryption keys should have their firewalls configured to block **all** incoming traffic by default and only receive incoming data as responses to outgoing requests that they have specifically

made to the server. The responses from the server are always authenticated using a shared API Key and any malformed or unauthentic responses are immediately discarded. This makes the client machines a lot more resilient to attack and exfiltration of encryption keys. Assuming the client machines are single purpose and are not used for untrusted web browsing or other activities this is a very small attack surface.

- The server should have a firewall as well which only allows incoming application data on one port and optionally on an SSH port for management. The server application expects a correctly authenticated packet from the client or it discards the request and responds with a **`HTTP/1.1 404 Not Found`** error. Of course there is the potential for a 0-day vulnerability in the server application layers but compromising the server will not compromise confidentiality or authenticity of messages as they are encrypted end-to-end. If an attacker gained access to the server they could only interfere with the server's operations to block messages from being sent or received at all.

How it works

- The first user will run a True Random Number Generator included with the software. This is explained further on. This random data is then broken up into separate one-time pads. It is possible to generate enough data for 1900+ messages in under a minute.

- Each user will be assigned pads for sending messages. This prevents re-use of a pad by another user. Re-using a pad can make cryptanalysis possible so it's very important to prevent this. The messaging protocol is explained in depth further on.

- Once the pads are generated, the program can export the pads for each user to text files. These should be saved onto removable flash media such as MicroSD, SD card, USB thumbdrive, CD, DVD or portable hard drive. There are some other precautions listed in the FAQ such as creating a TrueCrypt volume on the storage media first *before* copying the pads into it. This saves time in not needing to securely erase the pads or physically destroy the storage media after they have been transferred to the other user.

- To get the one-time pads to the other users, ideally they will arrange a physical meetup. This is to create a secure channel or air gap to deliver the one-time pads which ensures the encryption keys are not intercepted or compromised by transferring them over an insecure network like the internet. Key exchange is considered out of scope for the program itself as this is the user's responsibility. The best method is to hide in plain sight. Here are a few solutions that will usually work:

- Meeting the users in person for a coffee/drink/lunch/dinner provides an innocent cover for the exchange of encryption keys. If you are going about your daily activities this is the least suspicious option and the most likely to succeed.
- A dead drop. Arrange the time and place of the dead drop (or regularly scheduled dead drops) in person or using another method, but do not arrange this over an insecure channel like the phone or internet.
- If your country does not inspect internal mail you can hand sign a sealed courier envelope and send it with reasonable assurance that it won't be compromised. If the package appears to be opened on delivery, consider the encryption keys compromised and do not use them. International mail is sometimes opened by Customs so it is not a reliable method. It would not be difficult to hide an encrypted MicroSD within something else though.
- It is not difficult to get a laptop, portable hard drive, MicroSD card, or SD card through Customs at national or international airports. You can also hide them in your luggage or on your person. If your government generally clones or scans personal phones or computers as you are going through Customs you may need to use steganography to hide the one-time pads within your vacation photos, videos or other files.
- Governments and diplomats also have the option of using a diplomatic bag to transfer items to another country which have diplomatic immunity from search or seizure under the Vienna Convention on Diplomatic Relations.

The one-time pads absolutely cannot be sent via a less secure channel such as the internet even if you think you are using the best public key, block or stream cipher encryption there is. Your security will only be as strong as the cipher you used and you lose the perfect secrecy that the one-time pad provides.

There may be people that complain about having to deliver the one-time pads physically. If you seriously want to take your privacy back, you need to use encryption which the NSA absolutely and unequivocally can't break. If you are willing to trade security for convenience and blindly use the US government's recommended encryption algorithms then you will only receive the amount of privacy they want you to have. The US government does not have your best interests at heart, they want to monitor any and all communications and they will do that by whatever means necessary.

- As the one-time pads are in transit it is important to keep them securely on you at all times. Do not leave them anywhere unattended and keep them in a zipped up pocket to prevent pickpockets, and ideally have a tamper evident seal.
- Once the one-time pads are safely delivered and the server is set up, each user will load them into the program and begin

chatting. There are extra security considerations in the FAQ but it will be important to erase the text file containing the one-time pads from the removable media once they are loaded into the program. Storing the one-time pads and running the software from within an encrypted container on the removable media will help mitigate most issues. Portable versions of TrueCrypt and Firefox can be used for this.

Server configuration

The server basically functions as a temporary database store for the encrypted messages. One user leaves encrypted messages on the server and the other users can retrieve them when they are able. If both users are connected at the same time it is possible for realtime chat, plus or minus a few seconds. The messages are removed immediately from the server after they are read by all users.

The server is user owned and operated which means users are in complete control of the communications. No-one else knows about the server so this keeps it off the radar of the intelligence agencies, as opposed to having a central server somewhere that everyone on the internet is using. If everyone was using a single server it leaves it as a single point of failure and the intelligence agencies can raid it, hack it or shut it down with a court order. There is no useful data for them on the server but they would be able to shut down a lot user's of communications at once, at least until someone set up a new server.

The server code provides a REST API using JSON which will run on the Apache 2.2+ web server, MySQL 5.5+ database and PHP 5.3+ programming language. PHP was chosen because it is a memory safe language, fast to develop in and fast to deploy along with a basic LAMP stack. In a future version, the design may be ported to another server side language, database and web server that are considered more secure. The straightforward JSON REST API interface should enable the client to connect with multiple different backend code bases and users can implement the one of their choosing.

At the moment, to get the server side running all that is needed is to install a basic LAMP stack, copy the server files over to the web root directory, run the database script to create the tables, then finally edit the config file with the server's database and API login credentials. An easy to use guide with screenshots for Ubuntu server has been created for this. Users can install a server their own network, or pay for a cheap VPS somewhere. It is not recommended to use a VPS in the US to rule out the possibility of the provider being issued with a National Security Letter which would compromise security of the server.

Server authentication protocol

A new protocol from version 1.3 was designed to securely authenticate the requests and responses with the server API. This replaces TLS which secured the server API credentials (username & password) in transit in versions up to 1.2. The reason for this was mainly to simplify installation, avoid Certificate Authorities, have resistance against active MITM attacks and quantum computers. There have also been major security problems with common TLS implementations such as OpenSSL, GnuTLS and iOS which have led to a loss in confidence in these programs which have poor code quality and may be hiding other NSA backdoors.

Design goals

- Authenticate all API requests to the server to verify they are from valid server users.
- Authenticate all API responses from the server to verify the response came from the legitimate server, not an attacker.
- Mitigate passive MITM attacks where an attacker tries to steal the API credentials in transit.
- Mitigate active MITM attacks where an attacker attempts to send fake responses to/from the server or impersonate the server.
- Mitigate replay attacks and reject any request/response if the data was modified in transit.
- Prevent a request to the server being modified by an attacker to perform a different API action.
- Mitigate quantum computer attacks on the protocol.

Limitations and assumptions

- The server administrator will control the server and be a user of the chat group's operating on that server as well. The server administrator will be a trustworthy person not interested in interfering with his own chat group's communications. If a group of users want to communicate with each other, but the server administrator is not included in that group, then they should set up their own server. This rules out the server administrator having the will to interfere with communications.
- Users of the server have an interest in keeping the shared API key on the server a secret to protect their own

communications so they will not give that key to anyone else.

- There is no need for each user having a separate API key on the server to send/receive requests because the server administrator could access that key anyway and impersonate them or simply edit the database record to alter which user the message came from.

- There is trust between the users in the group communicating not to impersonate other users in the chat group. Because every user in a group has access to all the same one-time pads and same API key, it would technically be possible to pretend to be one of the other users in the chat group. This also doubles as a deniable authentication protocol because every message sent could have been engineered to come from one of the other users in the chat group.

- Encryption for the server protocol is not required because individual messages are encrypted using the one-time pads and signed on each client machine. Some anonymised, non-critical meta-data is viewable if an attacker in a privileged network position is passively watching the traffic. This may indicate a chat conversation is taking place using this protocol. If this is a problem, the user can easily add TLS to the connection and use a pinned certificate. Version 2.0 of the program will encrypt the connection between the client and the server using a cascade stream cipher to hide the meta data and to avoid relying on TLS. Hiding the meta data by sending only encrypted binary blobs between the client and server will make traffic analysis and fingerprinting much more difficult for spy tools like XKeyScore. This means TAO agents will not know whether a client machine is running this program or any other program sending binary data. This lowers the likelihood of them being able to target the client machines with a specific exploit for this program to extract the one-time pads or tamper with them.

- The server protocol does not anonymise IP addresses from users connecting to the server. If there is a requirement for anonymity, then users can tunnel their connection through a SOCKS5 proxy in their web browser or use Vidalia to tunnel their connection through the Tor network.

Initial setup

- A random 512 bit *API Key* is created using the TRNG included with the program and entered into the configuration file on the server. The user can use SSH to access their VPS, but ideally to get the key securely onto the server, the server could be hosted on their local network running a web server with a static public IP that is serving to the wider internet. Storing the key in the configuration file prevents SQL injection attacks to retrieve the key and also it saves a database lookup each request.

- The *API Key* is given to each user in person (not using a key exchange protocol or sent via an insecure network). This will be done in the initial key exchange between users as the program can store the *API Key* and server address along with the one-time pads.
- The protocol caters for 2 - 7 users per group using the server. If additional chat groups are required on the same server this can also be setup.

To send and verify an API request

- User creates a random 512 bit per request *Nonce*. The server keeps track of sent nonces to prevent replay attacks. The nonce is created using the Web Crypto API getRandomValues() method.
- User creates an API request including a group of data variables to send to the server as part of the *Message Packet*. For example, this can contain the one-time pad encrypted message and its MAC that the server will store.
- Each request is sent with an *API Action* to perform on the server. This prevents the attacker from changing what action to perform on the server because any change to the data packet will alter the MAC.
- The *From User* data indicates which user the message is from. The server uses this to retrieve the correct key. All users on the server are coded to NATO phonetic alphabet names i.e. Alpha, Bravo, Charlie, Delta, Echo, Foxtrot and Golf. This allows some anonymity when multiple servers around the world are using the same protocol. When exporting the one-time pads, the user can assign call signs/nicknames to the chat users within the group to override the default names. These nicknames are not transmitted to the server because they are kept in the user's local storage next to the one-time pads.
- A *Sent Timestamp* is included to indicate when the request was sent. This is a UNIX timestamp therefore both the client and server code use UTC time.
- The server rejects messages received that are received more than +/- 60 seconds of the received UNIX timestamp. The server and clients should be synchronized to an NTP server. The 60 second allowance should be enough to counter any clock drift and subtle differences between the server and client clocks.
- The server rejects duplicate messages/replay attacks received within the allowed time window by storing all received nonces and checking incoming message timestamps against past received nonces. If the nonce is the same and same

request is received twice, then the second request will be invalid. Sent nonces are kept on the server for at least 120 seconds and then discarded by a separate cleanup process.

- These variables are stringified into JSON and a MAC is calculated using version 1.3 of the Skein 512 bit hash function on the JSON data:

  MAC = Skein-512(*API Key* || *API Action* || *From User* || *Nonce* || *Sent Timestamp* || *Message Packet*)

- The client sends the JSON data and the MAC to the server. The server receives the request, looks up the username and verifies it is a valid user for that chat group, then verifies the request by recreating the same MAC with the data provided.

- The server rejects invalid MACs, which will also mean any attempt to modify the data sent will fail. The MAC validation method in the API is protected against timing side channel attacks using a similar method to Double HMAC Verification. It uses a single hash with the *API Key* and the *Skein-512* hash function rather than using HMAC.

- Failed requests can be re-sent manually by the client which will use a different *Nonce*, *Sent Timestamp* and *MAC*.

- Version 1.5 also Base64 encodes the entire packet before sending in preparation for version 2.0. In version 2.0 the packet will also be encrypted between the client and server and then Base64 encoded. The data going back and forth will essentially just be a random binary blob. This will help disguise meta data and make it much harder for spy agencies to fingerprint the traffic as definitely originating from this program and prevent them targeting the client machines or server with automated TAO hacking tools or malware that has been mentioned in the Snowden revelations. The current version 1.5 will just provide an extra annoyance to the Five Eyes agencies who will need to update their XKeyScore detection rules and decode the Base64 data first in order to analyse it.

Server API response

- On any valid server requests, the server signs the response with the *API Key* so that the user knows the response from the real server. Invalid requests throw an **`HTTP/1.1 404 Not Found`** error. This means an attacker does not even know if there is a valid application or web page on the web server if they do not have the *API Key* to make a valid request. All invalid requests respond with the server pretending it is unable to find what the attacker is looking for. This is similar to standard firewall behaviour by just failing to respond at all when an attacker is scanning for open ports on your machine, rather than

responding with a distinct 'closed' response which indicates there is something there.

- The *Server Message Packet* contains data sent from the server including status responses and the one-time pad encrypted messages.
- The original *Client Request MAC* is included in the server's response MAC calculation so the client can be sure it is receiving a response to the original request.
- The *Server Message Packet* is stringified into JSON and a MAC is calculated using the Skein 512 bit hash function on the JSON data:

  $$MAC = Skein\text{-}512(\ API\ Key\ ||\ Server\ Message\ Packet\ ||\ Client\ Request\ MAC\ )$$

- If the MAC does not match on the client then the response is not actually from the server and will be discarded. A warning will be shown to the user that interference has occurred.
- On a successful request and response the client will process the data received from the server, decrypt any received messages, check for XSS attacks and render the messages on the client.

Using Transport Layer Security (TLS) / HTTPS

Because of the new server authentication protocol in version 1.3, using TLS is not mandatory anymore. However it can be added as an optional layer of security to help mitigate monitoring from lower level attackers, for example when using the program at work, public WiFi or home connection where your employer, a casual hacker or your ISP could monitor your connection. TLS will not stop a powerful attacker like the NSA or GCHQ as it is possible they have obtained copies of the root keys for most Certificate Authorities anyway by using National Security Letters, therefore they can perform an active MITM attack as traffic is passing through the Five Eyes alliance countries (USA, UK, Canada, Australia and NZ). It is also possible they have quantum computers by now. The public key exchange protocols used in TLS are vulnerable to quantum computers, as are most of the common cipher suites which use symmetric keys of only 128 bits. Key lengths of 256 bits are the minimum required to remain safe against quantum computers in the immediate future. Care must also be taken to use a good cipher suite order to have forward secrecy and use the highest quality ciphers available.

Users still wanting to use TLS and don't mind the extra effort to configure it are recommendeded to generate a strong (4096+ bit) self-signed certificate themselves, install it on the server and deliver the fingerprints of the certificate to the chat group users at the same time as the one-time pad key exchange which they can manually verify when first connecting to the server.

Preventing information leakage and traffic analysis

In version 1.5, each client now sends decoy messages at random intervals to other users in the chat group to prevent information leakage (e.g. when real messages are sent) and to frustrate traffic analysis. This technique is similar in principle to the Chaffing and Winnowing cryptographic technique. On starting a chat session, each client will generate a random number between 1000 and 90,000 using the browser's CSPRNG. A timer will be started and after this number of milliseconds have elapsed the program will generate a random string of bytes up to 56 bits bits in length. Using the accuracy of milliseconds rather than seconds gives more randomness for each timer interval. Otherwise every decoy message would be sent on evenly rounded seconds and an attacker could determine that they were decoy messages.

The program will then check if these 56 bits exist as a pad identifier in their own set of one-time pads. If that pad identifier already exists, which is quite rare (1 in 2^{56} chance), then it will skip sending a decoy message for this interval, generate a new random number and try again after that many seconds have elapsed. If the pad identifier does not already exist, then the program will send a decoy message to the other users in the chat group. The other chat group users will safely ignore the decoy message as the pad identifier does not exist in their copy of the one-time pad database for the user that is sending the message. To send a decoy message the program simply generates a further random 1480 bits using the browser's CSPRNG and concatenates that to the end of the 56 bit random pad identifier, thus forming a random string consisting of 1536 bits which is the same size as a regular one-time pad and message packet. This method also avoids burning real one-time pads on decoy messages which would be wasteful.

