Jeff Connelly
CPE 456
June 11, 2008

A Practical Implementation of a One-time Pad Cryptosystem

0.1 Abstract

How to securely transmit messages between two people has been a problem for centuries. The first ciphers of antiquity used laughably short keys and insecure algorithms easily broken with today's computational power. This pattern has repeated throughout history, until the invention of the *one-time pad* in 1917, the world's first provably unbreakable cryptosystem. However, the public generally does not use the one-time pad for encrypting their communication, despite the assurance of confidentiality, because of practical reasons. This paper presents an implementation of a practical one-time pad cryptosystem for use between two trusted individuals, that have met previously but wish to securely communicate over email after their departure. The system includes the generation of a one-time pad using a custom-built hardware TRNG as well as software to easily send and receive encrypted messages over email. This implementation combines guaranteed confidentiality with practicality.

All of the work discussed here is available at http://imotp.sourceforge.net/.

Contents

1 Introduction

In this report I describe a practical implementation of a cryptosystem based on the one-time pad algorithm for perfectly secret communication between two people that previously met to exchange pads. Although the algorithm itself is trivial, the practical aspects of generating and using the pad to achieve maximum security in real-world situations requires careful engineering.

2 Implementation

2.1 Related Work

The one-time pad is said to have been invented in 1917 using modular addition. It was based on the Vernam cipher, divised by Gilbert Vernam, which combined a message from a loop of paper tape on which random characters were imprinted. The loop repeated, making it vulnerable to statistical cryptanalysis. Joseph Mauborgne recognizes that if the tape was made to not repeat and only truly random numbers were used, the system would be more difficult to crack. This led to the system making use of what is known as a *one-time pad*.

Two and a half decades later, Claude Shannon proved that the one-time pad has *perfect secrecy*. The ciphertext gives absolutely no information about the plaintext message.

Following this relevation, the world did not immediately begin using one-time pads everywhere, although they did see usage in special situations. The USSR used one-time pads in the 1930s, and the KGB has also used one-time pads. The U.S. National Security Agency (NSA) used one-time tape systems in operations known as SIGTOT and 5-UCO. British Special Operations Executives have used one-time pads to encode traffic between their offices during World War II, using cipher machines known as Rockex and Noreen.

Famously, the 1963 hotline between Moscow and Washington D.C. established after the Cuban missile crisis used one-time pads. The pad material was exchanged at their respective embassies, avoiding the need to for either country to reveal their secret cryptographic techniques to each other. The one-time pad served as a lowest-common denominator, albiet a secure one.

Soviet espinage during the late 1940s used a one-time pad in a project codenamed VENONA. However, the pad was reused and a few thousand of the several hundred thousand messages were successfully decrypted by the enemy. In the past, the FBI is said to have buglarized Soviet offices during World War II to obtain one-time pad material.

In modern times, several one-time pad implementations are freely available. Perl's CPAN offers Crypt::OTP, though no key management is included. Forumilab has a "high-quality pseudorandom sequence generator" it bills as a one-time pad generator. In 2001, Bernhard Spuida published a C# code to use one-time pads. Red Bean offers a program named OTP written by Karl Fogel, but the pad data offset must be specified manually with each decryption. Several trivial JavaScript one-time pad sites exist as well.

None of the implementions I've been able to find address the problem of ease-of-use. The authors seem to have implemented the one-time pad cipher for personal amusement and have not used it regularly to communicate with another individual in a real-world, practical setting.

2.2 Description

First, I connected a TRNG I previously assembled to my laptop and used it to capture truly random data.

I wrote a C library, `libotp`, to handle the pad management, encryption/decryption, and packaging/unpackaging. This library can be used directly or through `cotp`, a command-line tool to encrypt/decrypt messages.

Next I wrote `cotp.py`, a Python module to use `cotp` to encrypt/decrypt through pipes. This was needed so that the user interface could be written in Python, offering a richer array of functionality than would easily be implementable using portable C.

The next Python module, `SecureInbox.py`, interfaces with Gmail's IMAP and SMTP servers to allow sending and receiving messages. This module also performs the encryption/decryption by means of the `cotp` module.

Lastly, `cli.py` is the command-line interface to SecureInbox, allowing messages to be listed, read, and written through Gmail. The user first sets up a Gmail filter that tags all messages titled "(Secure Message)" with the "Secure" tag, and skips the inbox. `SecureInbox.py` reads messages from the IMAP "Secure" mailbox/Gmail tag so that in effect, `cli.py` presents an alternate interface to the user's mail, where all communication is secure. Through this interface, the encryption and decryption is transparent to the user.

