Mobile Phone Based Dead Drop Communication

Keith Stringer
B.A.I. Engineering
Final Year Project  April 2015
Supervisor: Donal O'Mahony

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

__________________________________________  ________________
Name                                      Date
# Table of Contents

1 Introduction .................................................................................................................. 5  
2 State of the Art ............................................................................................................. 6  
   2.1 Introduction ............................................................................................................. 6  
   2.2 Symmetric Cryptography ....................................................................................... 6  
      2.2.1 The Origins of DES ........................................................................................ 7  
      2.2.2 How It Works ................................................................................................. 7  
      2.2.3 Decryption ..................................................................................................... 9  
      2.2.4 Decline of DES ............................................................................................. 9  
      2.2.5 Advanced Encryption Standard (AES) ......................................................... 9  
      2.2.6 How it Works ............................................................................................... 10  
      2.2.7 Decryption .................................................................................................. 11  
      2.2.8 One Time Pad ............................................................................................... 11  
2.3 Asymmetric Cryptography ....................................................................................... 12  
2.4 Steam Ciphers vs. Block Ciphers ............................................................................ 13  
2.5 Key Agreement ......................................................................................................... 14  
2.6 Hashing ..................................................................................................................... 15  
      2.6.1 Message Digest ............................................................................................. 16  
      2.6.2 Secure Hash Algorithm ............................................................................... 18  
2.7 Message Authentication Codes (MAC) ................................................................... 20  
2.8 Digital Signatures ...................................................................................................... 21  
2.9 Key Derivation .......................................................................................................... 21  
      2.9.1 Single Step Key Derivation ........................................................................... 21  
      2.9.2 Extraction then Expansion Key Derivation ................................................ 22  
      2.9.2.1 Randomness Extraction ........................................................................ 22  
      2.9.2.2 Key Expansion ......................................................................................... 23  
2.10 Attacks .................................................................................................................... 23  
      2.10.1 Types of Break: ........................................................................................... 23  
      2.10.7 Man in the Middle ..................................................................................... 25  
      2.10.8 Replay Attacks ........................................................................................... 25  
      2.10.9 Hash Attacks ............................................................................................... 25  
2.11 Inspirational Systems ............................................................................................. 26  
      2.11.1 Pretty Good Privacy (PGP) ........................................................................... 26  
      2.11.2 Off-the-record Messaging ......................................................................... 27  
      2.11.2.1 Discussion ............................................................................................ 28  
      2.11.3 FireChat ..................................................................................................... 28  
3 Design: ........................................................................................................................... 30  
   3.1 What the System Does .......................................................................................... 30  
   3.2 Generating A Secret .............................................................................................. 30  
   3.3 Generating A Key ................................................................................................... 33  
   3.4 Message Format .................................................................................................... 33  
      3.4.1 Message Header ............................................................................................ 33  
      3.4.2 Ciphertext .................................................................................................... 35  
      3.4.3 Message Footer ............................................................................................ 35  
   3.5 Sending a Message ............................................................................................... 36  
   3.6 Receiving a Message ............................................................................................ 37  
   3.7 Storing Information ............................................................................................... 37  
   3.8 Generating New Keys ........................................................................................... 38  
4 Implementation ............................................................................................................ 39  
   4.1 What I Implemented .............................................................................................. 39  
   4.2 Tools ..................................................................................................................... 39
1 Introduction

A dead drop is a method of covertly exchanging items or information between two individuals. The individuals do not need to meet, nor do they need to be aware of the other’s identity. The information is left in a known location by one party, and retrieved by the other. The depositor can provide a signal, informing the receiver that a drop has been made. While it is not important that the two parties meet, it is important that they are both aware of the designated location or drop spot. Historically speaking, dead drops have been associated with espionage, secrecy and criminality. Exchanged items have typically been instructions, money and secret documents. Dead drop locations have been under rocks, holes in trees, even inside dead animals, while signals could simply be a chalk mark on a wall. In more modern times, USB devices have been “dropped” as they can store considerably larger quantities of information.

The goal of the project was to design a system that could mimic this behaviour, but instead of a physical drop spot, an online location would be used. Rather than hide the message in plain sight, it would be encrypted. Users of the system would not need know the other’s identity, but must be sure that they are communicating with the intended party.

Finally, this was to be implemented on a chosen platform, Android smart phones.
2 State of the Art

2.1 Introduction

“The art and science of keeping messages secure is cryptography” [Applied Cryptography]. It deals with the means and methods by which information is manipulated such that only intended parties can understand it. It involves confidentiality, authentication, integrity and nonrepudiation. Confidentiality means that information is kept secret from unintended parties and is achieved by encryption. Authentication of a message is confirmation of the identity of both the sender and the receiver. Integrity is confirmation that the message has not been altered. Non-repudiation is the ability to demonstrate that a message originated from a message sender (i.e. it aims to prevent a sender from falsely denying sending a message) [RFC3552]. It differs from authentication in that the receiver must be able to prove to a third party that the sender sent the message, rather than just being able to confirm their identity.

Cryptographic key-based algorithms can be split into groups depending on the relationship between encryption and decryption [Applied Cryptography]. These are symmetric and asymmetric.

2.2 Symmetric Cryptography

Symmetric algorithms use either the same key (or related and derivable keys) for both encryption and decryption [Applied Cryptography]. The security of this approach is based on keeping the key a secret, known only to the sender and receiver of a message.

\[ f_e(M, K) = C \]
\[ f_d(C, K) = M \]

Encryption is a function \( f_e \) of the plaintext message \( M \) and the key \( K \), the result of which is ciphertext \( C \). Decryption is a function \( f_d \) of the ciphertext and the key, the result of which is the original plaintext message.
2.2.1 The Origins of DES

In the 1970s, the National Bureau of Standards (NBS), now known as The National Institute of Standards and Technology (NIST), recognised that a standard cryptographic algorithm was needed [Applied Cryptography]. At the time strong cryptography was limited to governments and militaries, who did not share algorithms or research with the public. Cryptographic equipment could be purchased, but the solutions had many problems. They could not interoperate with other cryptographic equipment nor was there any guarantee of security due to the absence of independent bodies to carry out cryptanalysis and certify products.

The NBS issued a public request for proposals for a standard cryptographic algorithm in 1974 [Applied Cryptography]. An algorithm named Lucifer, developed by a team at IBM, was among those submitted. After feedback from the NBS, the NSA (who were consulted by the NBS) and the public had been taken into account, the Lucifer algorithm was modified and adopted as a federal standard (US) on the 23rd of November 1976. Known as the Data Encryption Standard (DES), it was the first NSA evaluated algorithm to become public. In 1981, DES was approved as a private sector standard by The American National Standards Institute (ANSI).

DES is a block cipher that operates on 64-bit blocks. It uses a fixed length 56-bit key (though this may be represented as a 64-bit number, with every eighth bit used for parity checking). Both encryption and decryption use the same key and do not change the length of the output.

2.2.2 How It Works

DES was announced as a federal standard in Federal Information Processing Standard (FIPS), Publication 46. This publication also described the internal workings. Whether it is encryption or decryption, the DES algorithm is roughly the same. First, an initial permutation is performed (on the 64-bit input block), followed by 16 rounds consisting of identical operations, before a final permutation is performed.
**Initial Permutation:** This rearranges the bits in the 64-bit block. It is performed to simplify the loading and unloading of blocks onto hardware chips that implement DES. It does not offer additional security and has been ignored in many software implementations.

**16 Rounds:** Each round consists of an identical set of operations based on the key and the intermediate data (64-bit output from the previous round or the initial permutation). The 64-bit input block is divided into two, a left half and a right half.

**Key Transformation:** The first step involves generating a 48-bit sub key from the 56-bit key (though it may require removing the parity bits from the 64-bit key). Each round will use a different sub key. It is generated by dividing the 56-bit key in two and circularly shifting each of the halves by a number of bits dependent on the round.

**Expansion Permutation:** The next step involves expanding the right half of the data block from 32 to 48 bits, as well as changing the order (note that some bits will be repeated). This is done to match the length of the sub key and to allow one bit to affect two substitutions in an effort to make every bit dependent on both the key and the input as soon as possible. The expansion is performed on 4-bit blocks, such that the new 6-bit block consists of the final bit from the previous input block, followed by the original 4-bit input block, followed by the first bit from the next input block. The 48-bit output is then XORed with the sub key. The result undergoes a substitution operation.