The random message packet is sent as is to the server and left on there for the other chat group users to collect. It effectively looks no different than a regular message being sent. The other chat group users will download it, determine it is not a real message because the pad identifier does not exist in the set of pads belonging to that user and discard it. The other users may not be saying anything but each client that is connected to the server will be sending decoy messages to other users in the group at random intervals. If two or more clients are just left unnattended it can look like an entire conversation is taking place without doing anything. This disguises when real messages are actually sent by the users. An outside attacker that can monitor all network traffic has no way of determining whether a message sent or received is a decoy or a real one.

If a client is online but there have been no real or decoy messages received from other chat clients for 3 minutes then the decoy timer will stop. This is so it doesn't appear like one user is just talking to themself. If another client comes online again and sends a real message or decoy message then the first client will start up their timer again on a random interval to keep sending decoy messages. This continues until a user quits the program or all other users are offline for over 3 minutes. This is twice the length of the maximum random timer interval (90 seconds) so there is some overlap when users are coming online/offline. These intervals may be customisable via the UI in future versions or further optimised for network bandwidth usage.

An added bonus of this functionality is that if a user has come online and received a new decoy message from another user they can know that the other user is most likely online (+/- 90 seconds) without any other kind of signalling protocol being needed. The program will also show that the other user is online immediately if a real message is received within the last 90 seconds as well.

True Random Number Generator (TRNG)

Nature is random, unpredictable and always changing. For example, the sand on the seashore changes every time the tide rolls in. Wave crests and currents change with the weather and tides. Deciduous trees change depending on the season of the year. Trees and leaves move in the wind. The sun strikes things in different angles and intensity throughout the day, casting shadows in various directions. The variance of cloud cover and light conditions alters the appearance of everything in different ways.

This section describes the True Random Number Generator (TRNG) included with the program which gathers the entropy contained in the shot noise of a high resolution photograph from a digital camera to obtain truly random data, then it runs a randomness extractor on the data to ensure a uniform distribution. The shot noise contains random data because the act of capturing something, converting it from analog to digital, and storing it in 24 bit values creates randomness due to the fact that the process creates noise on the least significant bits. Randomness tests are run on the random data after the randomness extractor has run to verify the quality.

From a photographer's position they have a unique viewpoint of a scene in nature. They can take a photo of anything in nature, giving an infinite number of possible photographs. The proposed method is to take a photograph of something that is random in nature such as a macro (close-up) shot of sand on a beach, grass, rocks, trees blowing in the wind, wave crests in

the ocean, or waves crashing against the sea shore. The photo should be taken with high resolution and in focus using a standalone digital camera.

The best results are obtained capturing the photograph using a digital camera's RAW mode then converting the RAW image file as-shot (without post-processing) to a lossless format like PNG/BMP for the TRNG program to process. Saving in a lossy algorithm like JPEG may have unintended side effects from performing optimisations on the photograph. Modern digital cameras in cellular phones may not be suitable as they usually compress the photo using JPEG. Users can however verify the processed results and check if the random data passes the required randomness tests.

TRNG photo processing algorithm

This process describes the full algorithm:

1. Load the user's photograph into memory and store it in a sequential array of RGB values.
2. Get the red, green and blue (RGB) integer values for each pixel. This will return a number between 0 and 255 for each colour.
3. Remove sequentially repeating black (0, 0, 0) pixels, white (255, 255, 255) pixels and pixel colours that are exactly the same. This removes sections of the photograph with underexposed pixels, overexposed pixels and those that have have little entropy. Generally it is very unusual to have adjacent pixels that are exactly the same colour unless there is a hardware failure, or the section of the photo is under/overexposed. Usually in a good quality photograph there are at least very slight colour variations in adjact pixels. This step removes these low entropy areas.
4. Estimate 1 bit of input entropy per set of RGB values (1 bit per 24 bit pixel). This is a very conservative estimate. Users can increase this in the TRNG settings to 3 bits per pixel which would account for the entropy in the least significant bit of each colour.
5. Gather 512 RGB values to get an estimated entropy input of 512 bits.
6. Convert the 512 RGB values to their hexadecimal representation and input them into a cryptographic hash with a 512 bit output. The user can choose which hash to use and the program allows either Skein or Keccak [c=2d]. Both are very strong finalist algorithms from the NIST hash function competition. Store this hash output of 512 bits as the temporary seed into

the next hash.

7. Start a loop:
    i. Check there is enough new input entropy for the new hash, or break out of the loop.
    ii. Get the temporary seed from earlier.
    iii. Get a new set of 512 RGB values (512 bits) as the new input entropy.
    iv. Concatenate the seed and input entropy together and hash them using: *Hash*(*seed* || *input entropy*).
    v. Append the first 256 bits of the hash output to the output random data.
    vi. Update the temporary seed to be the last 256 bits of the hash output.
    vii. Return to start of the loop.

It is important to note that the program does not use this collected entropy to seed a psuedo-random number generator to give an unlimited amount of random data. The program aims to be a true random number generator so only quality randomness is wanted and each uniformly random bit must be used solely to encrypt one bit of the plaintext. It's assumed that most psuedo-random number generators or even CSPRNGs that stretch out the entropy could produce a subtle bias in the output and allow the NSA or other governments to decrypt part or all of a message. With this program the aim is to avoid stretching the available entropy over more bits.

Testing the TRNG random output

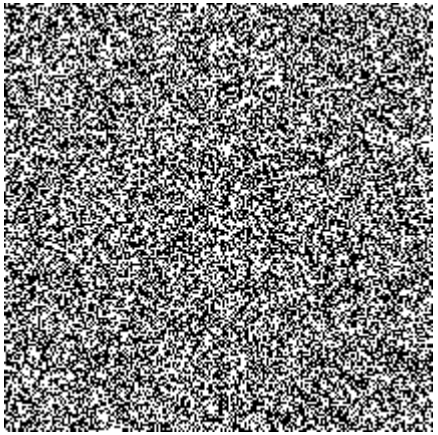
An important part of a random number generator is the ability to test the output. A few methods have been provided for this:

- Extracting the random data in various formats (binary file, hexadecimal & Base64) gives users the opportunity to verify the data with another randomness testing program. That will allow them to run additional tests and assure them of the quality of the random data.
- For testing the random data there are some statistical random number generator tests included from FIPS 140-2. These

include: The Monobit Test, The Poker Test, The Runs Test and The Long Run Test. The goal is to add more automated tests in the future to prove the quality of the program and improve on it in future versions of the software. The FIPS 140-2 test suite is run on every 20,000 bits in the extracted random data. If any of the tests fail after this, then the original source photograph was definitely not good enough and another one will need to be used.

- The output of the TRNG can also be viewed as a bitmap image which lets a user do a simple visual analysis of the output. Random.org explains that people are really good at spotting patterns and visualising the random data allows them to use their eyes and brain for this purpose. It also gives a rough impression of the TRNG's performance.

  To produce this, all the random data is converted to binary and rendered as a bitmap. A black pixel indicates a '1' bit and white pixel indicates a '0' bit. The output should look something like the image below at 100% zoom.



TRNG tests and analysis

For our testing we used a 12 MP Canon G9 digital camera. This camera is a high-end compact digital camera and allows taking photographs in RAW file format. Photos were taken in manual mode using the RAW file format. The photos were then loaded into Photoshop, converted as is to PNG which avoided using the camera's default lossy JPEG file format. Then tests were run by taking photographs of various places and things, processing the photos with the TRNG and running the test suite on everything. The results are presented below.

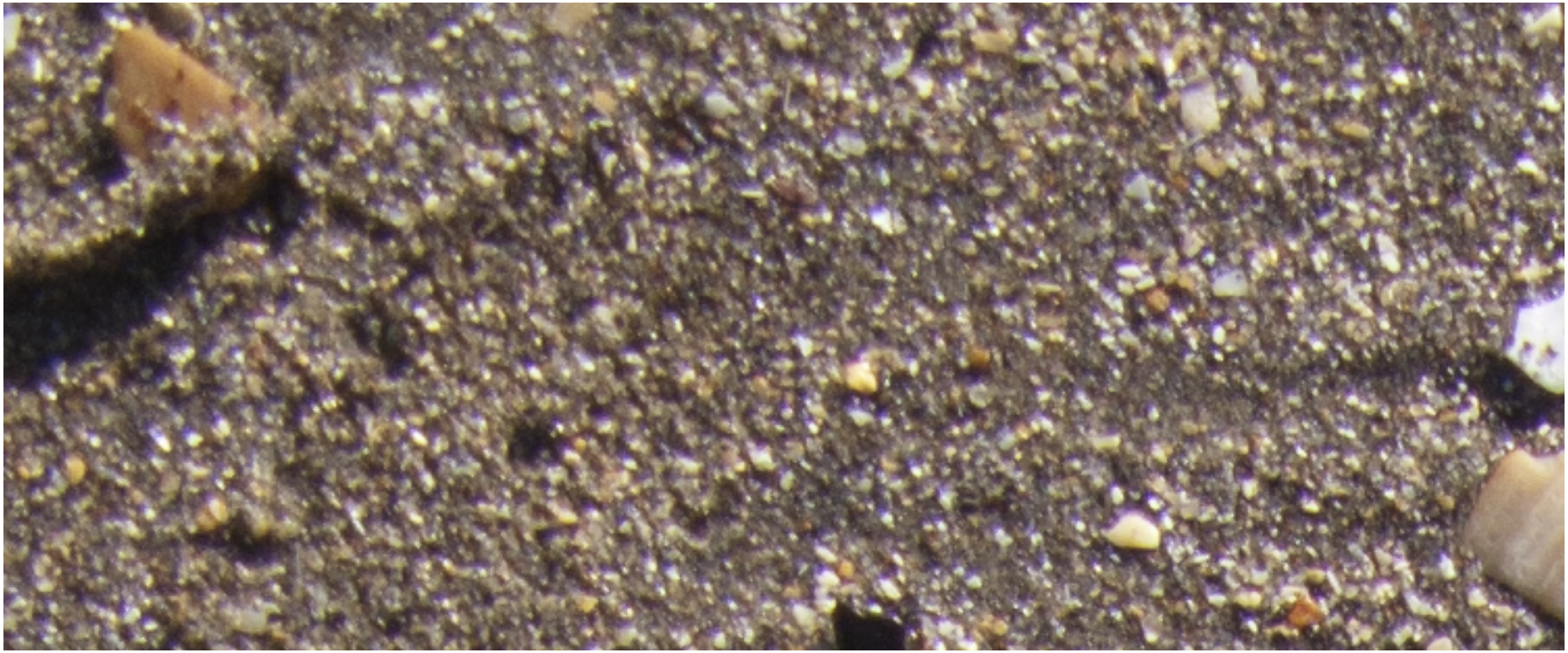
The original camera images were 4000 x 3000 pixels, then have been cropped to 1000 x 750 pixels for faster processing and to save bandwidth. The photos were processed with the Skein 512 bit hash algorithm and an estimate of 1 bit per pixel entropy input.

Click on the thumbnail images below to see a bigger image.