The details of this project are examined in the remainder of this paper, along with the investigation and overall analysis of the body of work. There is no separate "Investigation" section; it is integrated throughout the rest of this paper.

3 Generating Randomness

The one-time pad algorithm requires a pad composed of truly random data to maintain the secrecy of the message.

3.1 Inadequacy of Pseudo-random Number Generation

Truly random data cannot be generated by purely deterministic means.

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." – John Von Neumann[1]

Pseudo-random number generators (PRNGs) produce sequencies of seemingly-random numbers, commonly through feedback shift registers. There are various formulations, including linear feedback shift registers, linear congruential generators, generalized feedback shift registers, twisted generalized feedback shift registers, and Tausworthe generators[2], but all PRNGs have a finite period. Furthermore, PRNGs are *seeded* with an initial state, and by knowing the state one can generate all future numbers in the sequence. For most PRNGs, knowing the initial state also allows computation of all previous numbers in the sequence. Common PRNGs include:

- RANDU, only of historical interest
- Standard C Library `rand()`, standardized in ISO C90
- Standard C Library `random()`, first appearning in 4.2BSD

Cryptographically-secure PRNGs have been created to satisfy the property that if the state of the generator is known at a given moment, the numbers previously generated in the sequence cannot be found. Cryptographic PRNGs include:

- Mersenne Twister[3] / CryptMT[4], a fast generator with an extremely large period of $2^{19937}-1$
- Block ciphers in counter or output feedback mode

- Yarrow algorithm in Mac OS X and FreeBSD
- Microsoft's CryptGenRandom, although a cryptanalysis in 2007 revealed significant weaknesses[7]
- Blum Blum Shub[5][6]
- Fortuna
- Unix `/dev/random` on modern systems

Additionally, stream ciphers such as RC4 are essentially cryptographic PRNGs whose pseudo-random stream is XOR'd with the plaintext to produce the ciphertext.

While cryptographic PRNGs offer marginally better security, including periods too large to feasibly rollover, compromising the internal state of the algorithm will still allow future encrypted messages to be decrypted by an adversary. This threat can be protected against by periodically adding randomness to the generator from an outside source, such as disk events, network activity, or interrupt timing. The pool is periodically "stirred" with true randomness.

Greater randomness from truly random physical processes is of course desirable, so taking it to the limit would be acquiring *all* randomness from an external, unpredictable, nondeterministic physical source. This is the truly-random random-number generator.

3.2 Truly Random Data

Truly random numbers can only be generated by a physical process and cannot be generated via software.[8]

A few services exist to provide truly random numbers:

- random.org, atmospheric noise
- Hotbits at Fourmilabs[9], radioactivity from Cesium-137

Hotbits also provides a Java class, `randomX`, to programatically obtain random bits from their radioactive source. The problem with these services is that they trust the operator of the site. It may also be possible to intercept the bits as they are transmitted over the Internet. Hotbits provides the random bits optionally over an SSL channel, but then SSL becomes a "weak link" in the cryptosystem; if the algorithms used in SSL (often RSA/DSA, AES, and SHA) are compromised, the security of the whole cryptosystem is too. A one-time pad is of no use of it is public.

Therefore a truly random number generator (TRNG) that can be used in the privacy of one's own home is needed. The Intel i810, Pentium III, and VIA chipsets have a built-in random number generator for this purpose, producing its noise from a semiconductor junction or from Johnson thermal noise (shot noise) from a resistor[10]. Building a reliable TRNG into every computer making it available for application use would be ideal, but the majority of computer systems lack TRNG hardware.

Commercial random number generators can be purchased for about $140 to $350; a slightly dated list of products is available at http://world.std.com/r̃einhold/truenoise.html. The cost of such devices prohibits casual use.

Random data can be obtained from human input using common peripherals available on many computers, such as sound, video, keyboard or mouse input RFC 1750[11] makes recommendations on doing this. Indeed, many public/private keypair generation tools ask the user to wriggle their mouse to generate sufficient randomness to generate the keys. Unfortunately mouse movements are a relatively low bandwidth means of generating truly random data, making them unsuitable for generating all but the shortest one-time pad.