**S-Box Substitution:** The 48-bit number is divided into eight 6-bit blocks. Each block is processed by a different substitution box (S-Box), each of which contain four rows and sixteen columns. The first two bits of the input are used to select the row while the other four bits are used to select the column. At each row column intersection, there is a four bit number, which is the output. The S-Box step is the most important from a security perspective as it is non linear, making it the most difficult to analyse.
**P-Box Permutation:** This step rearranges the 32 input bits based on a mapping table. Every bit is used once, and no bits are left out. If it is not the final round, the left and right halves are then switched and another round begins.

**Final Permutation:** This step is the inverse of the initial permutation and simply rearranges the bits based on a mapping table. The output is a 64-bit block of ciphertext. It only occurs for the final round.

### 2.2.3 Decryption

The only difference between DES encryption and decryption is the order of the sub keys used. If the sub keys for encryption are K₁, K₂, ... K₁₆ then the sub keys for decryption would be K₁₆, K₁₅, ... K₁.

### 2.2.4 Decline of DES

As part of being adopted as a standard, DES was required to be reviewed every five years by NIST. As early as the second review, in 1988, there were concerns over its security [FIPS 46-1]. In 1993, NIST certified DES for another five years, yet pointed out that while there are approximately 70,000,000,000,000 possible keys “it is theoretically possible to derive the key in fewer trials” [FIPS 46-2]. Essentially, DES was broken. It also pointed out that an algorithm that offered more security should be sought to replace DES as the standard. In 1999, NIST removed support for DES, with the exception of legacy systems, and instead encouraged the use of Triple DES (encrypting plaintext with K₁, then decrypting with K₂, then encrypting with K₃, where K₁, K₂ and K₃ were 56-bit DES keys) [FIPS 46-3]. It also stated that Triple DES would likely coexist with the new encryption standard which was being developed. The new encryption standard was announced in FIPS 197, published 28th of November 2001, which also describes the internal workings [FIPS 197].

### 2.2.5 Advanced Encryption Standard (AES)

AES is a symmetric block cipher that has three options for the key length, 128 bits, 192 bits and 256 bits. It operates on blocks of 128 bits and outputs blocks of the same size, regardless of key length.
The encryption or decryption process takes place in two steps, key expansion and rounds, though there are variations for the first and last round. The number of rounds is determined by the length of the key used. A 256-bit key uses 14 rounds, a 192-bit key uses 12 rounds and a 128-bit key uses 10 rounds (as well as the initial round).

### 2.2.6 How it Works

**Key Expansion:** This involves generating a larger key, the key schedule, to use in the algorithm. Every encryption or decryption requires an initial 128-bit key, followed by additional 128-bit keys for each round. For each of the key lengths:

\[
\text{Expanded Key Length} = 128(\text{Number of Rounds} + 1)
\]

A 128-bit key is expanded to a 176-byte key, a 192-bit key is expanded to a 208-byte key and a 256-bit key is expanded to a 240-byte key. The bits of the initial key is copied into the key schedule. 4-byte blocks are added to the initial key until the required length of the key schedule is reached.

**Rounds:** The internal workings involve operations on a 4x4 matrix of bytes, referred to as the state. Initially, the input 128 bit block is copied into the state.

![AES State Array – Image 1](image)

Each round uses a different 128-bit round key, taken from the key schedule. The initial round involves an XOR operation between the state array and the first 128 bits of the key schedule. The following steps are then repeated for the required number of rounds.

- **S-Box Substitution:** each byte in the state is used as an input into an S-Box. The output bytes are copied into the state.
• **Shift Rows:** This transformation cyclically rotates the rows of the state. The first row is not shifted, the second row is shifted 1 byte to the left, the third by 2 bytes to the left and the fourth by 3 bytes to the left.

• **Mix Columns:** Each column of the state is transformed by multiplying it with a 4x4 matrix, except for the final round which ignores this step.

• **Add Round Key:** This consists of an XOR operation between the state and the next 128-bit block of the key schedule.

### 2.2.7 Decryption
Decrypting ciphertext using AES is just the reverse of the encryption steps. The key expansion is the same, however, keys are used in reverse order. All of the operations on the state can be inverted. The add round key step is an XOR operation which is the inverse of itself. The mix columns operation can be performed using an inverse of the multiplication matrix used for encryption. The S-Box substitution can also be performed with an inverse of the S-Box. The shift rows step is simply reversed.

### 2.2.8 One-Time-Pad
In theory, there is only one encryption scheme that is unconditionally secure [Applied Cryptography]. It is called the one-time-pad. A message is encrypted and decrypted with a random key of the same length. This is done by performing an XOR operation.

\[ M \oplus K = C \]

\[ C \oplus K = M \]

The reason this is considered unbreakable is that the ciphertext is equally likely to correspond to any possible plaintext of the same length. So, if the key is truly random and kept secret, no cryptanalysis can be used to decipher the message. It is also important that the one-time-pad is used only once.
2.3 Asymmetric Cryptography

Asymmetric (or public-key) algorithms use different keys for encryption and decryption [Applied Cryptography]. The decryption key must not be computable from the encryption key. In general, the key for encryption (public key) is made public and can be used by anyone to encrypt a message. The key for decryption (private key) is kept secret.

\[
f_e(M, K_{\text{public}}) = C \]

\[
f_d(C, K_{\text{private}}) = M
\]

Encryption is a function \(f_e\) of the plaintext message \(M\) and the public key \(K_{\text{public}}\), the result of which is ciphertext \(C\). Decryption is a function \(f_d\) of the ciphertext and the key \(K_{\text{private}}\), the result of which is the original message.

The most widely used asymmetric cipher is RSA, named after its creators (Ron Rivest, Adi Shamir and Leonard Adleman). Its security is based on the difficulty in factoring large numbers [A Method for Obtaining Digital Signatures and Public-Key Cryptosystems]. To generate two keys, two large prime numbers \((p\) and \(q)\) are chosen at random \((p\) and \(q\) should be of equal length for security reasons). The product of \(p\) and \(q\) is calculated. This value, \(n\), becomes part of the public key. Next, a randomly chosen encryption key, \(e\), is chosen such that \(e\) is relatively prime to \((p-1)(q-1)\) and \(e\) is less than \((p-1)(q-1)\). The decryption key, \(d\), is computed using the formula:

\[
d = e^{-1} \mod (p - 1)(q - 1)
\]

After \(e\) (part of the public key), \(n\) (part of the public key) and \(d\) (the private key) have been computed, the values of \(p\) and \(q\) should be discarded.

In order to encrypt data, the public key \((e, n)\) is published. Anyone can use this to encrypt data, but only someone with the private key can decrypt it. Encryption is computed using:

\[
C = M^e \mod n
\]
Where $M$ is the plaintext message, $C$ is the ciphertext and $e$ and $n$ make up the public key. Decryption is computed using the formula:

$$M = C^d \mod n$$

Public keys must be distributed in order to be used. This is done by key distribution centres which maintain a list of keys and the identities they belong to.

### 2.4 Steam Ciphers vs. Block Ciphers

Steam ciphers operate on data one bit or byte at a time [Applied Cryptography]. Block ciphers operate on blocks of data. Stream ciphers generally use a pseudo-random number generator to create a key stream. This stream of bits is combined with plaintext using an XOR operation to produce the ciphertext. In order to decrypt the ciphertext, the same key stream must be generated. Two parties wishing to use a stream cipher must be able to reproduce this key stream. If it were fixed, then it would be possible to analyse the ciphertext and likely figure out the key stream (though this analysis is outside the scope of this report). In order to generate two pseudo-random yet identical streams, both parties must initialise the cipher with a secret key.