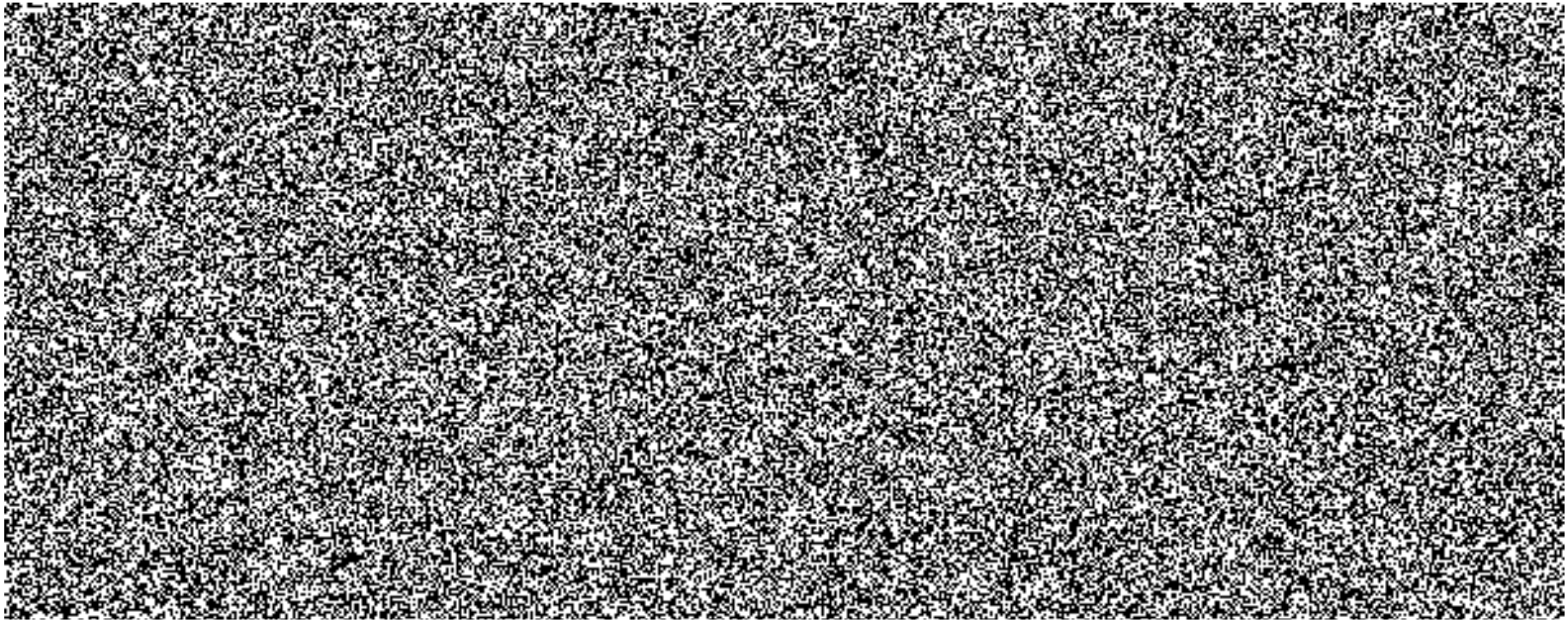
Macro shot of sand at a beach

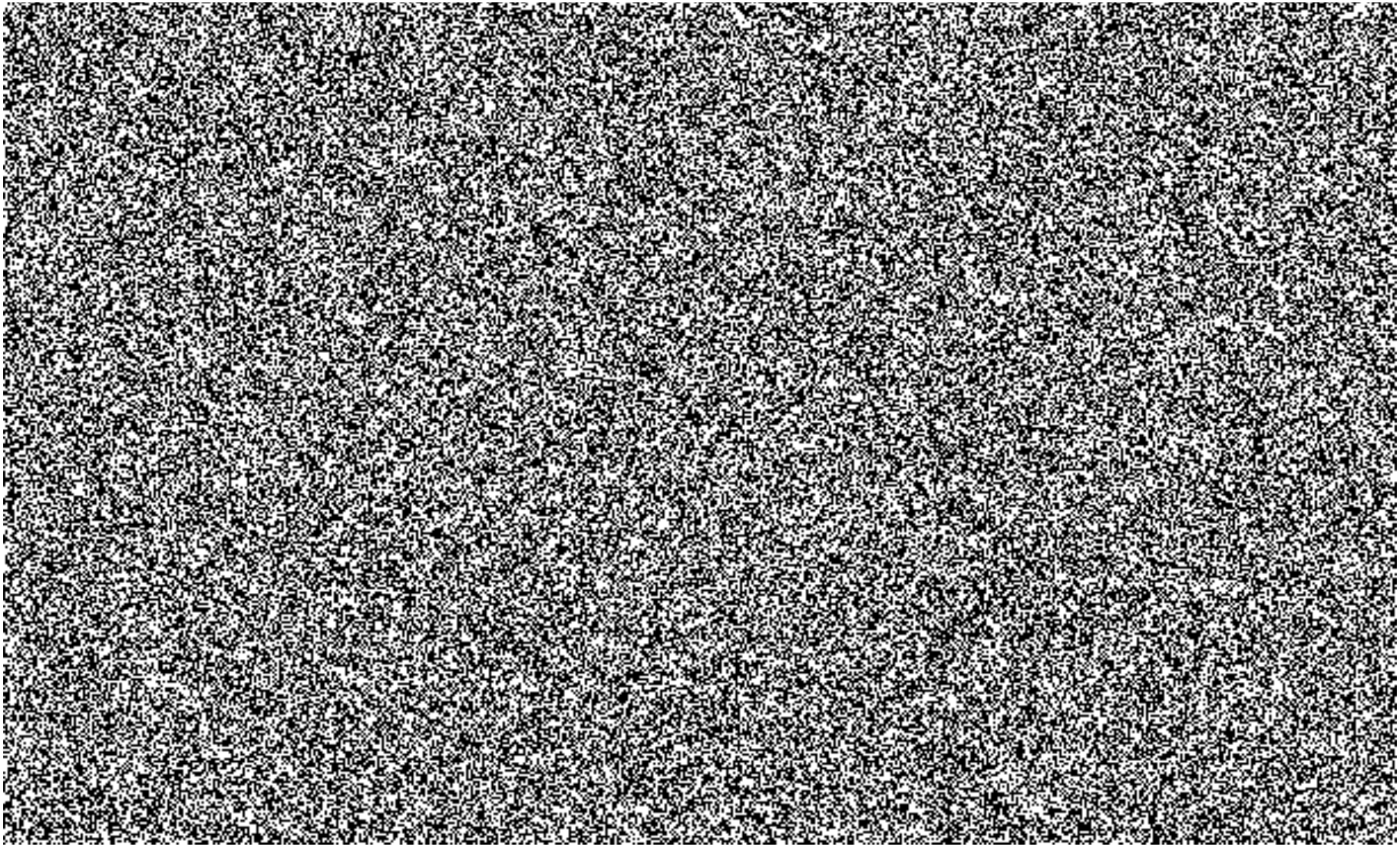
This is an example of the TRNG output from processing a macro shot of sand at a beach. See the full test results output here as a text file. This passes all the randomness tests and is one of the best photograph sources.

Original photo



Extracted entropy





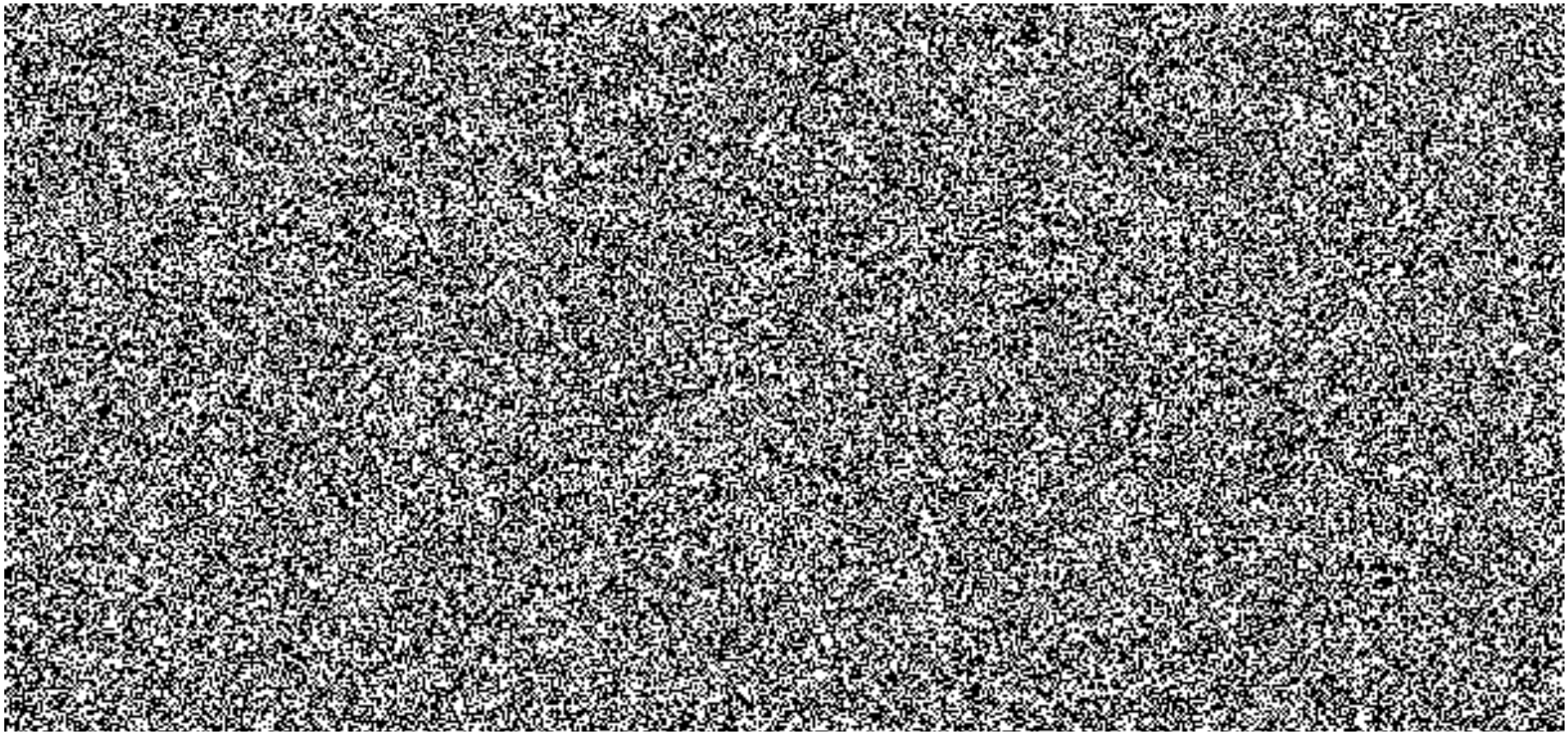
Extracted entropy FIPS 140-2 tests: **Pass**

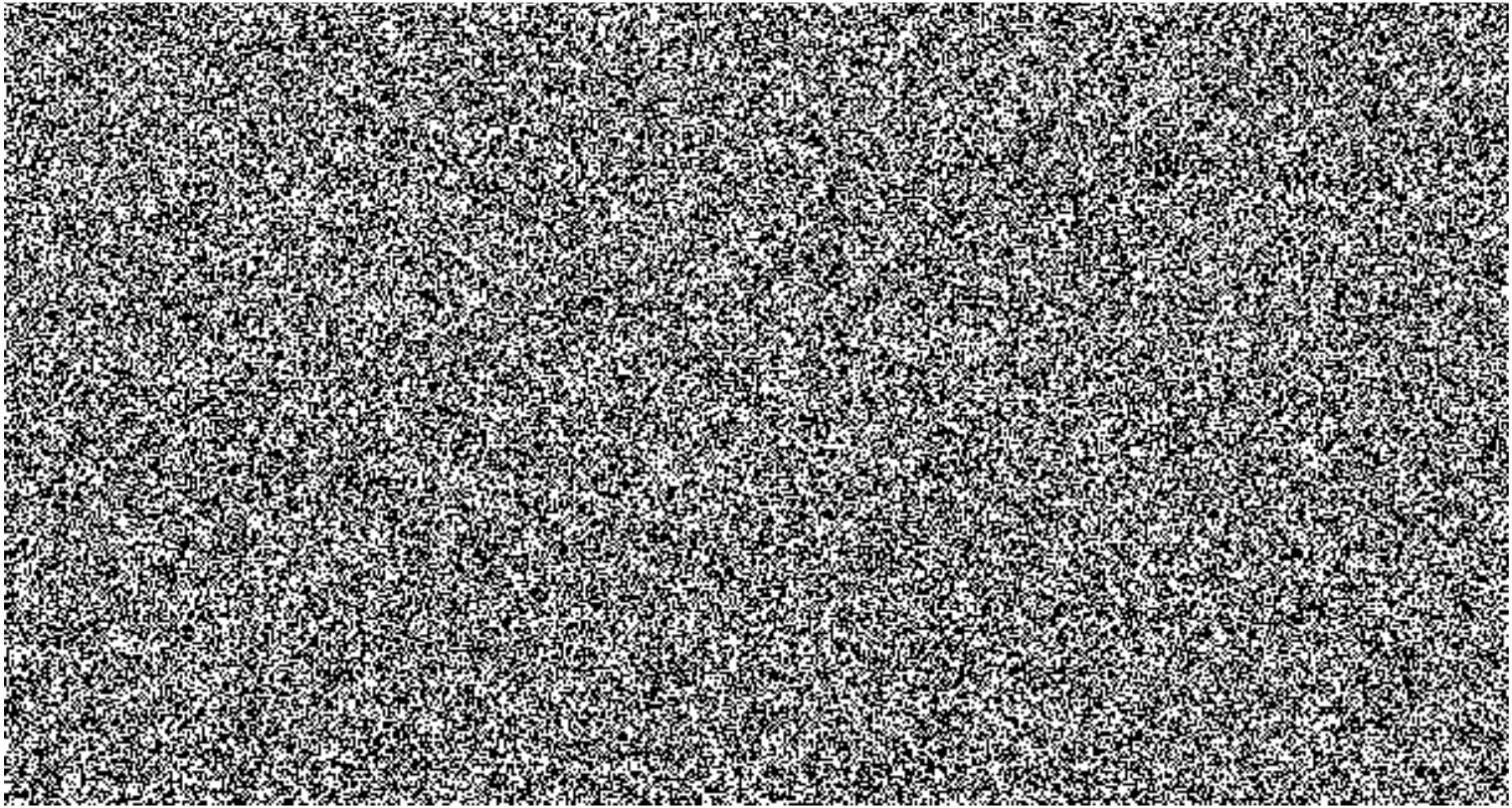
Macro shot of grass

This is an example of the TRNG output from processing a macro shot of grass in a field. See the full test results output here as a text file. This passes all the randomness tests and is one of the best photograph sources.