Fortunately TRNGs can be built from a handful of expensive easily obtainable off-the-shelf electronic parts. There are a number of designs available online of varying quality and complexity. One hard requirement for my purposes is that the TRNG must not use the RS-232 serial port to communicate with the host PC, because my computer, like most modern laptops, does not have a serial port. This leaves: USB 2.0, FireWire 800, 1 Gb/s Ethernet, infrared, Bluetooth, 802.11n Wi-Fi, video input, and line-in audio input.

I decided on a slightly modified circuit originally designed by David Eather in 2004[12]. He considered several circuit designs, in terms of performance, complexity of construction, and ease of interfacing to the PC. Eather's TRNG interfaces with a personal computer through the audio line-in jack, and requires a minimal number of components to build. The randomness is obtained from a reverse-biased PN junction in an NPN transistor, which is said to be noiser than noise from zener diodes that manufacturers work hard to minimize.

4 Software

4.1 Acquiring Audio

Audio can be acquired using a third-party audio capture program such as Audacity[13].

During testing it is useful to listen to the noise generator without recording it, in order to check if it is functioning correctly. On Mac OS X, this can be accomplished using Rogue Amoeba's LineIn utility[14]. If functioning correctly, sound reminiscent of a waterfall or light raindrops is present.

Mac OS X has a CoreAudio API for audio input and output. MTCoreAudio provides an easier-to-use Cocoa-like wrapper to use this API. However, no integrated tool is currently available to capture, analyze, and generate pads at the moment.

A video of capturing the output of a TRNG in Audacity is available at http://tinyurl.com/5ymk4x .

4.1.1 Interference

Several sources of interference were observed:

- 60 Hz hum from AC power mains
- AM radio stations broadcasting Dr. Laura
- External hard drive motor power up noise

These sources of interferences are low-entropy. However, if mixed properly, they do not need to specifically filtered, since while it doesn't increase the entropy of the pool significantly, adding low-entropy data does not decrease it.

4.2 Measuring Entropy

`ent`[15] measures entropy in bits per character, how much optimum compression would reduce a file by, chi-square distribution, arithmetic mean, Monte Carlo value for Pi, and serial correlation coefficient.

Diehard is another common test of randomness.

Note that PRNGs can produce data that will pass these tests for entropy, over a finite period. The data will repeat after some time. For this reason, entropy tests alone do not ensure that data is truly random.

In the demonstration video at http://tinyurl.com/5ymk4x , the data straight from the TRNG (amplified several times) had a serial correlation efficient of 0.024 as measured by John Walker's ent program.

4.3 Entropy Extraction

David Eather estimates that his TRNG will produce 5 bits of entropy per byte. The output of this TRNG should not be used as a pad directly because the non-random bits may allow cryptanalysis of the message. *Entropy extraction* is necessary to "compress" the medium-entropy data into a high-entropy file usable as a one-time pad.

Entropy extraction methods include:

- Repeated condensing[19]
- Extractors from Reed-Muller codes[22]
- De-skewing
- Mixing

4.3.1 De-skewing

Noise input may be biased towards certain bits. RFC 1750[11] suggests several methods to de-skew the input:

- Stream parity
- Transition mapping
- Fast Fourier Transform
- Compression

Stream parity involves computing the parity of a sequence of bits of fixed length, and replacing the sequence with the parity bit, whether it be even or odd.

Transition mapping was proven by von Neumann[17] to remove bias, and is done by mapping the bit sequences as follows:

Input Bits	Output Bits
00	(discard)
11	(discard)
01	1
10	0

The Fourier transform can reveal strong correlations to be discarded.

Lossless data compression algorithms also deskew the input, but may add fixed headers or other sources of non-randomness.

4.3.2 Mixing

RFC 1750 (1994) [11] discusses several methods that can be used for mixing, including a symmetric cipher, Diffie-Hellman key exchange, or a secure hash. Several options are summarized below, including those from RFC 1750 but also more modern alternatives:

Algorithm	Input Bits	Output Bits
US Data Encryption Standard (DES)	120 (64 key + 56 data)	64
US Message Digest Standard (MD2/4/5)	arbitrary	128
Diffie-Hellman-Merkle key exchange	variable	variable
US Advanced Encryption Standard (AES)	256 (128 key + 128 data)	128
	320 (192 key + 128 data)	128
	384 (256 key + 128 data)	128
NIST Secure Hash Algorithm, SHA-1/256/512	arbitrary	160, 256, or 512
Tiger-192 Cryptographic Hash Function[16]	arbitrary	192
RACE ... Evaluation Message Digest (RIPEMD)	arbitrary	128, 256, or 320
WHIRLPOOL Hash Function	arbitrary	512

Hash functions offer the most flexibilitiy. The input size can be varied to adjust the "compression ratio" of input:output size. This ratio can be below 1 to expand the input, but as this does not increase the entropy it should not be used. Instead, large compression ratios will take a large noise capture and reduce it to a medium-sized pad of maximum entropy.