Stream ciphers are useful as they tend to be faster than block ciphers and the amount of data that needs to be encrypted does not need to be known in advance or the data can be continuous. However, the fact that each key can only be used once and that authentication and integrity are more difficult to provide are disadvantages when compared to block ciphers. As each bit of plaintext is converted to a bit of ciphertext, altering the ciphertext to produce a meaningful change in the plaintext is possible. This is unlike block ciphers where a modification to the ciphertext will likely produce meaningless plaintext once decrypted. Block ciphers can also be used to encrypt data in units smaller than the block size, effectively operating as stream ciphers.
2.5 Key Agreement

Assuming (and it is a big assumption) that a chosen cryptographic algorithm is secure and that it based only on keeping the key a secret, then a problem arises. How do you share or agree on a key to use? A number of options are available.

2.5.1 Diffie-Hellman Key Agreement

In 1976, Whitfield Diffie and Martin Hellman published a paper entitled "New Directions in Cryptography". They identified a need to reduce the necessity of secure distribution channels for current cryptographic systems, which required communicating parties to keep a secret key known only to them. Exchanging and generating secrets was a difficulty at the time. Keys had to be sent in advance over a secure channel. At the time, the secure channel was generally either a trusted courier or registered mail. This meant that in order for parties to communicate privately they had to make prior arrangements or wait for the key. The solution that they proposed, which became known as Diffie-Hellman Key Agreement, enabled two parties communicating over an insecure channel to establish a secure connection using published techniques.

The basis for their solution is that solving the discrete log problem is very difficult. A simplification of the discrete log problem is that it is easy to compute $g^x$ when we know $x$, but hard to compute $x$ from $g^x$. Solving Diffie-Hellman is as hard as solving the discrete log problem (once weak values of $p$ and $g$ are avoided) [Diffie-Hellman is as Strong as Discrete Log for Certain Primes].

The technique is implemented as follows, for two parties, Alice and Bob, who wish to generate a secret over an insecure channel in the presence of an eavesdropper, Eve:

1. Alice and Bob agree on $g$, the base, and $p$, the mod
2. Alice generates a random number $x$
3. Bob generates a random number $y$
4. Alice calculates $A$ (sometimes referred to as Alice’s Diffie-Hellman public key), such that:

\[ A = g^x \mod p \]
5. Bob calculates B (sometimes referred to as Bob’s Diffie-Hellman public key), such that:

\[ B = g^y \mod p \]

6. Alice sends A to Bob, while Bob sends B to Alice

7. Eve can see both A and B

8. Alice calculates the shared secret S, such that:

\[ S = B^y \mod p \]

9. Bob calculates the shared secret S, such that:

\[ S = A^x \mod p \]

10. It is infeasible for Eve to calculate S from A and B as this is equivalent to solving the discrete log problem.

There are many other ways to agree on secret keys for use in a symmetric cipher. Keys could be chosen at random and encrypted using the public key infrastructure.

2.6 Hashing

A hash function is a mathematical operation that takes a variable length input and computes a fixed length output [Applied Cryptography].

\[ f_H(M) = H \]

A hash function \(f_H\) computes a hash \(H\) from an input message \(M\). There are a number of properties that must be observed one-way hash function (those used in cryptography).

1. M is an input of variable length
2. H is an output of fixed length
3. Given M it must be easy to compute H
4. Given H it must be computationally infeasible to find M
5. \( f_H \) must be collision resistant, i.e. it is hard to find two inputs \( (M) \) that will produce the same output \( (H) \)

Hashing is used to provide data integrity. If a message is sent along with a hash, it is simple for the receiver to verify (with a strong degree of accuracy) whether the message has been altered. This is done by recomputing the hash, using the same message and algorithm, and comparing that to the one sent. In the event that the message has been altered (either maliciously or accidentally), these will not match. To ensure that a message has not been tampered with, it is important to protect the appended hash. This prevents an attacker from changing the message and recalculating the hash.

There are a number of other uses for hash functions outside the cryptography field. These have different properties, than those outlined above. For instance, when indexing data, a hash function can be used to map data to memory locations. Speed of computation and an even distribution of the hash values are more important than the difficulty of finding an input that corresponds to a given hash. There are two common families of hash algorithms that are currently in use in the cryptography field. These are the Message Digest (MD) and Secure Hash Algorithm (SHA).

**2.6.1 Message Digest**

The MD hash series was developed by Ron Rivest and includes MD2 (1989), MD4 (1990), MD5 (1992) and MD6 (2008) (which was developed with additional designers). The steps for computing the hash are relatively similar for each of the algorithms (except for MD6 which is significantly more complex).

A brief overview of the MD5 algorithm, as described in RFC1321, for processing a message of length \( b \) bits is:

**Append and Pad the Message:** The message of length \( l \) is extended by adding a bit with a value of 1, followed by \( k \) bits with a value of 0, such that:

\[
(l + 1 + k) \mod 512 = 448
\]

for the smallest possible value of \( k \).
**Append Length:** A 64-bit representation of b is appended to the padded message. If b is greater than $2^{64}$, only the 64 least significant bits are appended. Note that the appended message length mod 512 is now 0.

**Initialise the MD Buffer:** The MD buffer consists of four blocks 32-bits long. Constant values are moved into these blocks.

**Define Functions:** Four functions are used, each of which takes three 32-bit inputs and outputs a single 32-bit value. The functions consist of slightly different bit-wise operations and are designed in such a way that if the three inputs are independent and unbiased then the output will also be independent and unbiased.

**Process the Message:** M is processed in sub blocks of 512 bits using the four functions to modify the four blocks of the MD buffer, in a total of four rounds with each round consisting of sixteen steps. Once all of the bits in the message have been used, the four 32-bit blocks of the buffer become the 128-bit hash.

Despite the widespread use of MD5 it has been deemed unacceptable for use as a collision resistant hash algorithm [RFC6151]. In 2006, “Tunnels in Hash Functions: MD5 Collisions Within a Minute” was published describing how an MD5 collision could be found in less than a minute on a standard notebook PC. While the messages have to meet certain criteria, the paper exposes severe flaws in the algorithm.

While MD6 appears to be cryptographically secure, the designers stated that it should not be accepted in the search for a new standard secure hash algorithm (SHA-3), for which it was originally developed [The MD6 Hash Algorithm]. This is because it is not provably secure for the number of rounds required to meet the speed criteria of the competition. (It is provably secure for the number of rounds in its initial design.) According to their website, there are no effective attacks known, yet “an absence of evidence of weakness is no evidence of absence of weakness.”
2.6.2 Secure Hash Algorithm

The SHA family of algorithms is the standard designed, maintained and certified by NIST. There are four sub groups that have been adopted, SHA-0 (1993), SHA-1 (1995), SHA-2 (2001), SHA-3 (announced 2014, but yet to be published as a standard). Both SHA-2 and SHA-3 have different variations allowing for different lengths of the computed hash.

The Secure Hash Standard (SHS) was first announced with the publication of SHA (which became known as SHA-0) in FIPS 180. It was originally designed for use with digital signatures in the public key infrastructure and allowed the hash of a message (generally much shorter than the message) to be signed instead of the entire message, to improve speed and efficiency.

Both SHA-0 and SHA-1 borrow heavily from the MD4 and MD5 algorithms. They use the exact same method for appending padding and the length of the message. They use similar functions, though slightly different bitwise operations, and a different number of rounds (sixty four for MD and eighty for SHA). They both process the message in 512-bit blocks. However, both MD4 and MD5 produce a 128-bit digest, while SHA-0 and SHA-1 produce a 160-bit digest. This is because the MD buffer consists of four 32-bit blocks while SHA-0 and SHA-1 use five 32-bit blocks.

In 2001, NIST updated the SHS to include SHA-2, with the publication of FIPS 180-2. There are three variations within SHA-2; SHA-256, SHA-384 and SHA-512. SHA-256 produces a 256-bit digest, SHA-384 produces a 384-bit digest while SHA-512 produces a 512-bit digest. SHA-256 can compute a hash for any input message with length less than $2^{64}$ bits, while SAH-384 and SHA-512 can compute a hash for any message with length less than $2^{128}$ bits.

The operations of the SHA-256 algorithm, as described in FIPS 180-2, are:

**Append and Pad the Message:** This follows the same procedure as MD5. The message of length $l$ is extended by adding a bit with a value of 1, followed by $k$ bits with a value of 0, such that:

$$(l + 1 + k) \mod 512 = 448$$
for the smallest possible value of k.