Original photo

Extracted entropy





Extracted entropy FIPS 140-2 tests: **Pass**

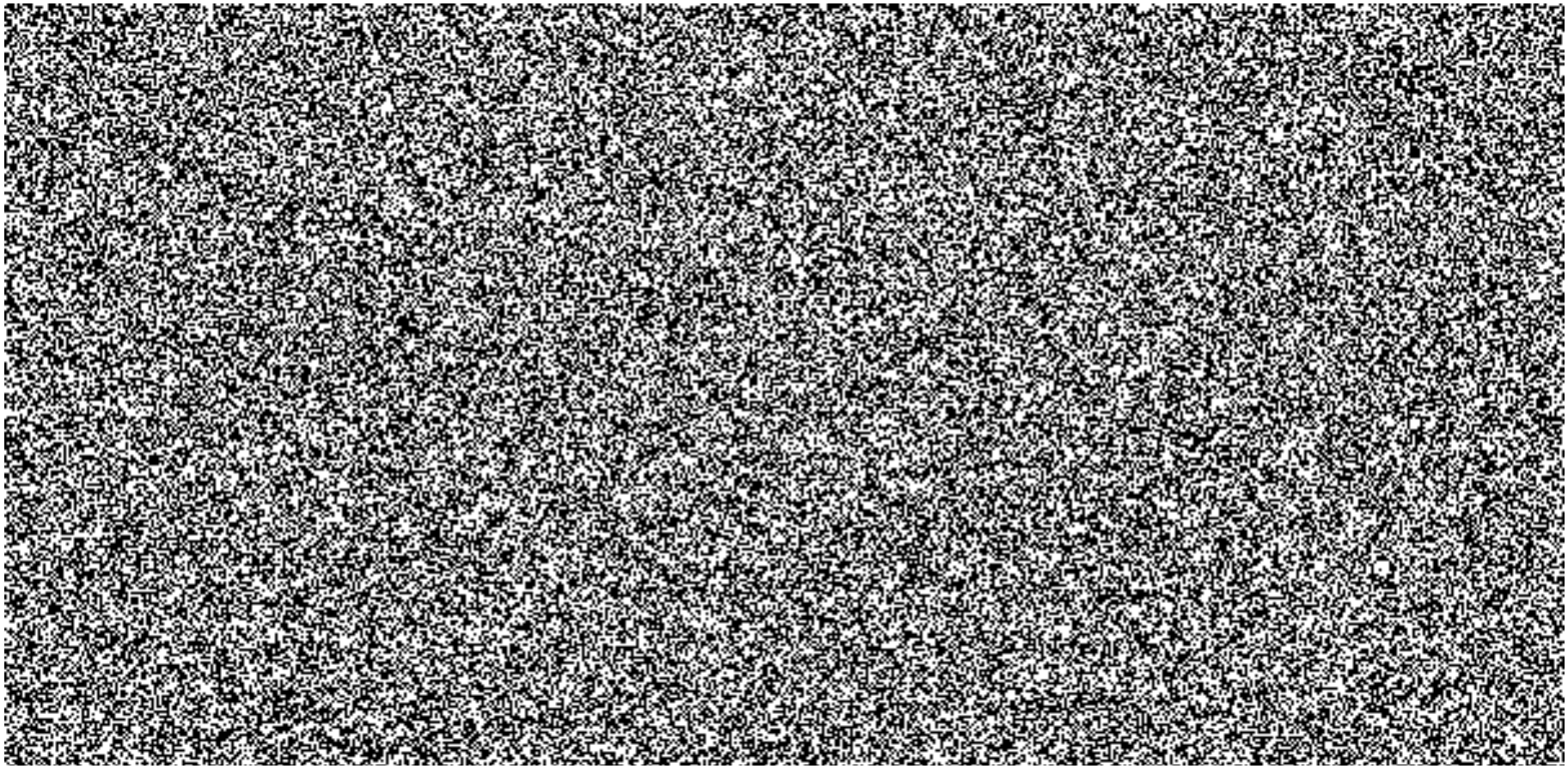
Photo of trees

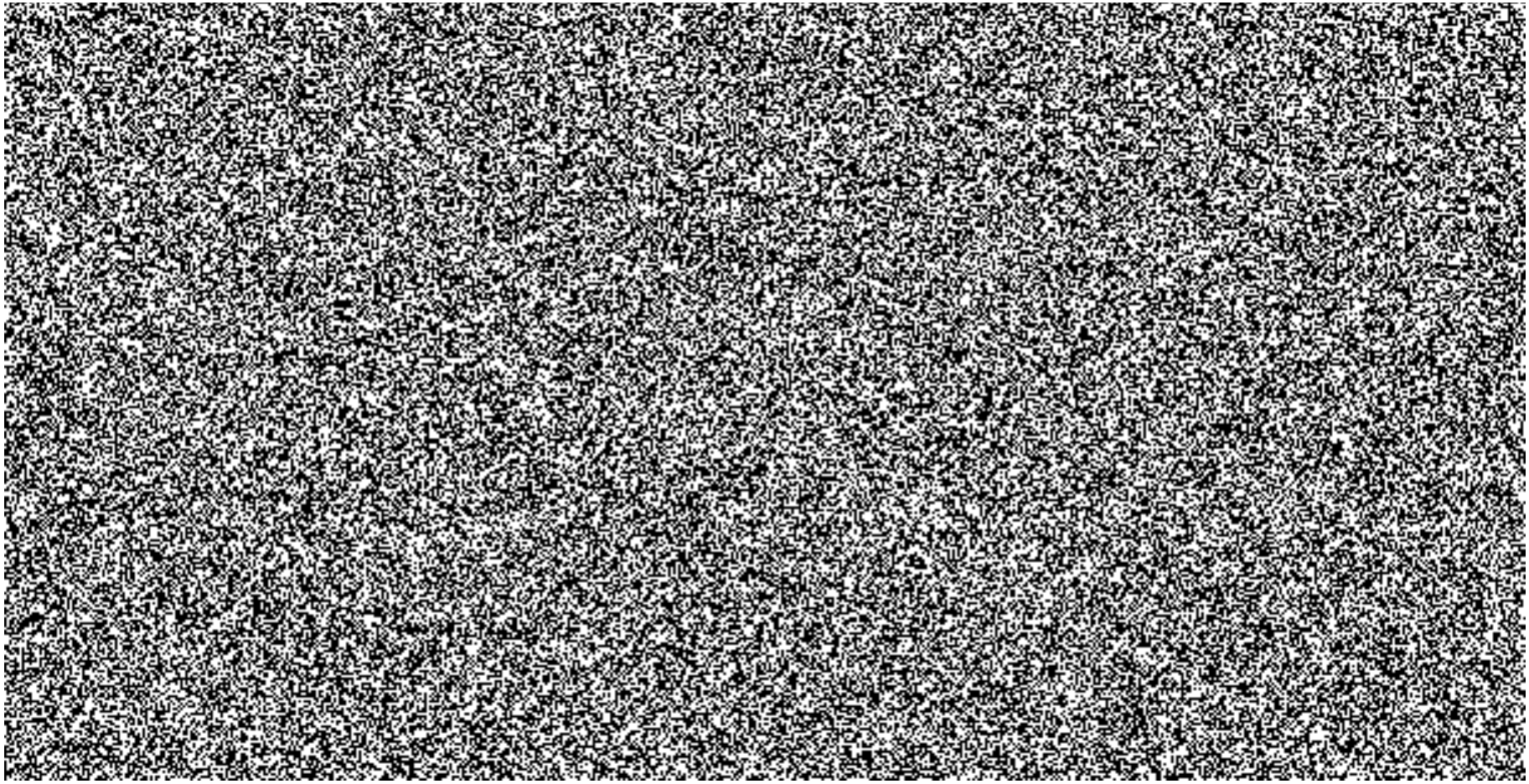
This is an example of the TRNG output from processing a photo of trees. See the full test results output here as a text file. This is not a particularly good photograph due to it being overexposed with some of the bright cloud showing through the trees, thus the overexposed sections will be a single repeated colour. The algorithm will remove these overexposed sections and the

extracted output still passes the tests, however there is less usable random data than normal.

Original photo

Extracted entropy





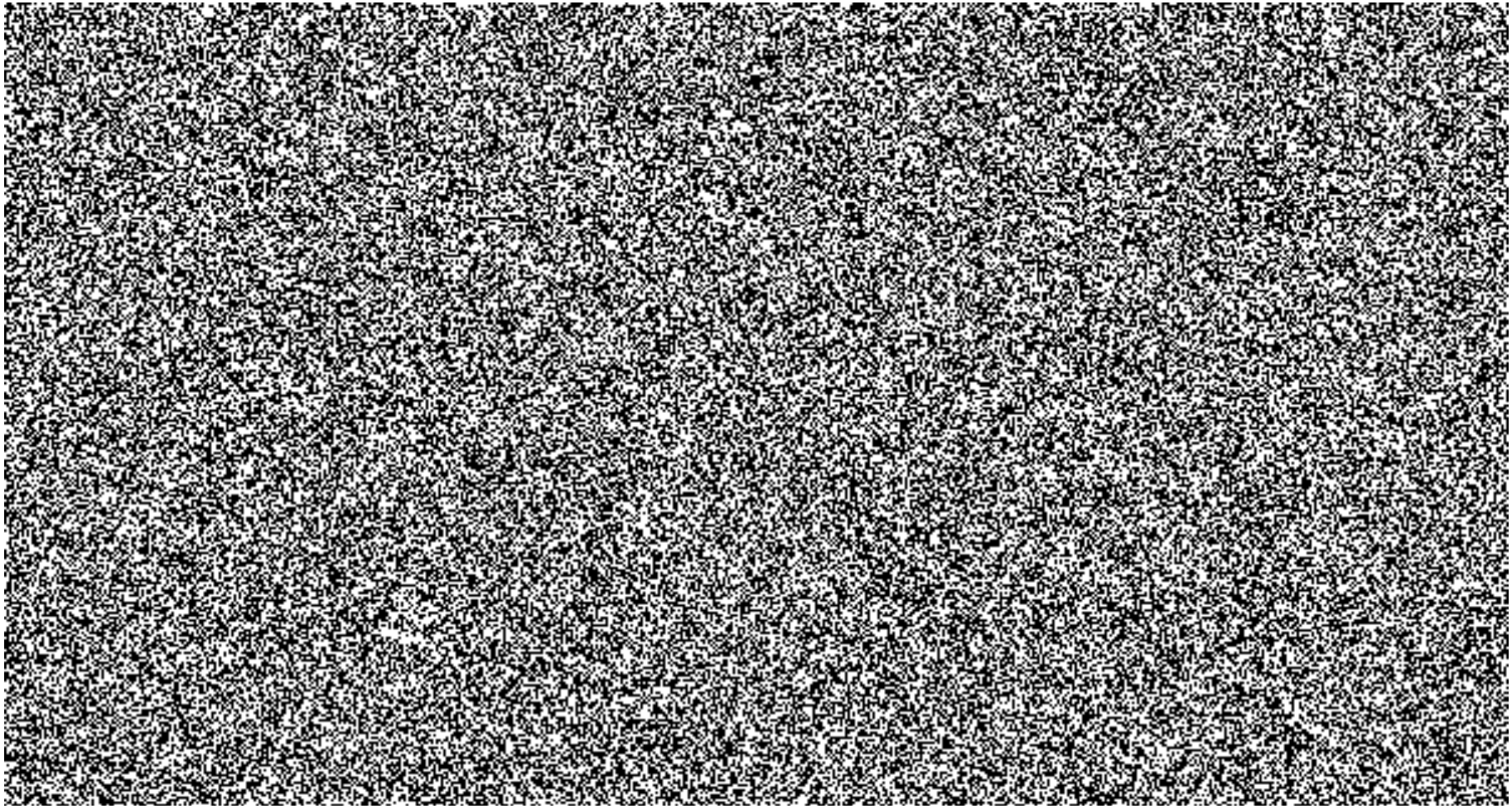
Extracted entropy FIPS 140-2 tests: **Pass**

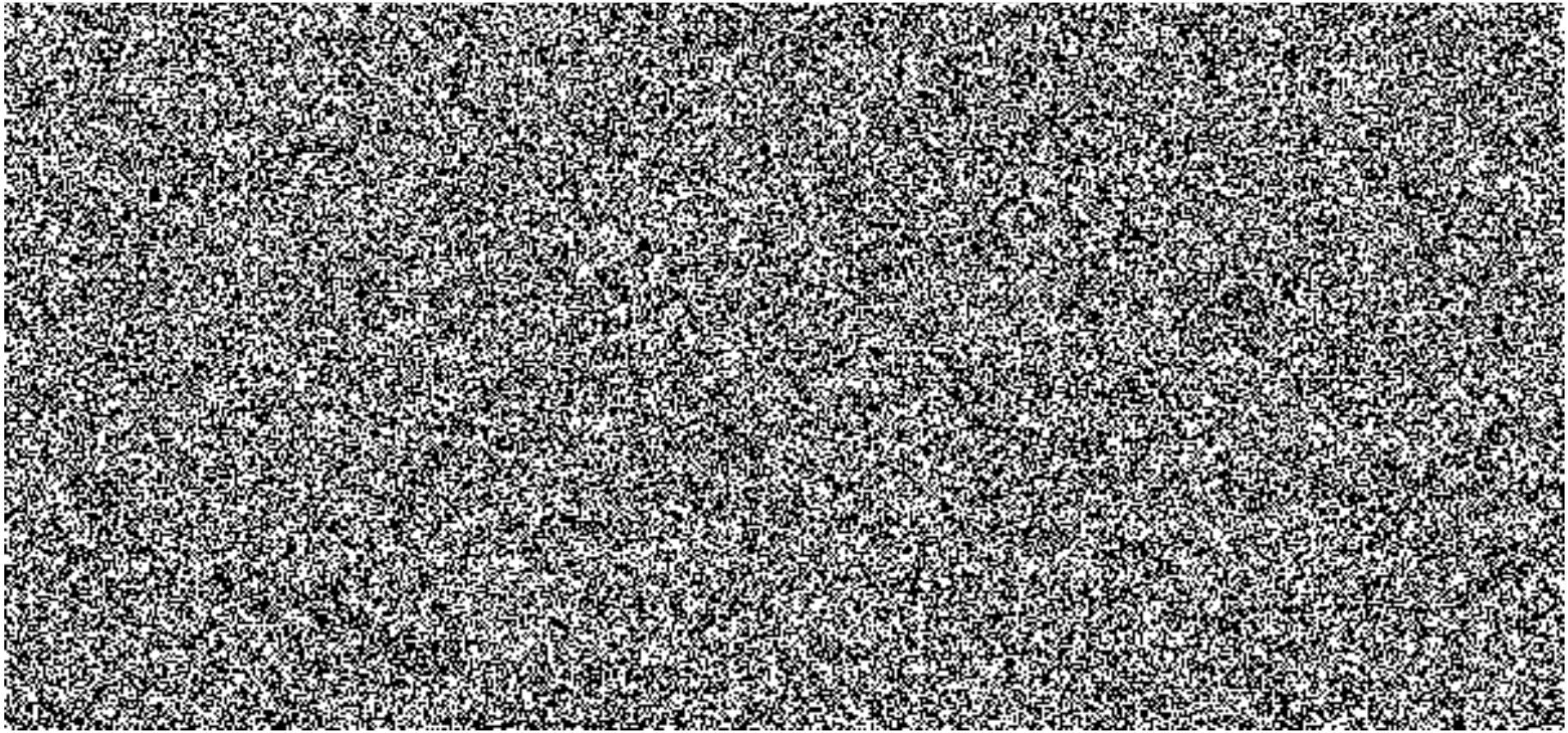
Photo of water

This is an example of the TRNG output from processing a photo of water in a harbour. See the full test results output here as a text file.

Original photo

Extracted entropy





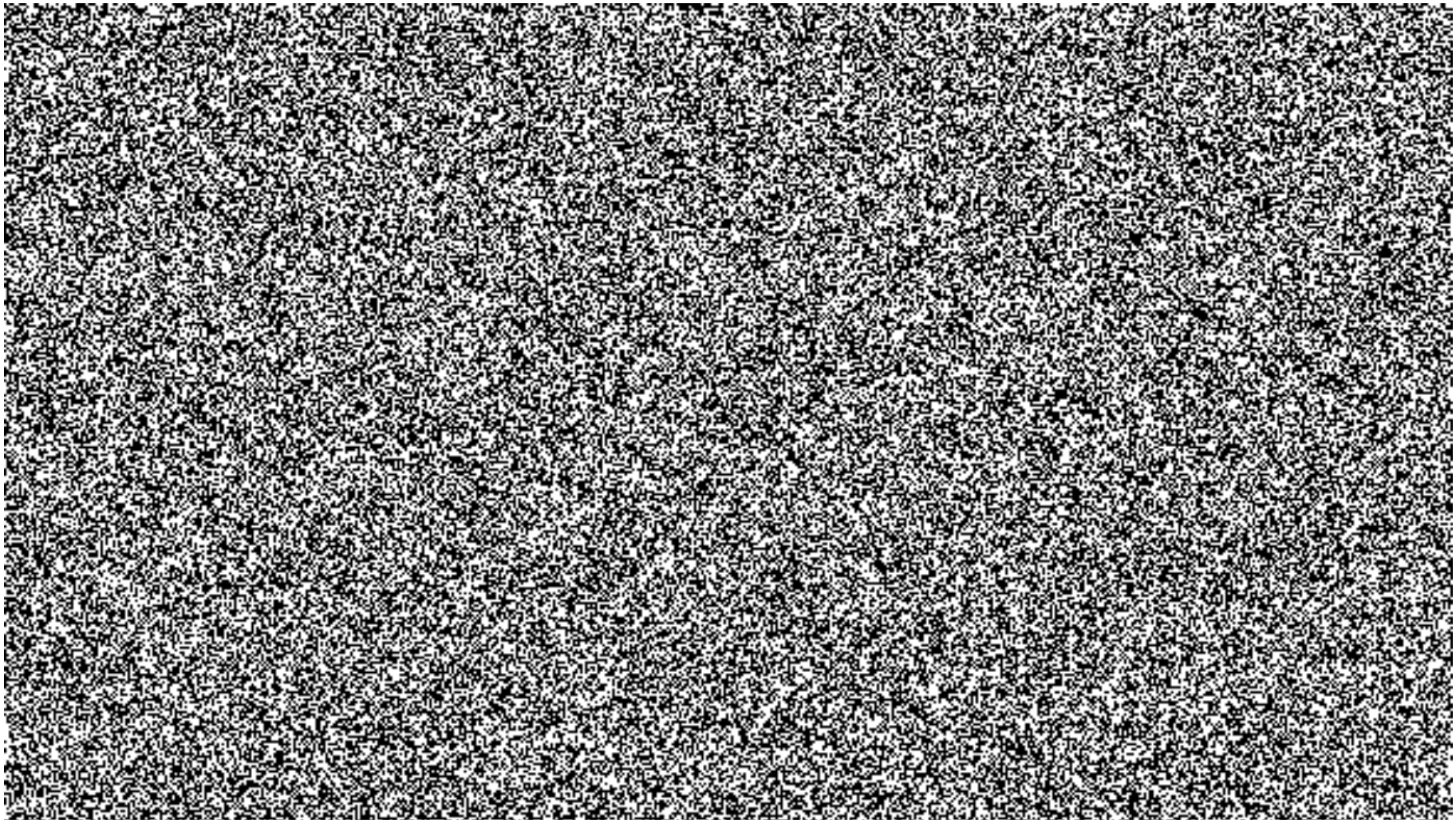
Extracted entropy FIPS 140-2 tests: **Pass**