I implemented a simple mixing program in `mix.py`. This script uses SHA-512 to condense N bytes to a multiple of 64 bytes (the hash length of SHA-512). The default in:out ratio is 2:1, but any ratio can be specified as an argument. For example, 256 bytes of data with a 2:1 ratio will mix to 128 bytes, half the size. Higher ratios can be specified if the data is believed to be less random. Ratios below 1 are even allowed. 0.5 will expand the data, doubling the size, decreasing the true randomness (however, it may appear somewhat random due to the nature of SHA-512). Of course, decreasing the entropy is not useful for this application; mix.py will exclusively be used to increase the entropy. It should be used on the raw data (without any headers) captured from the sound card from the TRNG.

A screencast of running `mix.py` is available at http://tinyurl.com/5ymk4x . With a 2: ratio, the serial correlation coefficient decreased from 0.023759 to 0.002156, where 0.0 indicates totally uncorrelated. The arithmetic mean of all the bytes also changed from 133.5074 to 127.5122, where 127.5 is ideal. Based on these statistics, the mixing works correctly.

5 Exchanging Pads

After high-quality random data is generated by one person, it is time to exchange the pad, so that both parties have the identical data.

Ideally the pad exchange will be done in person. The pad can be saved to a CD-ROM by the person that generated it (call him Bob), and brought over to the other person (Alice). Bob will insert the CD-ROM into Alice's computer and copy the pad to her disk drive. After copying, the CD-ROM will be microwaved to destroy it.

The pad should in no cases be transmitted over the Internet, encrypted or unencrypted. The unencrypted case exposes the pad to possible interception, and even if it is encrypted the cryptosystem's strength is only as strong as the encryption of the one-time pad.

The lack of in-person availability prevented the testing of this program with other people, but I tested it between me and myself.

5.1 Merkle Channels

However, with some care it is possible to relatively safely exchange pads remotely without meeting in person.

This is possible using *Merkle channels*: sending information required to reconstruct the pad over multiple independent communication channels, so that an attacker must sniff each channel. For example, the pad could be encrypted, burned to a CD-ROM, sealed in a tamper-evident envelope, and mailed via postal mail (the first channel). The temporary passphrase used to encrypt the pad can be transmitted over telephone (the second channel). This scheme is not completely safe from sniffing. Pad exchange in person should be used for the best security.

5.2 Local Pad Security

After the pads are securely exchanged, their confidentiality must be maintained in order for the encrypted messages to be confidential. This is accomplished primarily by physical security of the computers on which the pads reside. This cryptographic suite would not be appropriate for use on a public terminal because the pad could possibly be compromised; it is soley intended for use on personal computers on which the owner has full control.

Physical security includes locks on doors to the house in which the computer resides, locks connecting the computer to a desk, and having the user closely watch the computer keeping it in his or her presence. For example, if the computer happens to be a laptop or other mobile device, the owner is expected to keep it within sight at all times when travelling.

A computer system may also have additional software controls, such as requiring a password when returning from sleep. Encrypting the pad with a symmetric cipher adds another layer of security to protect against a situation where the computer is stolen or otherwise compromised, therefore employing the principle of *defense in depth*. The symmetric cipher's key is a local password, chosen and only known by the individual use. Note that the security of messages encrypted using the one-time pad is not reliant on this feature, as the pad would be symmetrically decrypted before use.

The block cipher *counter mode* ("CTR") would be most appropriate for local pad encryption, as this mode allows for arbitrary portions of the pad to decrypted without decrypting all of the pad before the desired offset.

Local pad encryption was originally planned for this project, but it was not completed. As a workaround, an encrypted disk image can be used. On Mac OS, an encrypted dmg is suitable, and on Mac OS X, Linux, or Windows, TrueCrypt is a good option. These tools allow a file to be decrypted and mounted as a real filesystem. The cryptography tools and pads could be placed in this image.