**Append Length:** This follows an almost identical procedure to MD5, a 64-bit representation of the length is appended to the padded message. The exception occurs when the message length is more than $2^{64}$ bits, a SHA-256 digest cannot be computed for the message.

**Initialise the SHA Buffer:** The SHA-256 buffer consists of eight blocks 32-bits long. Constant values, the most significant bits of the fractal parts of the square root of the first eight prime numbers, are moved into these blocks.

\[
\begin{align*}
H_0^0 &= \sqrt{2} \mod 1 = 0x6A09E667 \\
H_1^0 &= \sqrt{3} \mod 1 = 0xBB67AE85 \\
H_2^0 &= \sqrt{5} \mod 1 = 0x3C6EF372 \\
H_3^0 &= \sqrt{7} \mod 1 = 0xA54FF53A \\
H_4^0 &= \sqrt{11} \mod 1 = 0x510E527F \\
H_5^0 &= \sqrt{13} \mod 1 = 0x9B05688C \\
H_6^0 &= \sqrt{17} \mod 1 = 0x1F83D9AB \\
H_7^0 &= \sqrt{19} \mod 1 = 0x5BE0CD19
\end{align*}
\]

The initial values in the hash buffer ($H^0$) should be unrelated. Choosing the prime fractals rather than precomputed random values appears to be an attempt to ensure that random looking numbers with known vulnerabilities have not been selected. This allows the public to be confident in the initial values, particularly given the NSA’s involvement in the SHS.

**Define Constants:** SHA-256 makes use of sixty-four 32-bit constants. These are used in the processing of the message, and are set to the fractal part of the cube root of the first 64 prime numbers, similar to the hash buffer.

**Define Functions:** SHA-256 uses a total of six functions. Two consist of bit-wise operations, two consist of rotations and two consist of rotations and bit shifts.
The first four are used when updating the SHA buffer, while the final two are used to compute the message schedule.

**Prepare the Message Schedule:** Unlike MD5, the SHA-256 buffer is updated by performing operations on a message schedule, rather than the message itself. The message schedule is processed in blocks of 512 bits (64 bytes and is computed from the message. The first 16 bytes of the message are copied directly into the first 16 bytes of the message schedule. The remaining 48 bytes are calculated using two of the functions and the defined constants.

**Process the Message Schedule:** The message schedule is processed in sub blocks of 512 bits using the four functions to modify the eight blocks of the buffer. After processing each 512-bit block of the message schedule, the eight blocks of the buffer are concatenated to produce the hash.

### 2.7 Message Authentication Codes (MAC)

A MAC is used to provide integrity and authentication for a message. It requires the communicating parties to agree on a key in advance. The input to a MAC function is this key and the message. Unlike a one-way hash function, which would allow nefarious parties to change the content of a message and recomputed the hash, only a user with the key can generate a valid MAC.

A simple MAC can be generated by using a one-way hash function, by first calculating the hash of the message and then encrypting the hash.

RFC2104 defines an unofficial standard for MACs, called a hash based MAC or HMAC. It requires a secure cryptographic hash function for the underlying calculation. The formula for computing a HMAC is:

\[
H(K \ XOR \ opad, H(K \ XOR \ ipad, M))
\]

Where H is a chosen secure hash function (e.g. SHA-256), opad is the byte 0x5C (repeated for the length of the key), ipad is the byte 0x36 (repeated for the length of the message) and M is the message (plaintext or ciphertext). The length of the output is determined by the underlying hash function.
The HMAC format is “proven to be secure if the underlying hash function has some reasonable cryptographic strengths” [Keying Hash Functions for Message Authentication].

### 2.8 Digital Signatures
Like a handwritten signature, a digital signature is used to prove to third parties that a message originated from a source [Applied Cryptography]. They are used to provide non-repudiation. While there are a number of different protocols for digital signatures, they are commonly used with public key cryptography. This allows one user to compute a hash of a message and then encrypt the hash with their private key. Anyone can verify where the message originated by decrypting the hash, using the signers public key, then recalculating the hash using the message.

### 2.9 Key Derivation
Key derivation is the process by which a key can be generated from a source of initial key material, such as a shared secret or password. Key derivation methods use either a single step approach or a two step approach, which consists of randomness extraction and key expansion.

Note, a key distribution function will often use additional data to bind the key material to the contextual transaction [NIST SP 800-56Ar2]. This may include but is not limited to, public information about the parties, such as an identifier, an indication of the protocol or application for which the keys will be used, a label or session identifier. The reason this is included is to ensure that keying material is never used in more than one protocol, even if static public keys are used. For the discussion on key derivation, the presence of this data will be assumed.

#### 2.9.1 Single Step Key Derivation
This involves converting initial key material, Z, into a key of a required length, as described in [NIST SP 800-56Ar2]. This is done with a key distribution function
(KDF), which uses either an approved hash function (from SHS) or a HMAC with an underlying approved hash function.

\[ Key = KDF(Z, \text{key length}, \text{salt}) \]

A KDF will repeatedly hash \( Z \), along with a counter and the salt (if applicable), producing blocks of data equal to the length of the output of the chosen hash function, until the required key length has been reached. The blocks are then concatenated. For instance, to compute a 512-bit key using SHA-256 as a hash function (rather than a HMAC), would require two iterations.

\[
K_1 = H(0x00000001 || Z || \text{salt}) \\
K_2 = H(0x00000002 || Z || \text{salt}) \\
Key = K_1 || K_2
\]

In the event that the required key length is not an integer multiple of the digest length of the chosen hash function (e.g. SHA-1 digest is 160 bits being used to generate a 256-bit AES key), the concatenation only uses the required least significant bits of the digest from the final repetition.

### 2.9.2 Extraction then Expansion Key Derivation

This involves two steps to derive a key from a the initial key material [NIST SP 800-56C].

#### 2.9.2.1 Randomness Extraction

This step involves converting the key material into a fixed length pseudo-random key derivation key (\( K_{DK} \)), using a MAC function. Both HMAC and AES-CMAC are appropriate for use as the MAC function. Regardless, the inputs are the initial key material, \( Z \), and the salt.

\[ K_{DK} = MAC(\text{salt}, Z) \]

After \( K_{DK} \) is calculated, \( Z \) should be deleted immediately. The length of \( K_{DK} \) is dependent on the output of the chosen MAC function, and should be at least as long as the target security strength of the algorithm for which the key is being
generated, i.e. using a HMAC with underlying hash function SHA-1 to produce a 160-bit $K_{DK}$ would be insufficient to generate a 256-bit AES key.

2.9.2.2 Key Expansion

The $K_{DK}$ is used to produce key material ($K_M$) of length $L$ bits. This is done iterating calls to a pseudo-random function (PRF), then concatenating the outputs until $L$ bits have been generated [NIST SP 800-108]. A PRF is a function that takes two inputs, $x$ and $y$, and computes an output that is indistinguishable from a randomly generated number (of equal length) if the value of $x$ is kept secret. The MAC function from the previous step can be used as the PRF. There are three variations of key expansion, called modes. Each uses a different iteration variable to generate different block of $K_M$.

- **Counter Mode:** For each iteration, the parameters passed to the PRF are the $K_{DK}$ and the iteration count concatenated with salt.

- **Feedback Mode:** In this mode, each call to the PRF passes the $K_{DK}$ and the result of the previous PRF iteration concatenated with salt. An initialisation vector is required to generate the first block of key material. This can be secret or public, empty, random or predefined.

- **Double Pipelining:** This mode uses an initial call to the PRF, which is concatenated with salt and passed to the PRF again along with the $K_{DK}$ to produce each block of key material. An initialisation vector is also needed for this mode.

2.10 Attacks

2.10.1 Types of Break:

A cipher is considered broken if a weakness can be found and exploited in such a way that requires less complexity than a brute force attack [Applied Cryptography]. If a brute force attack requires $2^{256}$ key attempts and an exploit allows this to be calculated using $2^{240}$ attempts, the cipher is still classified as broken, even though this may still be computationally infeasible.
We can classify attacks based on the amount of information available to the attacker.