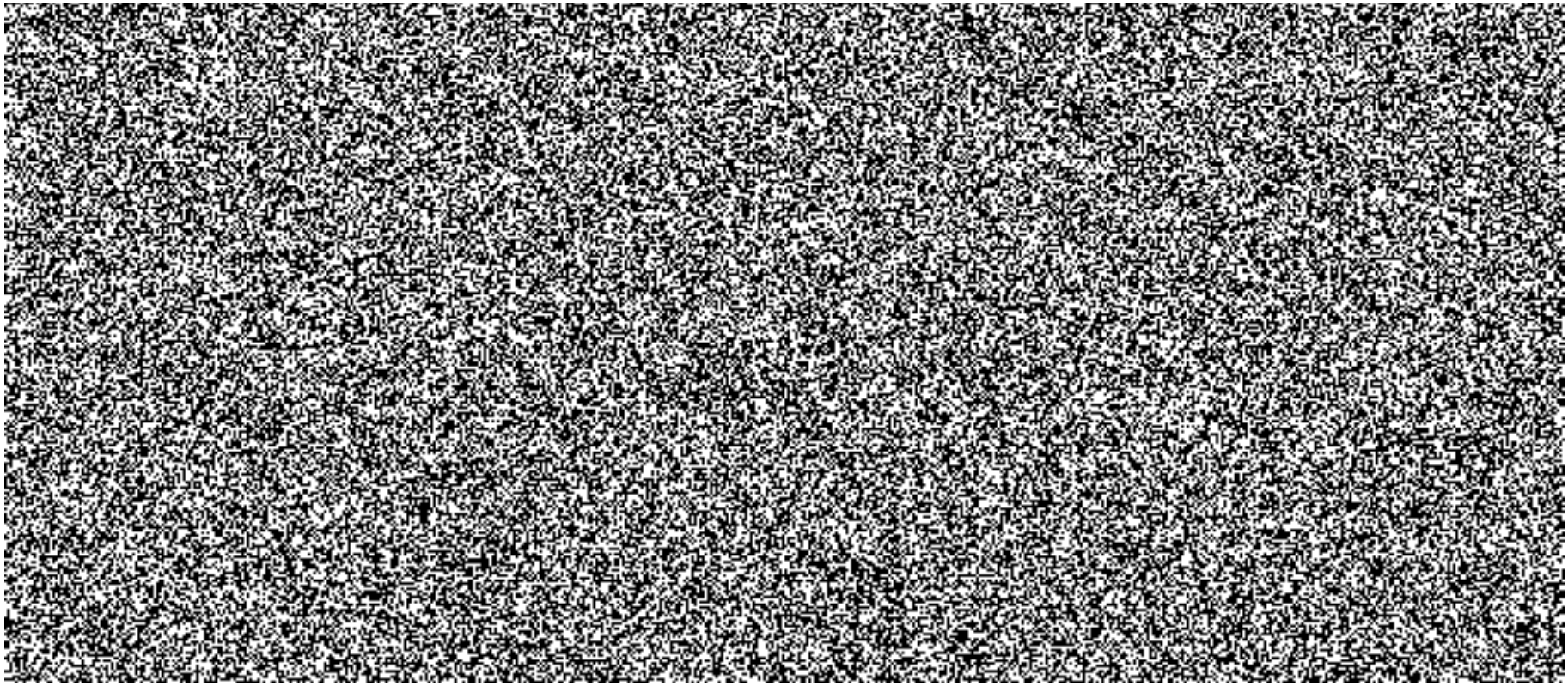
Photo of clouds

This is an example of the TRNG output from processing a photo of clouds. See the full test results output here as a text file. Clouds are generally not a good source photograph as there are a lot of repeating colours. These do get stripped out with the algorithm, but in the end you are left with less random data. The extracted output will still pass the FIPS 140-2 tests, however it may be wiser to use another source image.

Original photo

Extracted entropy





Extracted entropy FIPS 140-2 tests: **Pass**

Summary

In summary, the best results are achieved using the macro mode on a digital camera and taking close-up shots of things like sand or grass. Shooting a variety of things like rocks, trees or water can also work. Users can experiment taking photos of various things in nature on their own. Users should take care to verify the extracted random data passes the FIPS 140-2 tests. This will give full confidence for using it as one-time pads. With this TRNG and a single 12 MP photo, it can produce enough random data for approximately 3900 messages (depending on the source photograph quality).

The safest method is to use a photo from a standalone digital camera that has been captured in the camera's raw image

format, transferred to a computer by USB cable (or SD card), then losslessly converted to PNG (or BMP) file format. Avoid using a mobile phone as they generally have poor security and the contents can be secretly collected from them at any time due to the closed baseband processor. The user should erase the original photo after the one-time pads have been created from it. To be extra safe, users should run the TRNG program from a non-networked (air gapped) computer.

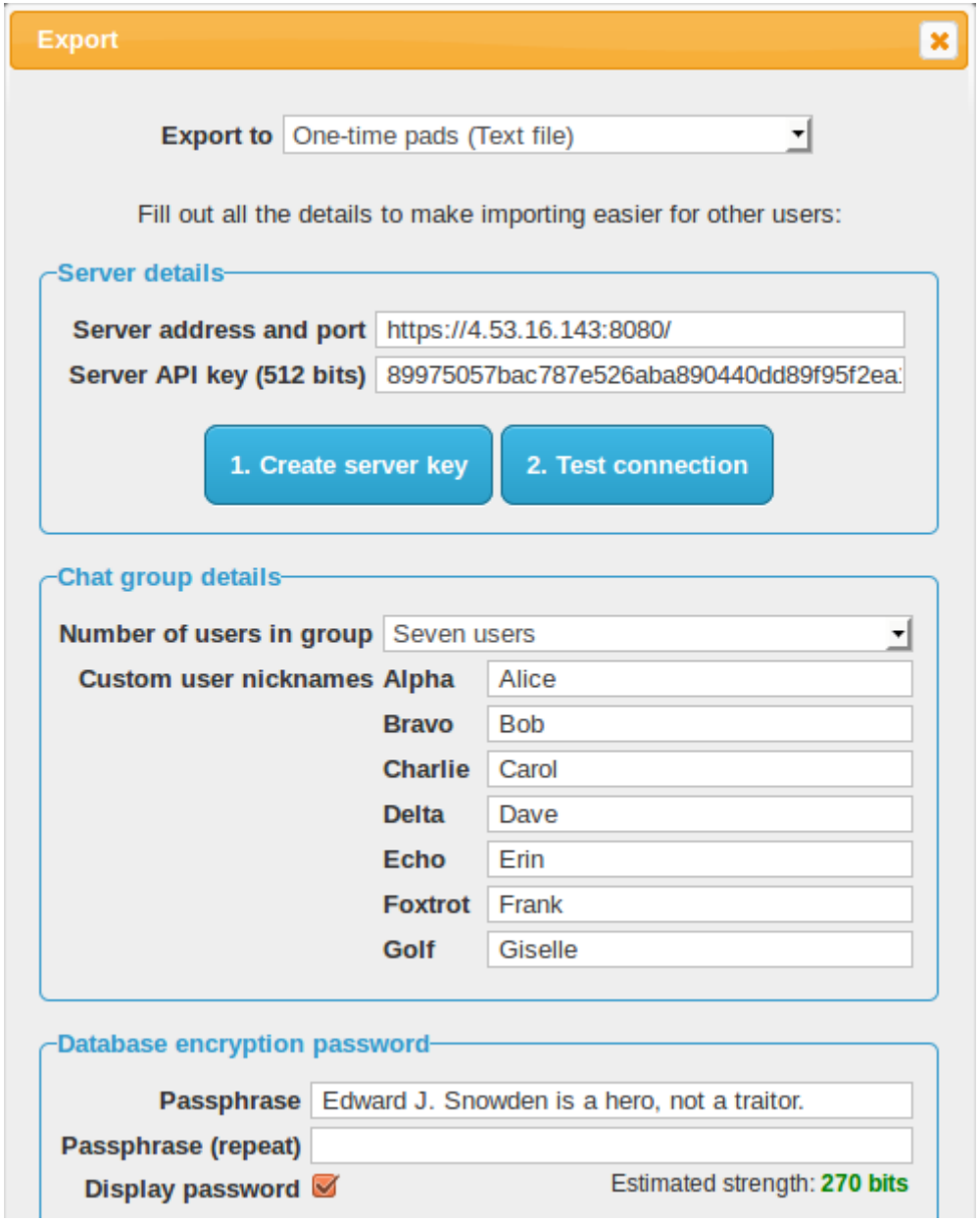
Custom random data loading

In the version 1.4 release onwards, the software allows users to import random data directly from their own trusted sources e.g. a hardware RNG or physical entropy source to use with the program. This will allow for creating a lot more one-time pads in a shorter period of time.

Users can change the upload type of the file from plain binary, to a plain text file containing hexadecimal symbols or Base64 characters. Once uploaded, the FIPS 140-2 tests will be run on the random data to prove that the random data is good quality. If the tests pass the user can export the random data and the program will split up the random data into separate one-time pads for use with the program.

Pad storage and exporting data

After the one-time pads have been created, they must be exported separately for each user. Part of this process automatically determines who will be sending with what one-time pads. The one-time pads are divided up equally amongst the group members and allocated to each user for sending. This prevents one user from accidentally sending with another user's allocated one-time pads, causing a two-time pad situation and allowing for cryptanalysis. All users get read access to the other group member's one-time pads, so when a message is received from another user they can decrypt it.


Export
Export to
One-time pads (Text file)
Fill out all the details to make importing easier for other users:
Server details
Server address and port
https://4.53.16.143:8080/
Server API key (512 bits)
89975057bac787e526aba890440dd89f95f2ea
1. Create server key
2. Test connection
Chat group details
Number of users in group
Seven users
Custom user nicknames
Alpha
Alice
Bravo
Bob
Charlie
Carol
Delta
Dave
Echo
Erin
Foxtrot
Frank
Golf
Giselle
Database encryption password
Passphrase
Edward J. Snowden is a hero, not a traitor.
Passphrase (repeat)
Display password
Estimated strength: 270 bits

- This screen contains all the details that will be saved to each export file for each chat group user. The person creating the chat group should input the server address and API key here so that when other users import the file they do not need to do any additional configuration. All they need to do then is load the one-time pads into their browser and start chatting.

- A server API key can be generated from this screen. The program effectively takes 512 bits from the start of the extracted random data and uses it as the key. This 512 bits is now no longer available for use as a one-time pad. This API key is manually loaded up into the server's config file by the user. After this the user can test the connection to the server to make sure everything is set up correctly.

- The user can also define how many users will be in the chat group. They can customise the user's name or nickname next to the call sign. The custom nickname is kept locally and stored with the one-time pads. This way each chat group user has the same set of names and knows who is in the conversation. The custom nickname is not sent over the network, only the call sign (alpha, bravo etc) is sent as part of each network request so the server can send/receive messages for that user. This means only the real users know who is talking to who. If there are multiple people and chat groups around the world using the same protocol and different servers, then it makes traffic analysis even more difficult.

- There are 3 export options for using the one-time pads with the program. **Export to clipboard** lets you copy/paste the pads from memory into wherever you want to put them. **Export to text file** pops up a save dialog to save the pads to a text file on your filesystem or portable storage media. Exporting to the **browser's local database** is if you want to export the created pads for that user into the local storage of that device/computer browser directly and start chatting immediately.

- Saving to removable media such as a CD, DVD, MicroSD, SD card or USB drive will be convenient and portable. Flash memory is at least small and compact which means you can conceal, destroy or get rid of it quickly and easily. The most secure option may be to use CDs/DVDs which can be written once, transferred to each user, then destroyed. The most convenient option is to get a USB thumbdrive, with a portable version of Firefox loaded on it. Save the one-time pads in there as well. Load up Firefox and then load the one-time pads from inside it. Now the program is portable and you can take it with you on your keyring, run it from any trusted computer (e.g. home or work) and stay in contact wherever you are.

- Hard drives and flash memory are notoriously difficult to remove data from. Version 1.5 of the program now allows for the one-time pad database to be encrypted and authenticated before transport. This is detailed in full in the next section.

- The password/passphrase used should have at least 256 bits of entropy for transport which is about 41 characters. In

version 2.0 the database will be encrypted all the time so users may opt for a shorter password for faster access on their mobile devices. The password strength estimator calculates a rough estimate of the password strength in bits as the user is typing. Password characters are assumed to be drawn uniformly randomly among the most commonly used characters on a standard US keyboard. This is calculated as uppercase A-Z (26 characters) plus lowercase a-z (26 characters) plus numbers 0-9 (10 characters) for a total of 62 characters. This will produce a more conservative entropy estimate than if special characters were included as well (i.e. the full 95 ASCII printable characters). The formula will also take into account the PBKDF iterations which roughly increases the security in bits by $log_2(\ Iterations\ )$ e.g. $log_2(\ 10{,}000\ )$ which is approximately 13~ bits. The full formula for calculating the entropy of the password in bits is as follows:

$$\textbf{\textit{Entropy Bits}} = (\ Number\ of\ Password\ Characters \times log_2(\ 62\ ))\ +\ log_2(\ Number\ of\ PBKDF\ Iterations\ ).$$

One-time pad database encryption and authentication

Since version 1.5, the program can now encrypt and authenticate the one-time pad database prior to export and transportation to other chat group users. The program uses a strong cascade stream cipher encryption and a cascade of two modern MAC algorithms for authentication. This provides additional assurances such as:

- The database has not been tampered with in transit. An attacker cannot replace the one-time pads with ones that they already know which would allow covert surveillance, set all the one-time pads to zero bits which would nullify the encryption, subtly duplicate the one-time pads in the database which would allow for two-time pad cryptanalysis, or swap pads between users which would cause lost/indecipherable messages for other users in the group.

- The database is not easily readable if stolen or seized in transit. A computationally unbounded adversary could in theory break the cascade encryption after many decades of brute force attack with a quantum computer, but this is very unlikely. If a user knew that their one-time pad database was stolen or seized (e.g. at an international airport) then they would notify the other chat group users to stop using that set of one-time pads immediately and switch to a different set. Then only the small number of messages which had been sent since one-time pad generation until that point in time would be compromised. Because only a small number of messages would have been sent during this time and the database is very difficult to crack, then this reduces the likelyhood an attacker would even try mounting a brute force attack.

- After successful transportation of the one-time pads without interception or tampering, the database can be quickly deleted

and the rewriteable transport media (e.g. MicroSD card, SD card, USB drive) can be re-purposed for something else which would overwrite the database eventually anyway. Because an attacker does not even have the encrypted one-time pad database, it is not absolutely necessary to secure erase the media or destroy it (e.g. write once CD, DVD media) unless absolute security is required.

In version 2.0 of the program, which is currently in development, the same encryption will be used to secure the database as it resides on the client devices. Pads will be decrypted as they are needed, used to encrypt a message, then deleted from the database. The program does not currently provide any steganography for transporting the database, so if this is needed in the near future it is advised to use a TrueCrypt 7.1a hidden volume and store the one-time pad database inside it. If an attacker forces you to reveal the password you can reveal the outer volume password which would reveal decoy files, and you would still have plausible deniability that a hidden volume containing the one-time pads does not exist.

Cascade database encryption

To encrypt each one-time pad in the database a cascade of two strong, reputable stream ciphers is used. The ciphers are the Advanced Encryption Standard (AES) in Counter Mode (AES-CTR) and Salsa20 with the full 20 rounds. AES, which is based on the Rijndael algorithm by Vincent Rijmen and Joan Daemen, won the Advanced Encryption Standard competition. A reduced 12 round variant of Salsa20 (Salsa20/12) by Daniel J. Bernstein was selected for the eSTREAM software portfolio. The full 20 round variant of Salsa20 was chosen for added security. Two random 256 bit keys are generated using the TRNG at export time. One key is used for AES-CTR and the other for Salsa20. A different nonce is used for each one-time pad to be encrypted in the database. The design Exclusive ORs (XORs) the AES-CTR keystream and the Salsa20 keystream together then XORs the combined keystream with the plaintext one-time pad.

The reason a cascade of two stream ciphers is used is because there may be secret cryptanalytic techniques against a cipher such as AES when it's used to encrypt something on its own. A recent publication of the Snowden documents revealed that NSA have their own in-house (non public) techniques against AES and other ciphers. While a trivial reversal of a strong cipher without any additional information is highly unlikely and would be indicative of the algorithm being very weak, it may be more likely that a single encryption algorithm becomes vulnerable to NSA when they have access to known or chosen plaintext encrypted by the algorithm as well. With a stream cipher cascade, the separate keystreams are XORed together. There is no way to determine which bits belong to each keystream and cipher if each cipher is suitably strong on their own. Even if there is a cryptanalytic break in AES, an attacker does not have access to the raw keystream created by the AES algorithm because

there is still plaintext and the Salsa20 keystream mixed in with it. Even if an attacker knows a lot of the plaintext they still won't be able to decrypt the AES layer of encryption because the layer underneath is a random Salsa20 keystream which they do not know. Likewise if they tried to decrypt the Salsa20 encryption layer first, the next layer is a random AES keystream so they would not even know when they have decrypted the first layer correctly. The best remaining attack against a stream cipher cascade may be a brute force of both keys which would take a very long time.