6 Encrypting and Decrypting Messages

Encryption and decryption is simple as XORing a unique portion, K, of the pad with the ciphertext or plaintext. Formally:

$$C = E(K, P) = P \text{xor} K$$
$$P = D(K, C) = C \text{xor} K$$

The contents of the ciphertext gives no information on the contents of the plaintext message, since any message P is equally likely to be enciphered to any ciphertext C because of the truly random nature of K. (The *size* and *timing* of the message, however, may reveal information. This is discussed later.)

The key must come from a part of the one-time pad that has never been used before. Libotp accomplishes this by keeping a running counter of the byte offset in the pad that was last used for encryption, incrementing it to the next available byte offset after each encryption. The first

message, of length n, uses bytes K_0 to K_n. The offset is locally stored on the user's machine in a file named after the one-time pad filename, with a `.off` extension suffixed. Libotp implements `read_offset()` and `write_offset()` to change the offset, although the offset is advanced automatically in `otp_encrypt()`.

6.1 Delay Attack: Messages Crossing In The Mail

Consider if Alice and Bob both encrypted messages using the same shared pad, K. Alice sends M_1, a 100-byte message to Bob:

$$A \rightarrow B : \mathrm{E}(K_{0-99}, M_1)$$

Bob *should* decrypt this message and advance his internal pad counter, so that the reply (suppose it is 100 bytes) he sends to Alice is encrypted with a new portion of the pad:

$$B \rightarrow A : \mathrm{E}(K_{100-199}, M_2)$$

No part of the pad would be reused in this scenario. However, this is not guaranteed to happen. Bob may be currently composing a message, or have not read Alice's messages for another reason. Bob may have not received Alice's messages due to timing delays in email, either of natural origin or maliciously induced by Mallory. If Bob does not decrypt Alice's message and advance his offset, he will send:

$$B \rightarrow A : \mathrm{E}(K_{0-99}, M_1)$$

Bob reused the same bytes of the pad that Alice used! Clearly, this is undesirable, as the one-time pad became a two-time pad.

This attack is best prevented by requiring Alice and Bob to have separate sending and receiving pads. Alice's sending pad is Bob's receiving pad, and vice versa, and both parties have copies of the pads. The delay attack now fails because Alice and Bob do not encrypt with the same pad.

6.2 Pad Reuse Attack

Another vulnerability occurs if the users intentionally modify the offset to reuse part of the pad. This may be done if the pad is running low, but Alice and Bob still wish to communicate with the illusion of security.

Suppose Alice sends the encrypted message $M_1 \oplus K$ and later sends another encrypted message with the same key, $M_2 \oplus K$. Even if the second message is sent years after the first, a resourceful Eve can be assumed to have captured and logged all transmissions from Alice, and will be able to compute $(M_n \oplus K) \oplus (M_m \oplus K)$ for all m, n (simply XORing each pair of ciphertexts together). After an exhaustive search, Eve will compute $(M_1 \oplus K) \oplus (M_2 \oplus K)$ which reduces to $M_1 \oplus M_2$. No truly random data (pad data) is included in this ciphertext: the pad has been "cancelled out" of the ciphertext by the double XOR.

A clever attacker can detect the lack of randomness (assuming the messages are not truly random, which is presumably the case; otherwise no real meaning would be transmitted!) and proceed to cryptanalyze the message. Since the message is encrypted with another message, it is equivalent to a *book cipher*, where one message is encrypted with natural language text, and can be cracked using frequency analysis along with digram and trigram analysis[18].

6.3 Modification Attacks

Libotp does not provide any integrity protection, preferring to completely focus on maximum confidentiality.

In particular, the one-time pad cipher does not provide any *diffusion*. One bit flip in the ciphertext corresponds to the same bit change in the plaintext.

To detect changes, one might include a cryptographic hash within the plaintext of the message, before encryption. However, this opens up the cryptosystem to a brute-force attack. An attacker could try every possible key, and would recognize the correct key by the stored hash matching the computed hash value. Although the keyspace for larger messages can be quite large, adding a hash compromises the perfect secrecy of the one-time pad.

Sending a hash outside of the encrypted message would also not provide integrity, because an attacker could easily modify the hash to match the ciphertext.

If desired, PGP or GPG could be used to sign the libotp-encrypted message, to detect the act of tampering with the message. This is not implemented in libotp because authentication is not a priority. An attacker can flip arbitrary bits in the plaintext, but without knowing their meaning, this effectively constitutes a denial-of-service attack on availability, in most cases. While it is true that it may be possible for an attacker to modify a message to change its meaning, it is not considered a serious enough threat for libotp to provide protection.