2.10.2 Ciphertext-Only Attack:
This is technique involves attempting to determine the key used to encrypt a message through analysis of the ciphertext [RFC4949]. It is possible that other information is known about the plaintext (e.g. language, subject, etc.) and/or the algorithm used.

A brute force attack involves trying every possible key to decrypt the ciphertext. The result is then analysed in an effort to determine the whether it is meaningful and if it is the original plaintext, in which case the key has been found.

\[
f_d(C, K') = M'
\]

If the decryption function is known \(f_d\) then trying all of the possible keys, \(K\), will result in all of the possible messages \(M'\). Choosing a sufficiently large key will make this computationally infeasible. For example, if a key is 128 bits long, then it will take on average \(2^{127} \times (2^{128}/2)\) guesses in order to find the key (assuming the correct plaintext can be matched from the resulting attempts).

To put this in perspective, it would take the world’s most powerful computer (the Tiahne-2 supercomputer) approximately \(1.6 \times 10^{17}\) years to find the key [calculations based on “How secure is AES against brute force attacks?”, with updated values for world’s fastest supercomputer]. This is number is astronomically large (considering the universe is approximately \(10^{10}\) years old).

Using a sufficiently large key space makes brute forcing completely infeasible given the current technology. As a result, attacks generally focus on weaker aspects of encryption.

2.10.3 Known-Plaintext Attack:
This involves attempting to determine the key when an attacker has both the plaintext message and the corresponding ciphertext.
2.10.4 Chosen-Plaintext Attack:
This involves attempting to determine the key when an attacker can choose the plaintext of the message to be encrypted and see the ciphertext.

2.10.5 Chosen-Ciphertext Attack:
This involves attempting to determine the key when an attacker can choose the ciphertext of the message to be decrypted and see the plaintext.

2.10.6 Rubber-hose Cryptanalysis:
By far the cheapest and simplest way to determine an encryption key, this technique involves threats, blackmail, bribery, social engineering or a combination of all. The weakest link in computationally secure cryptography is the human user.

While attacking well-designed ciphers may be computationally infeasible, there are many attacks on protocols that can be more successful.

2.10.7 Man in the Middle
A man in the middle attack involves a nefarious party who is capable of intercepting and modifying messages exchanged between two users are unaware of the attacker [Applied Cryptography]. The attacker impersonates both parties and can either choose decrypt and read all of the messages or modify them. The dangers of a man in the middle attack can be overcome by guaranteeing authentication (for example, using digital signatures as part of a public key infrastructure).

2.10.8 Replay Attacks
A replay attack involves the attacker recording old encrypted messages [Applied Cryptography]. These are then resend by the attacker. The attacker may have some knowledge of the meaning of the ciphertext without being able to decrypt it. The danger of this attack can be overcome using timestamps, to ensure that messages are not old and to ensure that each ciphertext message is unique.

2.10.9 Hash Attacks
While it is computationally infeasible to calculate the original message from the hash computed by a secure one-way hash function, it is possible to find values
that produce the hash. Rainbow tables contain precomputed hash values which can be searched to provide a corresponding input. They can be downloaded from the web for common hashing algorithms (such as MD5, SHA-256, etc.). They consist of a message and a corresponding hash. One simply enters the hash in an attempt to discover the original message. In general they are used to crack passwords that are hashed without salt. Using any salt, public or secret, overcomes this danger.

2.11 Inspirational Systems
There are a number of other systems that have influenced the design. This is a brief overview of some of them.

2.11.1 Pretty Good Privacy (PGP)
PGP is a cryptographic system designed to provide integrity, confidentiality, authentication and non-repudiation. It uses both public key and conventional cryptography to accomplish this [RFC4880]. It can be used on a variety of formats, from mail messages to data files. It will briefly be discussed with regard to two parties, Alice and Bob.

Alice wishes to send a message (or other electronic information) to Bob. She computes a digest of the message using a chosen hash function. This hash is encrypted with her public key. Next she generates a random key (for this message only) to be used with a symmetric cipher. She encrypts the message and prepended digest with this random key. This random key is then encrypted with Bob’s public key and prepended to the message. Alice then sends this to Bob. Bob begins by decrypting the random key using his private key. He will then use this key to decrypt the message and prepended digest. Bob now has the plaintext message. To verify that it came from Alice, he decrypts the digest using Alice’s public key and then computes a digest of the plaintext message using the chosen hash function. If the digests match, then Bob is sure that the message came from Alice and that it was not altered.
Note that messages need not be both encrypted and digitally signed. They can be one or the other.

### 2.11.2 Off-the-record Messaging

In the paper, “Off-the-Record Communication, or, Why Not To Use PGP”, the authors compare the properties of real-world social communications to the four components of cryptography; authentication, integrity, non-repudiation and confidentiality. When two parties, Alice and Bob are communicating in person, they can guarantee authentication, integrity and confidentiality. They cannot ensure non-repudiation. It is impossible for Alice to prove to a third party that Bob said something specific (and vice versa).

The off-the-record messaging system designed in the paper offers:

- Perfect forward secrecy
- Confidentiality
- Authentication
- Integrity
- Repudiation

Diffie-Hellman key agreement is used to establish a shared secret, which is used to generate a 128-bit AES key. The Diffie-Hellman public keys are signed using the parties private key (as part of a public key infrastructure), in order to authenticate each user.

Messages are encrypted using the AES cipher in counter mode (where AES is used as a stream cipher). This allows a nefarious party to alter the ciphertext so as to affect the plaintext in a meaningful way (as each bit of ciphertext directly corresponds to a bit of plaintext). The purpose of using a deliberately imperfect cipher is to make it obvious that anyone could have modified or sent the ciphertext.

A MAC is calculated for each message (using a different symmetric key to the encryption key). This provides authentication and integrity for each message. Once every message has been authenticated, the MAC key is published. Neither Alice nor Bob needs the key to verify messages (as this has already been done),
while it allows anyone to construct valid MACs for arbitrary messages. The authors claim that this means anyone can be a potential author of a message.

A Diffie-Hellman public key is sent with each message, which is used to generate a new symmetric key for the next message. This provides perfect forward secrecy, as in the event that one key is compromised, no past or future messages can be read, nor can any other keys be determined. The symmetric keys are deleted after all of the messages encrypted with the key have been read.

2.11.2.1 Discussion

The system does indeed offer repudiation in accordance with face to face social interactions. However, I disagree with the claim that publishing the MAC keys allows anyone to be a potential author of any previous message. Only someone with the secret MAC key at the time a message was sent could possibly have generated the valid MAC. I also have reservations about the use of AES in counter mode. The MAC attached to each message ensures both authentication and integrity (to anyone with the secret MAC key). If the message is altered in any way, both Alice and Bob would be aware of it. Once the MAC keys are published, any third party can calculate if a previous message was modified. In the event that is was not, the message must have been sent by Alice or Bob (assuming the keys had not been compromised).

Finally, it is the responsibility of both Alice and Bob to delete their symmetric keys after all messages encrypted with it have been read. While this does mirror an social interaction, either party could choose not to delete their key and would be able to decrypt and reread previous messages.

2.11.3 FireChat

FireChat is an open source communication app available for both Android and iPhone smart phones. It was introduced in March 2014. It is an “off the grid” instant messaging application, which does not provide any encryption or authentication. It allows users to chat without Wi-Fi or mobile network coverage, by using Bluetooth instead. It has gained popularity in countries where internet access is being restricted. Protestors, in countries like China and Iraq, use it to
communicate and organise. Each phone running the application acts as a node as part of a mesh network. Messages are then exchanged with nearby users.
3 Design:

3.1 What the System Does
The system is designed to allow two parties to exchange encrypted messages. The ciphertext is posted to a publicly accessible location, where anyone can access it. The DeadDrop name originates from this aspect, where spies would place information in plain sight at designated drop spots which would later be gathered by the intended recipient. Only the sender and receiver knew where to look for the message. While the original dead drop refers to steganography (the hiding of plaintext messages), this system uses cryptography to ensure that messages posted in public locations cannot be deciphered in a feasible amount of time by third parties.