The following describes the encryption for the database:

$\boldsymbol{Key_1}$ = A 256 bit random key for AES-CTR generated by the TRNG
$\boldsymbol{Key_2}$ = A different 256 bit random key for Salsa20 generated by the TRNG
$\boldsymbol{Nonce_1}$ = A unique 96 bit nonce for AES-CTR, changing for each database row (one-time pad) to be encrypted
$\boldsymbol{Nonce_2}$ = A unique 64 bit nonce for Salsa20, changing for each database row (one-time pad) to be encrypted
$\boldsymbol{Counter_1}$ = A 32 bit counter for AES-CTR, starting at 0 for each database row and incrementing by 1 for each block being encrypted
$\boldsymbol{Counter_2}$ = A 64 bit counter for Salsa20, starting at 0 for each database row and incrementing by 1 for each block being encrypted

$\boldsymbol{Keystream_1} = AES\text{-}CTR(Key_1, (Nonce_1 \,||\, Counter_1))$
$\boldsymbol{Keystream_2} = Salsa20(Key_2, (Nonce_2 \,||\, Counter_2))$

$\boldsymbol{Row\ Cascade\ Encryption} = Keystream_1 \oplus Keystream_2 \oplus One\text{-}Time\ Pad$
$\boldsymbol{Row\ Cascade\ Decryption} = Ciphertext\ One\text{-}Time\ Pad \oplus Keystream_2 \oplus Keystream_1$

- The reason to encrypt each row individually rather than encrypting the entire database at once is for performance. Also in the upcoming version 2.0 of the program the database will always be encrypted on disk, so it is advantageous to only have the small keys in memory and decrypt each row as needed then delete the row.

- The keys remain the same for the entire database but the nonce changes for each database row to be encrypted. The keys for encryption and authentication are obtained from slicing the required number of bits off the beginning of the TRNG generated random data. This ensures the keys are not used for anything else and the one-time pads are generated from the remaining random data.

- Because AES in Counter Mode does not need a random IV, a unique nonce is used for encrypting each row. Each one-time pad has an index number in the database starting from 0 up to the number of pads in the database. One user might have index numbers from 0 - 1000 in their set of one-time pads and the next user might have index numbers from 1001 to 2000 in their set. This ensures there is a unique index number for each row in the database. This number is converted to hexadecimal and left padded with 0 bytes (**`00`** in hexadecimal) up to 96 bits in length. The block counter for AES is 32 bits in length starting at 0 (**`00000000`** in hexadecimal) for each row and increments by 1 for each subsequent block being encrypted.

- The nonce for Salsa20 is 8 bytes, so the pad index number is is converted to hexadecimal and left padded with 0 bytes (**`00`** in hexadecimal) up to 64 bits in length. The block counter for Salsa20 is also 8 bytes starting at 0 (**`0000000000000000`** in hexadecimal) for each row and increments by 1 for each subsequent block being encrypted.

- The first 56 bits of a one-time pad is the pad identifier which is public and used to lookup the correct pad in the database when another user sends a message. The pad identifier is not encrypted and removed prior to encryption so only the remaining 1480 bits of the random pad are encrypted. This removes any remaining known plaintext for an attacker if they attempt to decrypt one of the one-time pads.

- There is some pad database information which is stored in the client database as well (program version, custom user preferences, server address, server key, user callsign and list of group user nicknames). This is JSON encoded to a string and encrypted with the same database keys, but the static 96 bit nonce **`ffffffffffffffffffffffff`** in hexadecimal for AES-CTR and the static 64 bit nonce **`ffffffffffffffff`** for Salsa20 is used for encryption. Because each pad index number is converted to a nonce and in the language being used integers cannot exceed $2^{53 - 1}$ (9007199254740991), this nonce cannot be accidentally be re-used for encrypting a pad, therefore it is used it to encrypt the pad database information.

Cascade database authentication

Using the safe principles of Encrypt Then MAC, the program creates a MAC of the database row information including the encrypted one-time pad by using a cascade MAC. The chosen hash functions for this are Keccak-512 with the capacity set at 1024 (same as the finalised SHA3) and Skein-512. Each MAC digest is calculated independently by computing *Hash*(*Key* || *Data*) with independent keys for each algorithm. The resulting digests are then XORed together to hide the individual MAC digests from independent cryptanalysis in case one of the algorithms has a flaw. Keccak and Skein are newer hash functions that are not vulnerable to length extension attacks with this simple MAC construct.

The following describes the authentication for each row in the database:

***MAC Key*$_1$** = A 512 bit random key for Keccak-512 generated by the TRNG
***MAC Key*$_2$** = A different 512 bit random key for Skein-512 generated by the TRNG
***Nonce*$_1$** = A unique 96 bit nonce based on the row index number which was used by AES-CTR to encrypt the one-time pad
***Nonce*$_2$** = A unique 64 bit nonce based on the row index number which was used by Salsa20 to encrypt the one-time pad
User Callsign = The user callsign (e.g. 'alpha', 'bravo', 'charlie' etc) which this one-time pad is allocated to for sending
Pad Identifier = The first 56 bits of the one-time pad
Ciphertext One-Time Pad = The remaining 1480 bits of the one-time pad which is encrypted by AES-CTR and Salsa20

***MAC*$_1$** = Keccak-512(*MAC Key*$_1$ || *Nonce*$_1$ || *Nonce*$_2$ || *User Callsign* || *Pad Identifier* || *Ciphertext One-Time Pad*)
***MAC*$_2$** = Skein-512(*MAC Key*$_2$ || *Nonce*$_1$ || *Nonce*$_2$ || *User Callsign* || *Pad Identifier* || *Ciphertext One-Time Pad*)

Row Cascade MAC = $MAC_1 \oplus MAC_2$

- The resulting MAC tag is stored along with the other information for each row. When the database is being loaded on a client machine, the program will calculate the MAC again for each database row and verify that the hash digest matches the stored MAC tag for the row. If there is a match for each row then no tampering has occurred, otherwise a warning will be shown to the user. If the warning is shown, then the user should abandon the database of one-time pads and look to transfer a new set.

- In version 1.5 of the program, the database is verified only when loading the pads initially after they have been exported and transported. Currently after the pads are verified and decrypted, the database is saved to the client PC in an unencrypted state. Running the application and browser profile from inside a TrueCrypt volume is still recommended for this release to keep the one-time pads encrypted locally on the disk. In the future, version 2.0 of the program will have the one-time pad database be fully encrypted and authenticated at all times. Each one-time pad row will need to be verified and decrypted before sending or receiving a message. The reason for why this functionality is not available in version 1.5 is that the application needs to be converted to a Single Page Application first. Currently each web page is run separately and there is no in memory data sharing between pages. Converting to a single page application will reduce code duplication and mean the master password only needs to be entered once on startup, not once for each page opened.

- The pad database information which is stored in the client database (program version, custom user preferences, server address, server key, user callsign and list of group user nicknames) is also authenticated using the same cascade MAC. This is checked before decrypting and importing the pad database information. In version 2.0 it will be verified each program load to ensure database integrity.

Database index authentication

The program also creates a MAC of the database index for each user's set of one-time pads by combining the pad index numbers for each row and then creating a cascade MAC. This ensures that the all user's one-time pads in the database have not been added, swapped, reordered, removed or otherwise tampered with.

The following describes the database index MAC:

***MAC Key*$_1$** = A 512 bit random key for Keccak-512 generated by the TRNG
***MAC Key*$_2$** = A different 512 bit random key for Skein-512 generated by the TRNG
User Callsign = The user callsign (e.g. 'alpha', 'bravo', 'charlie' etc) of the set of pads being authenticated
Pad Index Number = The index number of the row in the database for the user's set of pads

$\boldsymbol{MAC_1}$ = Keccak-512($MAC\ Key_1$ || $User\ Callsign$ || $Pad\ Index\ Number_0$ || $Pad\ Index\ Number_1$ || $Pad\ Index\ Number_2$ || ...)
$\boldsymbol{MAC_2}$ = Skein-512($MAC\ Key_2$ || $User\ Callsign$ || $Pad\ Index\ Number_0$ || $Pad\ Index\ Number_1$ || $Pad\ Index\ Number_2$ || ...)

Index Cascade MAC = $MAC_1 \oplus MAC_2$

- When a message is received or sent that pad is deleted from the database so the pad index MAC needs to be updated every time a new message is received or sent. This functionality will be added in version 2.0 when the database will be encrypted and authenticated at all times. At the moment in version 1.5, the full verification of the index is only performed as part of the initial importing of the one-time pads after transportation. This is mainly to verify that the transfer took place without tampering. Version 2.0 will be much more comprehensive and ensure that the database integrity and authenticity is valid at all times.

- Including the user callsign in the MAC means that an attacker cannot swap out one-time pads from one user into another user's set of pads, forcing a two-time pad situation and allowing cryptanalysis.

Protection of database encryption and authentication keys

The actual database encryption and authentication keys which were generated by the TRNG are stored in inside the database with the rest of the other information. To protect these keys while they reside in unprotected storage a simple key wrapping construction is used.

Database master key derivation

To encrypt the actual database encryption and authentication keys, a master key is created by deriving it from a password, two salts and two separate Database Password Based Key Derivation Functions (PBKDFs). The following describes the cascade PBKDF construction:

Password = A strong password/passphrase entered by the user
***Salt*$_1$** = A 768 bit random salt generated by the TRNG to be used with the Keccak PBKDF
***Salt*$_2$** = A different 768 bit random salt generated by the TRNG to be used with the Skein PBKDF
Keccak Iterations = The number of iterations to be performed by the Keccak PBKDF with the default set at 10,000
Skein Iterations = The number of iterations to be performed by the Skein PBKDF with the default set at 10,000

Intermediate Key = PBKDF2-Keccak-512(*Password*, *Salt*$_1$, *Keccak Iterations*)
Master Key = PBKDF-Skein-512(*Intermediate Key*, *Salt*$_2$, *Skein Iterations*)

- The first PBKDF is PBKDF2 with the Keccak-512 hash function. The output key size is set at 512 bits, the default number of iterations is set at 10,000 and a 768 bit random salt which was generated by the TRNG is used.

- Keccak can be used in conjunction with HMAC which is what PBKDF2 uses internally. The reason to use Keccak instead of SHA2 is because it is a newer hash function and it is not designed by the NSA. Anything designed by the NSA is avoided by this program in case it has deliberate secret weaknesses that are unknown to the academic community. An algorithm designer is in the best position to design an algorithm with a subtle weakness. In the case of SHA2 a weakness may not be discovered for many years if academia is well behind the state of the art in cryptography. We know at one point the NSA were 20 years ahead in cryptography when they knew about differential cryptanalysis before anyone else. They are likely still ahead by a large margin. Using Keccak which was the winner in an open competition and which has a design which is completely open is considered much safer. In comparison to SHA2, Keccak should provide better entropy in the derived key,

however it may be a lot faster in hardware which would provide a slightly better advantage for an attacker performing a brute force attack. However the main attacker to be considered is the NSA and we can reasonably assume they have dedicated ASICs and other hardware for cracking password hashes which were derived using PBKDF2-SHA1 or PBKDF2-SHA2 simply because everyone in the world is using those algorithms. If everyone uses the same algorithms recommended by the US government which is also running worldwide mass surveillance programs then this makes the NSA's job much easier. They can focus their efforts on attacking only a few algorithms at a much cheaper cost. By using the newer Keccak algorithm this forces the NSA to expend more money building new dedicated ASICs or re-engineer their supercomputer code just to crack an encrypted database created by this program.

- The second PBKDF uses the Skein hash function. The output key size is set at 512 bits, the default number of iterations/repetitions is set at 10,000 and a separate 768 bit random salt which was generated by the TRNG is used. The resulting 512 bit *Intermediate Key* from the previous PBKDF2-Keccak function is passed in as the *Password* for this function.

- The Skein PBKDF method is described in section 4.8 of The Skein Hash Function Family specification document (v1.3) - Skein as a Password-Based Key Derivation Function (PBKDF). Quoting the document:
"The application stores a random seed S, asks the user for a password P, and then performs a long computation to combine S and P." ... "An even simpler PBKDF is to simply create a very long repetition of S and P; e.g., S || P || S || P || S ..., and hash that using Skein. (Any other optional data can also be included in the repetition.) This approach is not ideal with a normal hash function, as the computation could fall into a loop. But in Skein, every block has a different tweak and is thus processed differently."

- For added security an option exists in the user interface to use custom iteration counts. The user can decrease the number of iterations for slower portable devices and use a longer password to compensate. The user may also choose to increase the iteration counts to make the database more resilient to attack. The default of 10,000 iterations for each of the PBKDFs is a good balance between strength and slow speed of the JavaScript runtime engine.

- The user can also choose not store the number of Keccak or Skein iterations with the rest of the database. The user would remember the iterations or write them down separately on a piece of paper. This may be useful if passing through airport security and there is a high likelihood of the data being confiscated and copied. This forces an attacker with only the database to try every iteration count for every password permutation. To counter an attacker simply caching the results of previous iteration counts and running the PBKDF on one password at a time, the number of iterations are appended to the end of the salt at runtime. This forces the attacker to do the full PBKDF iterations for every reasonable iteration count the

user could have chosen e.g. 1 - 10,000+ then repeat that for all possible password permutations.

- The total length of the two salts is the same length of all the database keys (256 bits AES-CTR + 256 bits Salsa20 + 512 bits Keccak + 512 bits Skein) which adds up to 1536 bits. Each 768 bit salt is fed into a different PBKDF. Normally passwords do not contain much entropy, so the salts which are randomly generated by the TRNG are used as a backup to add additional entropy to safely secure the database keys.

- Another option exists in the user interface to store the two 768 bit salts (1536 bits concatenated together) as a separate keyfile. The advantage of this is to store the keyfile on a separate storage device (e.g. MicroSD card) or written down on a piece of paper which can be easily hidden if the database is at risk of being compromised in transit. If an attacker can confiscate or steal the primary device (e.g. notebook PC) which has the encrypted database but can't find the keyfile as well then it is practically impossible to crack the database encryption in any reasonable timeframe. Future versions may allow this keyfile to be hidden inside an image file using steganography.

- In the future the program may support Argon2 which was the winner in the Password Hashing Competition. At the time of writing, this standard is not currently finalised and there is no library support for it currently.