6.4 Size Analysis Attacks

Even if the ciphertext cannot be decrypted by an attacker, it still may be possible to partially deduce the meaning of the message by the length of the ciphertext. For example, consider the messages "I'm fine" versus "Well, the troubles first started when..." [20], 8 bytes versus 40. Short, rapid communications can also indicate negotiations.

A naive one-time pad cipher is vulnerable to ciphertext length analysis since the size of the enciphered text is exactly the size of the plaintext. Although this satisifies Shannon's property of secrecy systems that the message should not be expanded during encipherment[23], it leaves the system vulnerable to the attacks described above.

A solution is to *pad* the end of the plaintext with nulls before encryption. The nulls are encrypted as usual, and are indistinguishable from actual plaintext when encrypted. When decrypted, the recipient can simply discard the nulls at the end. Note that this padding algorithm cannot operate on binary data since the end cannot be reliably determined. This could be rectified by prepending the true plaintext length to the plaintext before encryption.

`libotp` defines a `PADDING_MULTIPLE` constant, which causes the plaintext to have nulls added such that its length is a multiple of this number, effectively removing the least significant bits from the message length. For maximum security, `PADDING_MULTIPLE` should be greater or equal to the largest message length that will ever be sent. This will cause every message to be exactly the same size, meaning an attacker can gain no information whatsoever from the message length.

7 Sending and Receiving Messages

7.1 Packaging for Transport

Encrypted messages generated with libotp are intended to be embedded within email messages or other textual mediums. This necessitates several features:

1. Message start marker

2. Message end marker
3. "ASCII armor"

The message start and end markers are `--EMOTP_BEGIN--` and `--EMOTP_END--`, respectively. These markers were based on PGP's `-----BEGIN PGP MESSAGE-----` and `-----END PGP MESSAGE-----`, where "EMOTP" signifies an "email one-time-pad" message.

ASCII armor is simply base-64 coding, also used with PGP. This algorithm translates arbitrary 8-bit binary ciphertext into a text-only alphabet composed only of the characters `A-Za-z0-9+/`.

Lastly, the pad offset is sent with the message. Optionally, the pad name is also sent. If it is omitted, the default pad is used. Examples of encrypted messages:

```
--EMOTP_BEGIN--997,
9uQtwe8t3gjEJFxiPvr3KwTsz74AKyZwI/9PmSvEDOKsIi7KEclFcFuY0181T7PZSQtCS+vbVjdO
pA+Vrsv/3+PPfuE+OsjyK1LkqeofMKXfR698vbM74y8t60fk1ckxvA+WCi4z9F0863bpsXGkRHbX
mg+yu6gNOLs5xtVx0ZohWyNPnS9U37+jVJBJ5TPgUtDVuoj5WjNioY3bZP3JXssxnUrUfT1wv4Fk
oXOQxkO0hboNaANd1FrVNUFf/n5OLAH5vFzo4Va3H3KCS2j8ue6AZDkxJJR0Q6RjdJzY8nKOJ6bk
B63IeDnt6YU=
--EMOTP_END--
```

With an explicit pad name:

```
--EMOTP_BEGIN--997,dc,
9uQtwe8t3gjEJFxiPvr3KwTsz74AKyZwI/9PmSvEDOKsIi7KEclFcFuY0181T7PZSQtCS+vbVjdO
pA+Vrsv/3+PPfuE+OsjyK1LkqeofMKXfR698vbM74y8t60fk1ckxvA+WCi4z9F0863bpsXGkRHbX
mg+yu6gNOLs5xtVx0ZohWyNPnS9U37+jVJBJ5TPgUtDVuoj5WjNioY3bZP3JXssxnUrUfT1wv4Fk
oXOQxkO0hboNaANd1FrVNUFf/n5OLAH5vFzo4Va3H3KCS2j8ue6AZDkxJJR0Q6RjdJzY8nKOJ6bk
B63IeDnt6YU=
--EMOTP_END--
```

In libotp, `MESSAGE` structures are packaged and unpackaged to base64-encoded delimited ASCII strings using `package()` and `unpackage()`, respectively.

7.2 Authentication

Libotp does not provide authentication nor any methods for insuring data integrity, as it focuses completely on preserving confidentiality.

If authentication is desired, a libotp-encrypted message could be further signed by PGP or GPG.