The system is designed to provide confidentiality, using a symmetric cipher, message integrity and authentication, using a MAC. The authentication provided is not based on any identity per say, but on the users who share the symmetric key. The actual identity of both parties may be unknown. Non-repudiation is not provided for. The system is only designed for two communicating parties, so it should be easy for them to determine the sender and recipient of a message, however, this cannot be proved to a third party. This feature mimics that offered by “Off-the-Record Messaging.” In this instance, each encrypted message must have originated from one of the two parties, or the key has been compromised.

3.2 Generating A Secret
In order for users to encrypt data, they must generate secret keys in advance. This is done using the Diffie-Hellman protocol. As a result of the security this protocol offers, the communication can take place over an insecure channel. Two users who wish to exchange messages will begin by each generating a long random number. This can be considered their private key and must be kept secret. They calculate their public key using the formula:

\[ K_{\text{public}} = g^{K_{\text{private}}} \mod p \]
where $K_{private}$ is the random number, $g$ is the generator and $p$ is the prime. These values should be chosen from the appropriate group in RFC3526, which specifies good candidates for $g$ and $p$ depending on the desired security strength. The Diffie-Hellman protocol should not be the weakest link in the system. RFC3526 also specifies approximate equivalent lengths for $p$ in relation to AES key lengths. The public key is then exchanged over the agreed channel. A shared secret is generated by both parties using the formula:

$$S = (K_{public})^{K_{private}} \mod p$$

Where $K_{public}$ is the public key they received and $K_{private}$ is their private key.

If we assume that an eavesdropper has been listening on the insecure channel, they will see both Diffie-Hellman public keys. From this, it is infeasible to calculate the shared secret as this would involve calculating one of the private keys from the equation:

$$K_{public} = g^{K_{private}} \mod p$$

As outlined previously (in section 2.5.1) breaking Diffie-Hellman is as hard as solving the discrete log problem (when weak values of the generator and the prime are not used, and the values can be known to the attacker).

The design does not provide authentication between users. It is the responsibility of the users to ensure that they are communicating with the person they intend on communicating with. As a result, this Diffie-Hellman key exchange is susceptible to both man in the middle and impersonation attacks.

A nefarious party could respond to any Diffie-Hellman public key with their own Diffie-Hellman public key, and generate the shared secret with the legitimate user as both the generator and prime are publicly known. The lack of authentication means that a legitimate user cannot differentiate between an attacker and another legitimate user. Obviously a user unknowingly communicating with an attacker is completely unacceptable, however a successful man in the middle attack would be worse in that it would allow an attacker to either eavesdrop on a conversation between the two intended users.
(in a passive attack) or to simultaneously impersonate either or both (active attack).

![Diffie-Hellman Man in the Middle Exchange – Image 2](image)

In this instance, our man in the middle would intercept Alice and Bob’s public keys, calculate a private key and derive a public key of their own to send to both Alice and Bob. Alice would calculate a shared secret, as described previously, using the attacker’s public key, mistakenly assuming it was Bob’s public key. Bob would also calculate a shared secret using the attacker’s public key, thinking it was Alice’s public key. The attacker would calculate a shared secret for both Alice and Bob, using their public keys and his private key. Neither Alice nor Bob would be aware that they are talking to the attacker. From the shared secret a symmetric key is generated. The attacker will have a key for communicating with Alice and a separate key for communicating with Bob. Depending on their motivation, the attacker may simply decrypt all of Alice’s messages using their symmetric key, encrypt them with the symmetric key shared with Bob, then forward them to him and vice versa. Alternatively, the attacker could alter or create messages, or just prevent any communication between Bob and Alice.

As the system does not provide authentication, it is the responsibility of the users or implementers to ensure that they are communicating with the intended parties. There a number of possible solutions to this problem. For instance, Diffie-Hellman Authenticated Key Agreement outlined in RFC5246, makes use of the public key infrastructure to digitally sign messages in order for users to ascertain their origin and prevent any would-be attackers from modifying messages without the users becoming aware. Unfortunately, this requires the
additional overhead of a trusted third party and asks the question why users would choose to use Diffie-Hellman over existing public key cryptography to generate symmetric keys. Other possibilities include doing the Diffie-Hellman exchange over email (if addresses can be confirmed in advance). While the system does not provide authentication to the users, the implementation does overcome this problem and will be discussed later.

3.3 Generating A Key
The result of the Diffie-Hellman exchange (assuming the users have avoided any attacks) is a shared secret. This must be used to generate a symmetric key, which will be used for both encrypting and decrypting messages. This can be done using a key derivation process as outlined in section 2.9.

3.4 Message Format
The message format consists of an optional header, the ciphertext and a footer. Both the header and the footer are computed using a hash function, and their length is dependent on the chosen function. If SHA-256 was used, both the header and the footer would be 256 bits long.

<table>
<thead>
<tr>
<th>256 bits</th>
<th>Variable</th>
<th>256 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Ciphertext</td>
<td>Footer</td>
</tr>
</tbody>
</table>

3.4.1 Message Header
Each user should have a 32-bit identifier associated with them. This is a random number which should be kept secret. Each user should also generate a random salt for each user they wish to communicate with. This is used with their identifier to compute a hash. This hash is exchanged with other users who wish to communicate. If a user does not change their salt for each user with whom they wish to communicate, other users will be able to determine of they are communicating with the same user.
The message header is used for parties to easily determine if a message is meant for them. Both parties should have sufficient information to determine from the header:

- If the message was sent to them
  - And if so, who sent it
- If the message was sent by them
  - And if so, who it was sent to

To all other parties, the message header should be meaningless. It should not be possible to determine the identifier of either party, nor should it be possible to track or predict any previous or future correspondences. This can be accomplished using the sender’s id, the receiver’s id, the shared encryption key and some salt, with neither of the legitimate parties able to determine the id of the other.

At some point, possibly during the initial Diffie-Hellman secret generation, each party should exchange additional information to be used in the header. Rather than sending their id, each user should compute a hash of their id and the randomly generated salt. This can be sent over an insecure channel as an eavesdropper will not be able to determine the original ids, nor will they be able to use this information to impersonate either party (assuming the eavesdropper cannot modify the content on the communication channel). The parties may also choose to exchange some initial salt to use in calculating the first headers, though a zero byte value is sufficient.

The header for each message is calculated using the HMAC function (as described in section 2.7 and in RFC2104). The inputs are the shared symmetric key, k, the received hash of the other users unknown id and salt, the sent hash of the users id and salt and some additional salt. The header is computed using:

\[ H(K \text{ XOR } \text{ opad}, H(K \text{ XOR } \text{ ipad}, M)) \]

Where H is a chosen secure hash function, opad is the byte 0x5C (repeated for the length of K), ipad is the byte 0x36 (repeated for the length of M) and M is a concatenation of the sender’s hash, the receiver’s hash and the salt. In order for
messages not to be linked, the value of the salt must change for each header. This can be done by incrementing the value of the initial salt each time a header is used. Alternatively, salt to be used in the next header could be sent with each message.

The purpose of the header is to reduce unnecessary decryption. Only two legitimate users exchanging messages can compute the correct header. This can be done in advance. When looking for messages, each user need only search for messages with the correct header. This presents nefarious parties from being able to post many messages that a party will need to decrypt. It is important to note that once a header has been posted, there is nothing to prevent third parties from posting fake messages reusing the previous header. For this reason, the first message posted that contains a legitimate header should be the valid message.

3.4.2 Ciphertext
The message itself must be encrypted. The current date and time, in addition to some optional salt (for use in the next header) is appended to the plaintext. This is encrypted using the shared symmetric key and chosen cipher.

The purpose of the timestamp is to prevent replay attacks, where a nefarious party could post an old message unbeknownst to a legitimate user. While the use of the header could also prevent this, it is optional. The timestamp must be in a format independent of time zones, such as UTC date and time.

The purpose of the salt is to allow the users to change the header of the next message without simply incrementing the initial value. This is optional.

3.4.3 Message Footer
The message footer contains a MAC (as outlined in section 2.7). This should provide both integrity and authentication. A MAC could be calculated by computing a hash of the ciphertext, then encrypting the hash. However, it is recommended to use the HMAC. This guarantees that no information about the inputs, the ciphertext or the key, is revealed to any third parties once the underlying hash function is secure.
If a header is included, then it too should be included in the MAC calculation. While a header will only be meaningful to the intended parties, a previous header could be combined with a different message. The timestamp would prevent a user from thinking this is a valid message, but it would still require extra computation in order to decrypt the message before checking if it was valid.