Sub key derivation

Four sub keys are derived from the master key and are used to encrypt the actual database encryption and authentication keys. This is basically a simple KDF2 construction but uses a cascade of two hash functions for each counter value to protect against flaws in either algorithm. The newer hash functions used are already secure against length extension attacks and do not need HMAC. The following describes the cascade sub key derivation:

Master Key = The 512 bit master key derived from the cascade PBKDF used earlier
$\boldsymbol{Counter_i}$ = A 32 bit numeric counter to be combined with the hash function e.g. (**`00000001`**, **`00000002`**, ... in hexadecimal)

$\boldsymbol{Derived\ Key_1}$ = Keccak-512(*Master Key* || $Counter_1$) ⊕ Skein-512(*Master Key* || $Counter_1$)
$\boldsymbol{Derived\ Key_2}$ = Keccak-512(*Master Key* || $Counter_2$) ⊕ Skein-512(*Master Key* || $Counter_2$)
$\boldsymbol{Derived\ Key_3}$ = Keccak-512(*Master Key* || $Counter_3$) ⊕ Skein-512(*Master Key* || $Counter_3$)
$\boldsymbol{Derived\ key_4}$ = Keccak-512(*Master Key* || $Counter_4$) ⊕ Skein-512(*Master Key* || $Counter_4$)

- A simple KDF2 construct to get encryption and authentication keys from a master key would apply the hash function twice. Once with the *Master Key* and a unique *Counter* (e.g. 01) to make an encryption key, and again with the *Master Key* and another unique *Counter* (e.g. 02) to gain a unique MAC key. If the keys used for encryption is compromised it is computationally hard to find a pre-image for the one-way hash function to determine the *Master Key* or the derived MAC key. Similarly if the MAC key is compromised it is hard to reverse the process to find the master key or encryption key. This cascade construct performs two hashes for each derived key, once using Keccak on the *Master Key* and unique *Counter*, then again with the Skein algorithm and another unique *Counter*. Finally it XORs the resulting random hash digests together to produce the derived key. This adds additional assurances that the derived key will not be easily reversed if there is a flaw in either algorithm discovered in the future.

- The resulting derived keys are 512 bits in length. The first and second derived keys are used for AES-CTR and Salsa20. Because the key lengths for these encryption algorithms are only 256 bits in length, these two derived keys are truncated to just the first 256 bits. The third and fourth derived keys which are used for Keccak and Skein remain at 512 bits.

Encryption and authentication of database keys

The following describes the encryption and authentication of the actual database keys using the derived keys from earlier:

Database Keys = The actual AES-CTR, Salsa20, Keccak and Skein database keys, concatenated together
$\boldsymbol{Nonce_1}$ = A static 96 bit nonce for AES-CTR (`000000000000000000000000` in hexadecimal)
$\boldsymbol{Nonce_2}$ = A static 64 bit nonce for Salsa20 (`0000000000000000` in hexadecimal)
$\boldsymbol{Counter_1}$ = A 32 bit counter for AES-CTR, starting at 0 and incrementing by 1 for each block being encrypted
$\boldsymbol{Counter_2}$ = A 64 bit counter for Salsa20, starting at 0 and incrementing by 1 for each block being encrypted

$\boldsymbol{Keystream_1} = \mathit{AES\text{-}CTR}(\mathit{Derived\ Key}_1, (\mathit{Nonce}_1 \,||\, \mathit{Counter}_1))$
$\boldsymbol{Keystream_2} = \mathit{Salsa20}(\mathit{Derived\ Key}_2, (\mathit{Nonce}_2 \,||\, \mathit{Counter}_2))$
Encrypted Database Keys = $\mathit{Keystream}_1 \oplus \mathit{Keystream}_2 \oplus \mathit{Database\ Keys}$

$\boldsymbol{MAC_1}$ = Keccak-512($\mathit{Derived\ Key}_3$ *Encrypted Database Keys*)
$\boldsymbol{MAC_2}$ = Skein-512($\mathit{Derived\ Key}_4$ *Encrypted Database Keys*)
Cascade MAC = $\mathit{MAC}_1 \oplus \mathit{MAC}_2$

- After encryption and authentication, the encrypted database keys and the MAC are stored in the database. To decrypt the database, a user will enter their password, load the keyfile and set the number of iterations. It will perform the same steps above to generate the derived Keccak and Skein keys, recreate the MAC against the stored encrypted database keys, then match that against the stored MAC. Any incorrect match may mean an incorrect password, keyfile or number of iterations used. Alternatively a more serious issue could be that someone has tampered with the database.

Using HTML5

Matasano Security raises some points about JavaScript cryptography and how JavaScript being delivered by the web server is insecure. There are definite sensible and secure solutions to the problems they raised. Also if you read the FAQ, all of their other concerns have been mitigated one by one.

Most points they made are not applicable for this software due to the fact that the source code is downloaded as a .tar.gz archive file and users are expected to verify the file's GPG signature with the one from this website to ensure its authenticity. From there the code can be run *locally* from the machine by going to the directory and running index.html which will load the website and code into the browser. This means all code is always running locally from the local hard drive and the web address will be similar to **`file:///media/truecrypt1/jericho/client/index.html`**. All the executable code is self contained and does not rely on any server delivered JavaScript at all. This itself mitigates the majority of the problems the Matasano Security article raised.

HTML5 has more advantages than disadvantages. It's easier and faster to develop with. New APIs allow for cryptography, persistent database storage, messaging and file management. Browsers are first in line for security updates and the best ones are open source and trustworthy. People rely on browsers to have good security to do their internet banking and shopping online. The source code does not need to be compiled. You can verify the source code being run live in the browser using Chromium or Firefox built-in Web Development tools or with addons like Firebug. This allows you to verify the code is doing exactly what it should be. HTML5 is cross platform, one code base can literally run on Windows, Linux, Mac, phones and tablets simply with the latest web browser. One of the goals of the project is to get a truly secure chat program functioning on an open hardware platform and an open operating system like Firefox OS. This is a true open source OS for smartphones and tablets from a reputable organisation that believes that individual's security and privacy on the internet is fundamental.

Currently this software has been tested to work in Firefox, Chromium and Google Chrome. Some of the other browsers have

not implemented the Web Crypto API yet. From version 1.4 onwards, the layout will be responsive and will work on desktops, tablets and mobiles. Some more work and testing is still required to get it working nicely on mobile. In particular a method to load the one-time pads into the program using later versions of Android. Using Firefox is recommended as they are open source and are not involved in the spy scandal with PRISM. Unfortunately nobody can say for sure about that with Google. Chromium may be another alternative but the proprietary Google Chrome is not recommended.

It is recommended to run the browser profile from inside a TrueCrypt volume to protect the one-time pads when they are stored inside the browser storage. Future versions of the software will encrypt the one-time pads within the program before storing locally.

The program is built using sound, provably strong cryptography. The design and implementation to bring the solution together uses secure cryptographic primitives (Skein and Keccak) for authentication which were designed by publically reputable and trustworthy cryptographers. We have avoided using algorithms from the NIST who have been shown to be heavily influenced by the NSA. The NSA have been secretly weakening and promoting weak or broken public encryption standards.

The main crypto library this program uses is CryptoJS. There were some other libraries that would be good to use but they either did not work in a HTML5 Web Worker which makes the hashing process take significantly longer and blocks the user interface, or their outputs couldn't be verified from the reference test vectors. Some more SHA-3 finalists would be nice to include from the recent competition however there is a lack of hash algorithms implemented in JavaScript. It is important to verify that the code can produce the same hash outputs as the test vectors from the specification documents. This ensures the program is not using a faulty or backdoored version of the algorithm that gives incorrect results.

REST API using JSON

- The client browsers communicate to and from the server using JSON and the REST API on the server. The server side code is currently written for the latest version of PHP 5 using best security practices for protection against SQL injection and other attacks. This will likely be rewritten in a different language in the next release e.g. NodeJS to have a common cryptographic library base across the client and server.

- The server sends CORS HTTP headers so that the client browsers can connect and make a cross-origin request. This is needed because the program is run locally from the **`file:///`** location and not served by a webserver. If the JavaScript code

was served from a web server it could let an attacker perform a MITM attack and alter the code, this could then make the user send unencrypted messages and they would not be any wiser.

- For now the browsers use an AJAX request to send/receive data to/from the server. The reason for this is that the chat can be delayed like email (encrypted messages remain on the server until they are received), or realtime if users are connected at the same time. If a user is connected it will check for new messages from the server every 3 seconds which is fast enough to appear as realtime and does not stress the server too much.

A summary of the request/response protocols are shown below:

Send message

This is the request the client makes when sending a message.

Property name	Type	Description	Example
data	string	A JSON encoded string containing the data to be processed by the server.	"{"user":"alpha","apiAction"...}
- user	string	Who the message is sent from. This is the pseudonymous call sign for the user e.g. 'alpha', 'bravo', 'charlie' etc.	"alpha"
- apiAction	string	The requested API action to perform on the server.	"sendMessage"
- msg	string	The encrypted message and its encrypted MAC concatenated together.	"5f9b231..."
- nonce	string	The 512 bit random nonce for this request.	"ab826f9..."
- timestamp	number	The UNIX timestamp of when this message was sent.	1405848703
mac	string	The 512 bit MAC of the JSON encoded data being sent to the server for the message.	"89175ad..."

Send message server response

This is the response the server makes after it has received a request to send a message.

Property name	Type	Description	Example
data	string	A JSON encoded string containing the data sent by the server.	"{"success":true,"statusMess…}
- success	boolean	Boolean flag to determine if the message was sent successfully or not.	true
- statusMessage	string	Message status to be displayed on the client to inform the user if the message was sent successfully or not.	"Sent"
mac	string	The 512 bit MAC of the JSON response data being sent from the server.	"ac1841d5..."

Receive messages request

This is the request each client makes every 3 seconds. This will get any new messages from the server for the user making the request.

Property name	Type	Description	Example
data	string	A JSON encoded string containing the data to be processed by the server.	"{"user":"alpha","apiAction"…}
- user	string	Who the request is sent from. This is the pseudonymous call sign for the user e.g. 'alpha', 'bravo', 'charlie' etc.	"alpha"
- apiAction	string	The requested API action to perform on the server.	"receiveMessages"
- nonce	string	The 512 bit random nonce for this request.	"df3e673312..."
- timestamp	number	The UNIX timestamp of when this request was sent.	1405844703
mac	string	The 512 bit MAC of the JSON encoded data being sent to the server.	"89175ad..."

Receive messages response

This is the response the server will make after receiving a request from a client for retrieving new messages.

Property name	Type	Description	Example
data	string	A JSON encoded string containing the data sent by the server.	"{"success":true,"statusMess…}
- success	boolean	Boolean flag to determine if messages were received successfully or not.	true
- statusMessage	string	Message status to be displayed on the client to inform the user if the messages were received successfully or not.	"Messages received successfully."
- messages	array	On success, will contain an array of objects. Each object contains a message sent by one of the other users.	"[{"from":"alpha","msg":"5f9..."}, ...]
- from	string	Which user in the chat group sent the message. This is the pseudonymous call sign for the user e.g. 'alpha', 'bravo', 'charlie' etc.	"alpha"
- msg	string	The encrypted message and its encrypted MAC.	"5f9b231..."
mac	string	The 512 bit MAC of the JSON response data being sent from the server.	"ac1841d5..."

Initiate auto nuke request

This is the request a client makes if they want to wipe the all the other chat group user's local one-time pad databases. It is very disruptive and designed for use only in an emergency. This command is sent encrypted and authenticated via the secure message channel. This is so only a valid user with the set of one-time pads can trigger an auto nuke. An attacker with access to the server cannot change anything to trigger the chat group users to delete their local databases.

To trigger the request, the user will visit a separate screen in the user interface and click a button. The client will send a regular message with the string code "**`init auto nuke`**". This will be encrypted with the one-time pad, authenticated with the MAC and sent to the server like a regular message. The initiating user's current local database of one-time pads will then be wiped and their screen cleared.

Auto nuke initiated response

If a chat group user is checking for new messages and an authentic message comes through with the code "**`init auto nuke`**", instead of receiving and processing new messages, the user's client will show a notification that the auto nuke has been initiated by the user who initiated it, then the program will immediately wipe their local database of any data and clear the screen of any sent or received messages.

Message encoding

Each one-time pad is made up of 192 bytes which is 384 hexadecimal symbols or 1536 binary digits. See the table below for more information:

	Full one-time pad
Bytes	192
Hexadecimal symbols	384
Binary digits	1536

Each message is made up of three main parts, the pad identifier, the message parts and the message authentication code. The one-time pad is used to encrypt the message parts and the MAC tag.

	Pad identifier	Total message parts	MAC tag
Bytes	7	121	64
Hexadecimal symbols	14	242	128
Binary digits	56	968	512

This can be further broken down to the individual message parts. See below for the length of each part:

	Pad identifier	Message	Actual message length	Message sent timestamp	MAC tag
Bytes	7	115	1	5	64
Hexadecimal symbols	14	230	2	10	128
Binary digits	56	920	8	40	512

- At the moment the program only supports messages typed using the ASCII printable character set. This is roughly all the characters you can see on a standard US keyboard. In practicality this limits the languages that can be used to English and ones that use the basic Latin alphabet at the moment. This is for ease of development at the moment and the program will support other languages with UTF-8 in the future.

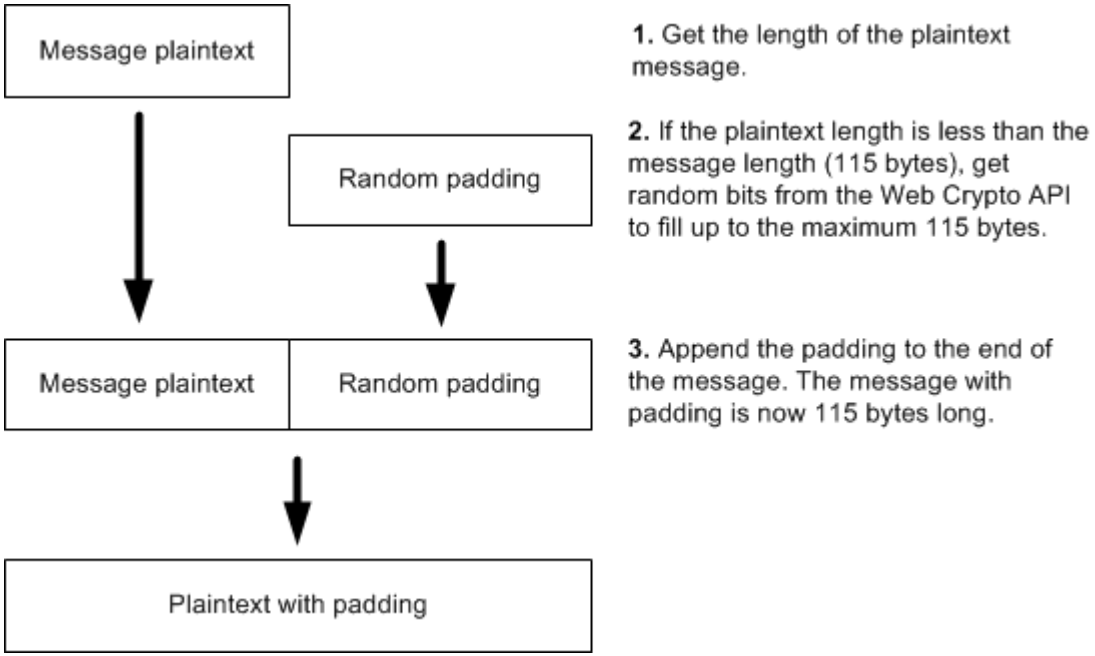
- All messages that are sent/received are encoded to hexadecimal format first which is just stored in a normal JavaScript string. This is to make transport using JSON easier rather than trying to send UTF-8 encoded data over the wire. A hexadecimal symbol (Nibble) is 4 bits, and two hexadecimal symbols makes 8 bits (one Octet/Byte) and can be represented as a single ASCII character.

- The **Pad identifier** is the first 7 bytes (56 bits) from the one-time pad. This identifies to the receiver which pad should be used to decrypt the message. This is used rather than sending the sequence number of the one-time pad to remain in sync with the other users as that could reveal to an attacker the number of messages sent so far.

- The maximum message length has been set at 115 bytes (115 ASCII characters) for now. This is slightly less than the size of a tweet (140 characters) or an SMS (160 characters). This is because generating enough random data takes a long time. Making a message length longer than that, where the one-time pad may or may not be fully used is wasteful. If users need more than 115 characters they can simply send a second message. In future a new feature will be added to allow a message to be spread over multiple one-time pads.