7.3 Channel Filling

With optimal padding, message lengths reveal no information. However, traffic analysis also takes into consideration the *timing* of message transmissions in an attempt to breach confidentiality.

Traffic analysis can be classified as passive or active[25]. Active attacks rely on sending traffic, such as pings, and observing the resulting traffic. An example is *idle scanning*, a stealth technique which proxies the scan through an idle host to map out open ports on a machine[24]. Libotp does not appear to be vulnerable to active traffic analysis attacks, provided both parties have their own (shared) sending pad. (If a single shared pad is used, an attacker could inject a fake message with a false offset, maliciously advancing the pad to the end, causing a denial-of-service attack.)

Passive traffic analysis can reveal the type of communication[21]:

Communication	May Indicate
Frequent	Planning
Rapid, short	Negotiations
None	No activity, or completion of a finalized plan
To a central station	Chain of command
Who talks to whom	Which stations are in control
Who talks when	Which stations are active in connection with events
Who changes from station to station	Movement, fear of interception

To combat traffic analysis, libotp could always send messages at a constant interval, such as once a day at a specific time. A message to be sent will be queued up and sent only at the decided interval. If no messages are ready to be sent, a fake dummy message of nulls will be transmitted.

8 After Sending and Receiving

Several operations can be performed after-the-fact for various reasons.

8.1 Pad Destruction

To prevent anyone from reading a message ever, even the original intended recipient and sender, a section of the pad can be destroyed. This should be done using a secure deletion program. Read the pad into memory, overwrite a section of it with nulls, securely delete the original pad, then output the new pad. The deleted part of the pad will be filled with zeroes and thus the corresoponding message will be indecipherable.

8.2 Pad Rewriting

It is possible to go a step further and instaed of destroying part of the pad, to rewrite it such that decrypting a message encrypted with that part of the pad results in plaintext determined after-the-fact. Any sent message can be repudiated by rewriting the pads.

With pad rewriting, it is possible for the shared pads between both parties to differ. Alice could have the same message decrypt to a different plaintext than Bob would, after both (or one) rewrote the message. If desired, the pads could be re-exchanged so they would be identical.

This is implemented in cotp with the -r command-line flag, and in cli.py with the x command. A screencast demo of cotp is available at http://tinyurl.com/3snaz2 and of cli.py at http://screencast.com/t/fZOl6q4O56m .

8.3 Limitations and Future Directions

Future improvements could be:

- Local pad encryption built-in to the program (current workaround is to use encrypted disk images).
- Tools to break a two-time pad (including a Kasiski tool).
- Improved pad management, to setup pairs of pads to communicate between more than two individuals.
- A graphical user interface for easier usage than the command-line.

- Improvements on the interface after performing a usability test.
- Channel filling when the program is not running, through the use of cron, launchd, or a scheduled task.

8.4 Conclusion

A practical one-time pad cryptosystem was implemented, from hardware to software.

The hardware is easily constructable using off-the-shelf components and allows for truly random data capture through a standard PC sound card. Although it produces an estimated 5 bits of entropy per byte, the data can be mixed down to any level of entropy using a Python script which hashes larger chunks of data into smaller chunks, therefore extracting the entropy.

The software is a command-line interface to the user's secure mailbox. The list of encrypted messages can be listed, encrypted messages can be sent, all through the interface which currently uses Gmail. A channel filling thread can be enabled to only send messages at a given period, queuing up messages to be sent, and sending dummy messages if no message is ready to be sent. Length padding to the maximum message size further prevents traffic analysis.

Overall, this project demonstrates that it is possible to apply the strongest cipher, the one-time pad, to practical circumstances involving communication between two trusted individuals. Although the setup may be more than it is worth for the casual secure communication, the system implemented here allows for communication of the highest confidentiality when such security is required.

References

[1] J. von Neumann, "Various techniques used in connection with random digits". National Bureau of Standards, Applied Mathematics Series, vol. 12, pp. 36–38, 1951.