3.5 Sending a Message

Two users, Alice and Bob, share a symmetric key and Alice wishes to send a message to Bob. If she wishes to include a header. It should be calculated using the HMAC function. The salt is either the salt included in the previous message, the salt that was initially exchanged, a sequence of zeros or an incremental value based on the last salt used. The order of the sender’s and receiver’s id is important as it is used to determine who sent the message. In this case, as Alice is sending the message, she uses her id first. This simplifies searching for messages.

Alice then enters her plaintext message. The ciphertext is this message with the current time and date and some (optional) salt for use in the next header encrypted with the symmetric key, using the chosen symmetric cipher. This information is then posted to a public location.

The sender and receiver must agree on a location in advance. This could be anywhere on the internet, including but not limited to a public forum, the drafts folder of an agreed email address, a Twitter feed, the comments section of a news article, etc. The location must have the following properties:

- Messages cannot be deleted without both parties knowledge
- An identifying account (such as a college email address) must not be required to post or view messages
- A timestamp for each message posted

The communicating parties may decide to reuse the same location. Alternatively, they could include the details of another location in any of their message. Before the first message is sent, they must have an agreed initial location. This can also
be exchanged in the initial Diffie-Hellman key agreement and does not need to be encrypted.

3.6 Receiving a Message

In order to receive a message, the location of the message must be known in advance. If a header is being used, the receiving party should calculate this in advance. This would be calculated using the HMAC formula, where the inputs are the key, the stored value of the hash of the sender's id and salt, the hash of their id and salt and the salt from the previous message, the zero byte value or an increment of the previous salt. The location could then be searched for this value. Note that each party should store the salt used for the next header, in case a message in the conversation is deleted. If no header is used, the receiver should search for ciphertext. Depending on the location used, it may be obvious where this is.

Once a ciphertext message has been found, a MAC for the message must be computed and compared to the appended MAC. If these values match, then the receiver can be assured that the message was composed by someone with the secret key and that the message has not been altered.

The message is then decrypted using the secret key and the chosen symmetric cipher. If salt is present, the receiver should store this value. If a new location is proposed, the receiver should remember it.

3.7 Storing Information

Each user must store the secret key, the hashed id of the other party, the salt they use to hash their id (or the precomputed hash of their id and the salt) and the salt required to calculate the next message, for each party they wish to communicate with. It is their responsibility to protect this information and ensure that no third party can access it. This could be done by storing it in an encrypted file.
3.8 Generating New Keys

Users should be able to generate new keys. This should be done using another Diffie-Hellman protocol to generate a shared secret and then compute a symmetric, secret key. The public keys can be posted in the same location as messages, and should include a HMAC to provide authentication and integrity. Neither party should randomly generate a new key, encrypt it with the old key and then send this to the other party. In the event that any previous keys were compromised, this would mean that all future keys would be compromised.
4 Implementation

The system that I designed was implemented on two Android devices, one running Android version 2.2 and one running Android version 2.3.4.

4.1 What I Implemented

The implementation allows two users to exchange encrypted messages from their Android phones. The ciphertext is posted to a publicly accessible internet forum, where anyone can access it. I set up a specific location for messages to be posted and retrieved. This is located at www.reddit.com/r/fyp_dead_drop. There are a couple of simplifications to the design that were present in the implementation.

4.2 Tools

Android Bluetooth Chat Application: I used a sample, open source, Android project called Bluetooth Chat. The functionality includes scanning for other devices, connecting two devices and asynchronously transferring data over a Bluetooth connection.

Spongy Castle: Android has a basic cryptographic library preinstalled on devices. This is based on the open source library called Bouncy Castle. Unfortunately, it is quite limited. Spongy Castle is a more thorough library based on the full Bouncy Castle library, rather than the default, reduced version on Android. This was used to perform most of the cryptographic functionality.

jReddit: In order to post and retrieve messages from the location, some code from an open source, Reddit, Android application was used. This made use of the Reddit APIs needed to interact with the site.

Base64: Base64 is an encoding scheme used to convert bits into ASCII data. There are sixty four characters used, each of which represents a 6-bit value. All values can be represented in this format, though padding may be required.
Android has a built-in Base64 encoding, which was used to store data and post messages.

### 4.3 Storing Information

The application uses two files in order to store data. When a user first launches the application, they are required to enter both a good password and a bad password. Some random salt is generated. The bad password and the salt are hashed (using the SHA-256 algorithm). The resulting 256-bit value is converted to a Base64 encoded string and written to a file along with the salt. The good password is also hashed with the salt. This value is not written to the file. This 256-bit value is used to encrypt another file.

Each instance of the application must store the following information for each contact:

1. Conversation identifier (for the application)
2. Contact name or pseudonym (for the user)
3. 256-bit AES key

If a malicious person were to obtain this information, they would easily be able to impersonate a user, as well as read all of the previous messages. For this reason, this information is stored in an encrypted file on the Android device. The file is encrypted using the hashed value of the good password and the salt. A digest for the plaintext is computed and appended to the ciphertext, all of which is written to a file.

In order to gain access to the application, a user must enter their password each time the application launches. This password is combined with salt (stored in the plaintext file) and hashed using the SHA256 algorithm. If this value matches the stored hash of the bad password, then the encrypted file and the plaintext file are deleted. Otherwise, the 256-bit value is used as an AES key to decrypt the contacts file. A hash of the decrypted file is calculated and if it matches the appended digest, then the user has successfully logged in.
4.4 Generating a Secret

This system implements the Diffie-Hellman protocol in order to establish the secret. The communication channel is an unsecure Bluetooth connection between both mobile phones. The values of g and p are chosen from the 2048-bit group in RFC3526, which specifies good candidates for g and p.

As well as exchanging public Diffie-Hellman keys and calculating a shared secret, the two users, Alice and Bob, also exchange their identifiers. Unlike the design, this is not hashed in order to keep it secret.

While the design does not guarantee authentication, the implementation using Bluetooth does. In order to exchange keys, both parties have to connect their respective devices. As part of the Android Bluetooth connection, both parties pair their devices. This requires scanning for available Bluetooth devices and then selecting the relevant device name. To prevent nefarious parties from attempting to hijack the connection by using a device of the same name, both parties generate a random code which they should confirm and enter in order to pair the devices. This is handled by the Android operating system.
The code used to establish the Bluetooth connection and exchange messages was based on an open source Android Bluetooth Chat application. This allowed two users to pair their devices and exchange messages asynchronously. I added the Diffie-Hellman protocol to generate a public key, using the Spongy Castle library. When keys are being exchanged, both Alice and Bob will generate their public key. They then click the sent button, and a notification will inform them if the key was sent successfully. A notification will tell them when they have received the other parties public key. When a party has successfully sent a key and received a key, the save button is enabled. They are also prompted to include a name for their new contact, though this can be an alias.

When the save button is clicked, the received public key, which is sent as a string, is converted into a big integer (a data type for storing large numbers). This is then converted into Spongy Castle's type Key, which along with the user's Diffie-Hellman private key is used to initialize the shared secret. This functionality is part of the SC library.

4.5 Generating a Key

Both Alice and Bob have successfully used the Diffie-Hellman protocol to generate a secret. This must be used to generate a symmetric key. The chosen cipher for the implementation is AES, with 256-bit keys. While the design recommends a more sophisticated two step key derivation, the implementation simply hashes the shared secret using the SHA-256 hashing algorithm. This is a method within the SC library. The result is that both Alice and Bob have a shared 256-bit key. The contact is then added to the contact list, which includes the contact name, the secret key and the contact's identifier.
4.6 Message Format

The message format used in the implementation differs from that outlined in the design.

<table>
<thead>
<tr>
<th>96 bits</th>
<th>Variable Length</th>
<th>288 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Ciphertext</td>
<td>Footer</td>
</tr>
</tbody>
</table>

The main difference is the header, which is not optional in the implementation. Both the ciphertext and the footer are also slightly simplified.

4.6.1 Message Header

The message is 96 bits long and is expressed in hexadecimal notation, though it is posted and retrieved as a string.