  If a message is less than 115 bytes in length it is padded to the right (up to the maximum 115 bytes) with random bits generated from the Web Crypto API. This hides the actual length of the message to frustrate any cryptanalysis. For example if no padding was added and the message was simply "hi" then the ciphertext would be the same length which could aid the attacker. Of course there are a few other words with only two letters which allows for some uncertainty. However if the message is padded up to the full 115 bytes each time, then an attacker knows nothing about the true length of the message,

only that it is somewhere between 1 and 115 bytes long.



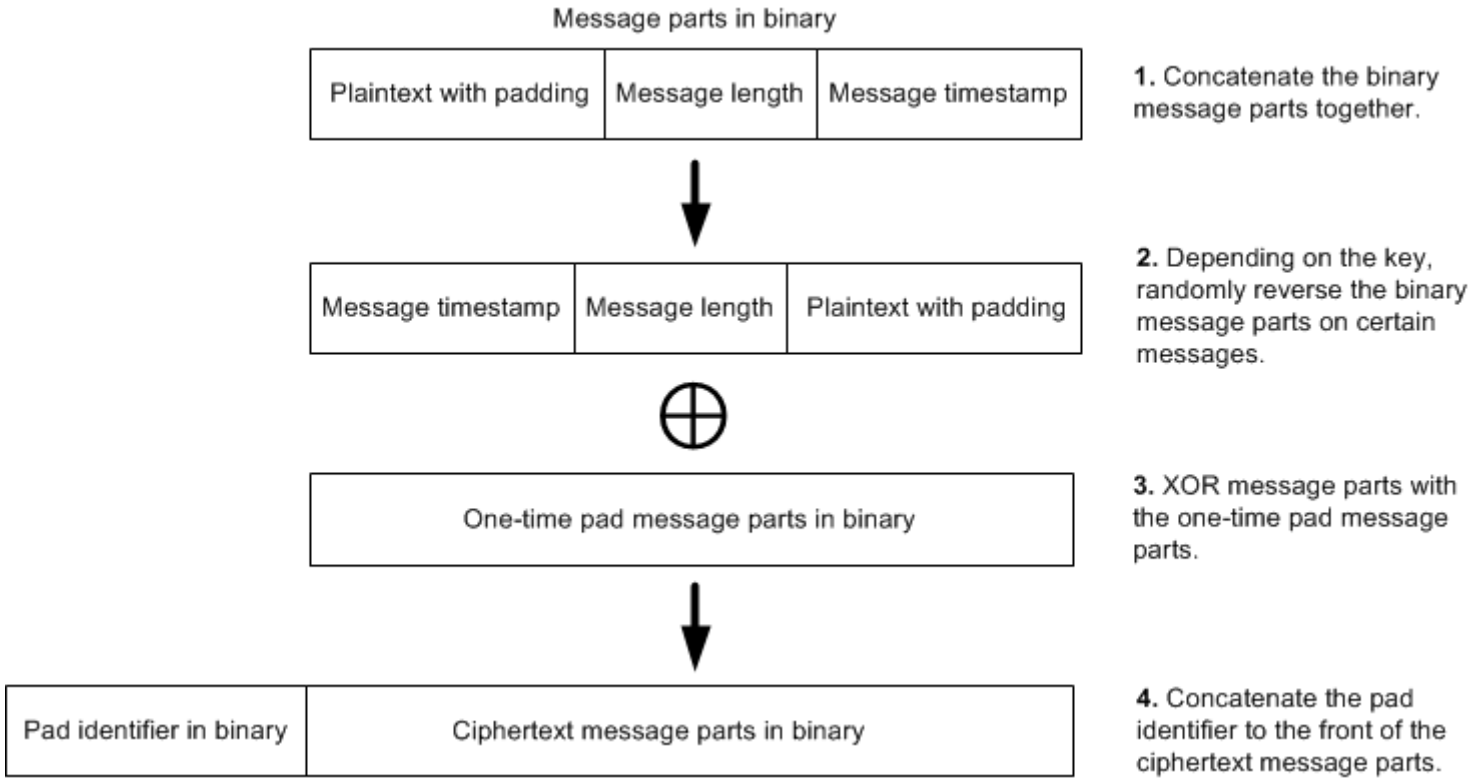
- The **Actual message length** is the true message length without any padding. It is an aid in the decryption process so it can remove the correct number of padding bytes from the end of the message and reveal the original plaintext automatically. This field is always 1 byte in length which is enough to represent the message length. For example, if the message was 70 bytes long, then the number 70 is converted to binary and left padded (if necessary) to be 8 bits long. 8 bits can have 2^8 (256) possible values which represents the numbers from 0 to 255. One byte of information can store the actual message length of 1 to 115 bytes easily. The actual message length is also encrypted with one byte from the one-time pad so it is impossible for an attacker to know the actual message length without the one-time pad. An attacker only knows that the length of a message is between 1 and 115 bytes. When decrypting the message, the message length value is checked to make sure it is in the correct range. This helps avoid denial of service and/or buffer overflow attacks.
- A UNIX timestamp is sent along with the message in the **Message sent timestamp** field to signify when the message was sent from the sender's computer. This is converted to binary and sent with the message packet. 5 bytes are reserved for

this. It could easily be 4 bytes (32 bits), but an extra byte was added to avoid the year 2038 problem. This timestamp is also encrypted with 5 bytes of the one-time pad. This prevents an attacker from interfering with the date or time of the message which could be critical in some circumstances. It is also used for correctly reordering messages on the client side when retrieving multiple messages from the server. This prevents an attacker reordering messages from them gaining access to the server or by delaying server responses.

- The final part of the encoding is the **MAC tag**. This MAC tag is sent along with each message for authentication and integrity to ensure that the message has not been tampered with. The MAC tag is also encrypted with part of the one-time pad so it is also information-theoretically secure. The process is explained in depth here.

Encryption process

- The program first receives the plaintext message from the user from the text box when they click the *Send message* button.
- Then it does a lookup on the user's local database of one-time pads and selects the first available one-time pad allocated to that user for sending messages. The one-time pads for sending/encrypting messages are evenly pre-allocated and grouped under each user in the chat group e.g. alpha, bravo, charlie etc. This prevents one user from encrypting a messsage using the same one-time pad as another user.
- Once a one-time pad has been selected, it removes it from the local database and splits it into the pad identifier, the message parts and the MAC parts. The encryption process is as follows:



- In step one, the plaintext with padding, the message length and the timestamp are converted to binary and concatenated together.
- In step two, because some of the bits of the 40 bit UNIX timestamp can be predictable, this could leave a crib for an attacker and they could recover those few bits of the key. This however would not compromise the remainder of the plaintext because each bit of the one-time pad is random and independent from the rest. For example, if the timestamp was 1406440512 for 2014-07-27 at 5:55am in UTC then that would convert to binary as **`00000000 01010011 11010100 10010100 01000000`**. If we compare another time in the future, 1503040500 for 2017-08-18 at 7:15am in UTC then that would convert to binary as **`00000000 01011001 10010110 10010011 11110100`**. The first 12 bits are the same in both timestamps even though the dates are years apart. This is because the timestamp field is larger than currently required in

order to future proof the protocol. If it was the usual 32 bits then eventually there would be incorrect dates and times shown in the program after 19 Jan 2038.

To remove this as a possiblity for being a crib, the program randomly reverses the binary message parts (including the plaintext with padding, the message length and the timestamp) depending on the second last byte in the one-time pad. It does this by converting this byte to an integer value (0 - 255), then uses that number modulo 2. This will return a random integer of 0 or 1. A one will mean the message parts get reversed while a zero will mean they stay the same. This means that every message, an attacker does not know for certain whether the timestamp is at the front or end of the message parts. They also do not know whether the true plaintext begins at the start of the message or the end. Because all users have the same one-time pad, they can reverse this transformation to get the message parts back in proper order after decryption. This transformation has a similar purpose to Russian copulation.

- In the third step, the Exclusive OR (XOR) operation is what does the encryption. Each bit of the plaintext is encrypted with one bit of the one-time pad. With a truly random one-time pad the encryption is unbreakable even in theory.
- The final step concatenates the pad identifier to the ciphertext message parts. The pad identifier helps the other users determine which one-time pad was used to encrypt the message.
- Once the message has been encrypted, the MAC is created using a random MAC algorithm that was selected and then encrypted with part of the one-time pad. This process is explained further on. The MAC is concatenated to the end of the ciphertext and sent with the message to the server. The server holds the message until all the other users have retrieved it.

Decryption process

The user first checks for encrypted messages on the server that are not sent by them and have not been read already by them. This will retrieve all other messages sent by users in the same chat group. Once the encrypted messages have been retrieved by the user, the messages are marked as read on the server by them. Once all users have read the message, they are deleted from the server in a cleanup process which runs every 30 seconds. The process for decryption is generally the same as encryption but in reverse order. For each encrypted message that is received:

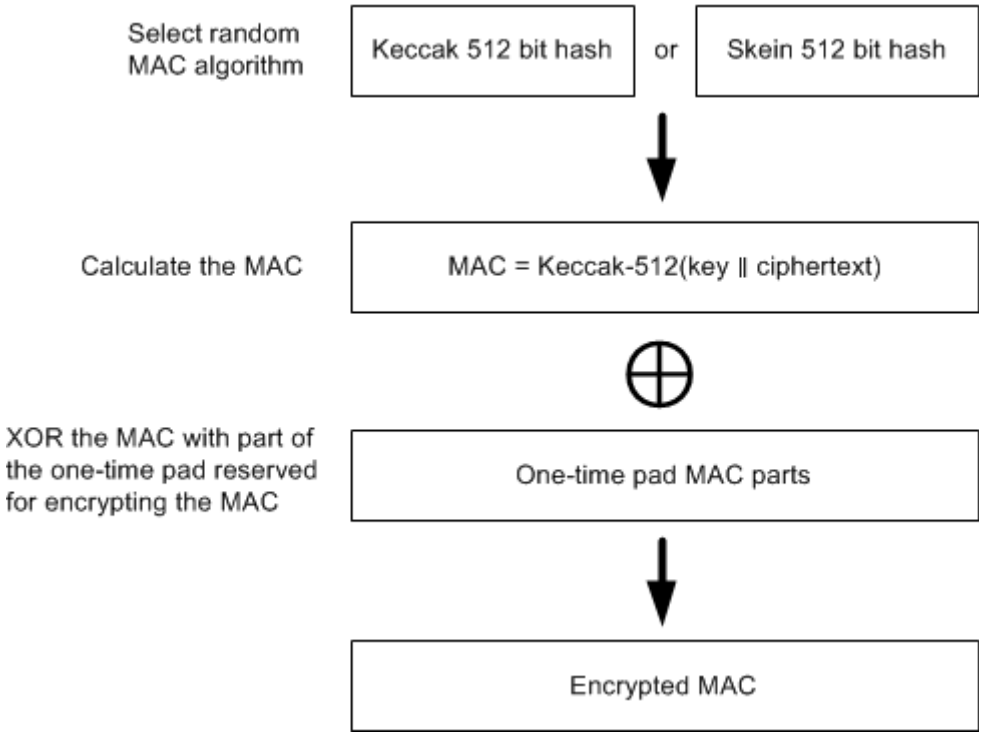
- The program selects the sender of the message e.g. alpha, bravo, charlie etc and does a lookup on the local database of

one-time pads for that user. The program selects the pad identifier (first 7 bytes) from the ciphertext which will match the first 7 bytes from one of the one-time pads in the database. The pad identifier for each one-time pad is stored in a separate field which makes searching faster. It then retrieves the one-time pad for the message and takes off the pad identifier.

- The MAC is then decrypted with the last 64 bytes of the one-time pad. Using the ciphertext message parts, the one-time pad and the random algorithm for the MAC, the MAC is calculated. If this matches the MAC sent, then the message is valid and has not been tampered with. Decryption of the message will follow. If the message matches the MAC sent with the message then an 'Authentic' status is displayed to the user. If the message is not valid, the user is warned that tampering has occurred and the decryption process will not be attempted.
- The one-time pad message parts is then XORed with the ciphertext message parts (including the plaintext with padding, message length and timestamp). This returns the decrypted text with padding, the actual message length and the time the message was actually sent.
- Depending on the second last byte of the key, the decrypted message parts are returned to their original order (unreversed) if that transformation was made in the encryption process.
- The message length part is read and this gets the length of the actual message in bytes. Reading from the start of the plaintext message up to the message length will retrieve the actual plaintext without padding.
- The one-time pads for any messages received and verified authentic are then deleted from the user's local database.

Message authentication code (MAC)

The one-time pad is vulnerable to a bit-flipping attack if not authenticated with a MAC. Therefore the program calculates and sends a one-time information-theoretically secure MAC with each message. Both users have a shared secret, which is the one-time pad for each message so the MAC can be calculated and verified by either person. The construction of this is as follows:



- For each message sent, a random MAC algorithm from a pool of algorithms is chosen to authenticate the message. This provides some protection in the case that a fundamental flaw is discovered in one of the hash algorithms in the future. It also makes message forgery considerably more difficult as an attacker needs to guess the correct hash algorithm as well. Currently there are only 2 hash algorithms that are used with the program due to the lack of JavaScript library support. More may be added in future. Potentially some of the NIST hash function competition finalists. Currently the hash functions used are the 512 bit versions of Keccak and Skein. These 512 bit hashes will provide 2^{256} collision resistance and 2^{512} pre-image resistance against regular computers. They will provide 2^{170} collision resistance and 2^{256} pre-image resistance against quantum computers.

  The program first gets a random index number from an array of available algorithms, then it uses this algorithm to create the MAC. It selects the random array index by using the last byte of the one-time pad. It converts this byte to an integer

value (0 - 255), then uses that number modulo the number of MAC algorithms available. Because there are only two MAC algorithms at the moment, that will return an integer of 0 or 1 which references the index of the algorithm in an array.

- The process is to perform the encryption on the message parts first, then calculate the MAC from the ciphertext and use the one-time pad as the key. This provides integrity of the ciphertext and integrity of the plaintext. Also it does not provide any information on the plaintext since no structure from the plaintext has been carried into the MAC. Skein and Keccak are newer, more secure hash algorithms and do not need complicated constructions like HMAC to prevent length extension attacks unlike the NSA designed SHA1 and SHA2. The MAC can be created simply by prepending the message with the key and hashing it, i.e. $H(K \mid\mid M)$.
- Finally the MAC tag is encrypted with the last 64 bytes (512 bits) of the one-time pad. This retains the information-theoretically secure properties for the MAC tag as well as the message. No attacker can know if they have successfully deciphered the encryption by brute forcing combinations of the key to create a valid MAC tag. Nor can an attacker know if they have created a successful forgery when they do not know the correct key.

Auto nuke process

The protocol normally erases the one-time pad as soon as a message is sent. The one-time pad is also removed from a receiver's database after they have sucessfully received and authenticated a message. This is a more secure form "off the record" chat similar to the OTR messaging protocol. OTR has good principles but lacks the perfect secrecy and plausible deniability of the one-time pad.

One of the key features of the program is being able to instantly and automatically wipe the local database of one-time pads, the server database of encrypted messages, the other user's database of one-time pads and clear any messages remaining on screen.

This should be initiated in an emergency situation only. Potentially if a chat group user believes their database of one-time pads may be compromised soon, or 3 letter agencies are inbound on a helicopter assault then they should initiate the auto nuke. This means the users are no longer in possession of the decryption keys so it means they cannot be compelled to produce them under duress or in a court of law. No encryption keys means no way to decrypt past messages. Without the real encryption keys, a user under duress can easily think of any plausible plaintext message for any encrypted message and an

aggressor will not know the difference. A simple way to calculate this for a one-time pad, given any ciphertext is to simply create a fake message, convert the ciphertext and fake message to binary, then XOR them together which will produce a plausible key to give to an attacker.