[2] Pierre L'Ecuyer, Francois Panneton, *A New Class of Linear Feedback Shift Register Generators.* Départment d'Informatique et de Recherche Opérationnelle Université de Montréal. Proceedings of the 200 Winter Simulation Conference. Available: http://citeseer.ist.psu.edu/518016.html

[3] M. Matsumoto and T. Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. Model. Comput. Simul. 8, 3 (1998). Available: http://portal.acm.org/citation.cfm?doid=272991.272995

[4] Makoto Matasumoto, Takuji Nishimura, Mariko Hagita, and Mutsuo Saito . Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher. Hiroshima University. June 1, 2005. Available: http://eprint.iacr.org/2005/165.pdf

[5] Lenore Blum, Manuel Blum, and Michael Shub. "Comparison of two pseudo-random number generators", *Advances in Cryptology: Proceedings of Crypto '82.* Available: http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C82/61.PDF

[6] Lenore Blum, Manuel Blum, and Michael Shub. "A Simple Unpredictable Pseudo-Random Number Generator", *SIAM Journal on Computing*, volume 15, pages 364383, May 1986.

[7] Leo Dorrendorf, Zvi Gutterman, Benny Pinkas. "Cryptanalysis of the Random Number Generator of the Windows Operating System." November 4, 2007. Israel Science Foundation Grant. Available: http://eprint.iacr.org/2007/419.pdf

[8] Planet Math. "Truly Random Numbers." Derived from the Data Analysis Briefbook. Available: http://planetmath.org/encyclopedia/TrulyRandomNumbers.html

[9] Walker, John. "HotBits: Genuine random numbers, generated by radioactive decay." Fourmilab Switzerland. Available: http://fourmilab.ch/hotbits/

[10] Jun, Benjamin and Kocher, Paul. The Intel(R) Random Number Generator. Cryptography Research, Inc. White paper prepared for Intel corporation. April 22, 1999. Available: http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf

[11] D. Eastlake, 3rd (DEC), S. Crocker (Cybercash), J. Schiller (MIT). RFC 1750: Randomness Recommendations for Security. Network Working Group. December 1994. Available: ftp://ftp.ietf.org/rfc/rfc1750.txt

[12] Eather, David. Random Number Generation with a Simple Transistor Junction Noise Source. Version 1 Revision 2. 22 June 2004. Available: http://users.tpg.com.au/users/eather/images/noise.pdf Mirrored at: http://imotp.sourceforge.net/noise.pdf

[13] Audacity: Free Audio Editor and Recorder. GNU General Public License. Mac OS X, MS Windows, Unix. Available: http://audacity.sourceforge.net/

[14] Rogue Amoeba. LineIn: Inside The Lines. Enables soft playthru of audio from input devices. Available: http://rogueamoeba.com/freebies/

[15] Walker, John. "ent - pseudorandom number sequence test". Forumilab. January 28th, 2008. Available: http://www.fourmilab.ch/random/

[16] Anderson, Ross and Biham, Eli. "Tiger - A Fast New Hash Function". Proceedings of Fast Software Encryption 3, Cambridge. 1996. Available: http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html

[17] von Neumann, J. Various techniques used in connection with random digits, von Neumann's Collected Works, Vol. 5, Pergamon Press, 1963

[18] Nico, Phillip. "The Book of the Class for cpe 456". Spring 2008. Department of Computer Science, California Polytechnic State University, San Luis Obispo, CA 93407. May 23, 2008. Available: http://users.csc.calpoly.edu/~pnico/class/s08/cpe456/notes/class.pdf

[19] Reingold, O., Shaltiel, R., Wigderson, A. "Extracting randomness via repeated condensing" *Proceedings of the 41st Annual Symposium on Foundations of Computer Science* IEEE Computer Society. Available: http://citeseer.ist.psu.edu/reingold00extracting.html

[20] Hettinger, Raymond. "A Monograph on Cryptographic Attacks and Countermeasures". Available: http://users.rcn.com/python/cipher.htm

[21] Wikipedia. "Traffic analys – Wikipedia, the free encyclopedia". Available: http://en.wikipedia.org/wiki/Traffic_analysis

[22] Ta-Shma Amnon, Zuckerman David, Safra Shmuel. "Extractors from Reed-Muller codes" *Journal of Computer and System Sciences*, Volume 75, Issue 5 (August 2006), Special issue on FOCS 2001. Pages 786-812, 2006. http://portal.acm.org/citation.cfm?id=1150653

[23] Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656715, 1949. Available: http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf

[24] Fyodor. Nmap: Idle Scanning and Related IPID Games. Available: http://nmap.org/idlescan.html

[25] Xinwen Fu, Bryan Graham, Riccardo Bettati, Wei Zhao. Active Traffic Analysis Attacks and Countermeasures. Available: http://students.cs.tamu.edu/xinwenfu/paper/ICCNMC03_Fu.pdf