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Identifier</td>
<td>Sender ID</td>
<td>Receiver ID</td>
</tr>
</tbody>
</table>

It contains eight consecutive ones (0x1111111 – 4 bytes) indicting the beginning of the message. This is used in order to make finding, retrieving and processing messages easier. The next 4 bytes are a hexadecimal representation of the sender's identifier, while the remaining 4 bytes are a hexadecimal representation of the receiver's identifier. These are used to determine the sender and recipient.

From this, we can tell the identifier of the sender and the identifier of the receiver. Unlike the design, this header does reveal some information about the communicating parties. Anyone can track the messages between the two parties. Anyone can determine if either of the parties are communicating with other users. While this information does not reveal any information about the content of the messages, it could be used to determine the identity of the individuals, or the Android device from which the message was posted, by analysing the web traffic of the location (depending on the how the application accessed the internet).
4.6.2 Ciphertext

The current time in milliseconds since January 1, 1970 00:00:00.0 UTC is appended to the plaintext message. This format is independent of time zones and is calculated by the Android operating system. The plaintext message and time are encrypted using the AES cipher with the shared secret key. Unlike the design, the optional salt used in the message to generate new headers is not included. The maximum length of the message and the time is limited by the hashing algorithm used to compute the footer.

4.6.3 Message Footer

The message footer is a 256-bit MAC and a sequence of eight consecutive zeros (0x00000000). Rather than use the HMAC formula as recommended in the design, the implantation computes a hash of the ciphertext and then encrypts that with the shared symmetric key. The hashing algorithm used is SHA-256. The eight consecutive zeros are used to tell the application that the end of the message has been reached.

<table>
<thead>
<tr>
<th>256 bits</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>Application Identifier</td>
</tr>
</tbody>
</table>

The maximum length of the ciphertext is determined by this hash function. SHA-256 can only compute a digest if the length of the input is less than $2^{64}-1$ bits. This size is more than sufficient for all messages exchanged. It is approximately 2,305,843 Terabytes. This is far more than either of the Android devices is capable of managing, both in terms of storage and in terms of computational expense in calculating the hash.

4.7 Sending a Message

This will be discussed relative to two parties, Alice and Bob, who have generated a secret, symmetric, AES 256-bit key. They have also used their each other’s first names when they added each other as contacts. In this instance, Alice wishes to send Bob a message.
Alice opens up the application on her Android device. She logins into the application, which gives her access to the encrypted contacts. She selects the name “Bob” from the drop down list of contacts. Bob's name, identifier and their shared key are now stored in a global object.

![Screenshot of Alice's Login Page – Image 4](image)

Alice then clicks the “Post Ciphertext” button. From there she enters the plaintext message. Once entered, she clicks the post button. The plaintext message entered is stored in a global variable as a string. A request is made to the Reddit website for permission to post a message, using the API’s as previously described. The website will respond with a CAPTCHA which is then displayed on the screen of the Android device. Alice enters the correct characters in the CAPTCHA and then clicks the “Encrypt & Post” button.

The current contact object is Bob. As Alice is sending the message, the header will be composed of the application identifier, her id, followed by Bob’s id. All of these values are 32-bit hexadecimal numbers, however, they are processed as strings. The header is a concatenation of all of them.
The current time (in milliseconds since January 1, 1970 00:00:00.0 UTC) is returned as a long integer by the Android operating system. This is converted to a string and appended to the plaintext message string.

The plaintext message and time is converted into a byte array based on the character encoding UTF-8 (default Android encoding). This byte array is then encrypted using the 256-bit AES key. This is performed using the SC library. The result is a byte array of ciphertext. AES processes data in 128-bit blocks. If the data is not divisible a multiple of 128 bits, it is padded in order to generate the output. The ciphertext byte array is then converted to a base64 encoded string.

The footer is calculated by computing a digest of the ciphertext. This is done by first converting the base64 encoded ciphertext into a byte array. This is the input to the SHA-256 algorithm, implemented by the SC library. The output is a byte array containing a digest of the ciphertext. This is then encrypted with the AES 256-bit key and then converted to a base64 encoded string. The hexadecimal number 0x00000000 (represented by a string) is then appended to the MAC, to complete the footer.

The footer is appended to the ciphertext, which is appended to the header. The result is the encrypted message in the required format. This is then posted to the public location (www.reddit.com/r/fyp_dead_drop) using the Reddit APIs from an account (not linked to any email address) which I set up (anon_dead_drop). All posts use this account, which has the username and password hardcoded in the application. A confirmation is received by the phone.
4.8 Receiving a Message

This will be discussed relative to two parties, Alice and Bob. In this instance, Bob wishes to view all messages between himself and Alice. Bob opens up the application on his Android device. He logs into the application, which gives him access to the encrypted contacts. He selects the name “Alice” from the drop down list of contacts. Alice’s name, identifier and their shared key are now stored in a global object. Bob then clicks the “View Messages” button.

![Screenshot of Bob's Login Page – Image 5](image)

This gets a list of submissions to the public location ([www.reddit.com/r/fyp_dead_drop](http://www.reddit.com/r/fyp_dead_drop)) filtered by the account username (anon_dead_drop). This query is performed using the Reddit APIs. The information returned from the query includes the entire message (note that a + sign is part of the base64 encoding characters, however the Reddit API removes this and replaces it with a space, as a result, each space between the header and the footer is replaced with a + symbol) and the time it was posted at (in seconds since January 1, 1970 00:00:00.0 UTC). A maximum of 500 messages can be
returned, though these may not be for the intended parties. The application searches the retrieved submissions based on a number of parameters. Each submission must contain:

1. The hardcoded character sequence in the header that the system uses to identify submissions it has made (11111111)
2. The hardcoded character sequence in the footer that the system uses to signify the end of the message (00000000)
3. Alice's identifier
4. Bob's identifier

For each post where the parameters are present, the initial sequence of ones and the final sequence of zeros is removed. Next, the MAC of the ciphertext is computed. This is done by hashing the ciphertext and then encrypting it with the symmetric key. If the computed MAC is equal to the appended MAC, then Bob can be certain that the message was composed by someone with the shared key and was not changed by a third party. If the MACs do not match, the message is ignored. Next, the ciphertext is decrypted using the 256-bit AES key. The time stamp is removed and compared with the time the message was posted at (retrieved via the API call).

To prevent replay attacks, the time stamp makes each ciphertext unique, as even identical plaintext messages will have a different time stamp. Therefore, if identical messages are found, only the first message shall be considered valid.

Finally, all messages are prepended with the sender's name. In this case, message from Bob will contain the prefix “Me: ”, while messages from Alice will contain the prefix “Alice: ”. This is done based on the order in which the identifiers appear in the header.
4.9 Generating New Keys

The implementation should allow two communicating parties to generate new keys using the same Diffie-Hellman key agreement protocol. It should be possible for the Diffie-Hellman public keys to be posted to the public location, however, this has not been featured in the application. Instead, users would have to delete each other from their contacts and add each other again. This would involve meeting in person and generating the shared secret over Bluetooth again. This is impractical and inconvenient. It would also mean that all messages encrypted with the previous AES key would be indecipherable. If a key was compromised, messages encrypted with different keys would remain secret though.
5 Conclusion

The implementation achieves the goal of allowing two users to communicate anonymously while being sure that they are talking to their intended target without the requirement of knowing their identity. The design is much broader and leaves this feature up to each implementation. Unfortunately, the implementation does reveal information in the form of the identifiers of each user. As explained, this can be used to track conversations and communicating parties. The design outlines how revealing this information can be avoided.

Neither the implementation nor the design specify how a signal could be used to tell the other party that a drop has been made. This could be agreed by the users, though is entirely optional. For instance, if two users were friends on Facebook, then something as discreet as liking an old photo could be used as the signal (though this would obviously mean that they would need to use alternative profiles or be aware of each other’s identity).

Another aspect not considered in either the design or the implementation, is how a user connects to the internet. Ideally, all of their traffic would be anonymised using public Wi-Fi, a VPN or something like the Tor network. This would prevent traffic analysis from being able to determine the identity of the user based on the ip address of the device used to post messages.

The Android application is actually relatively functional, despite some of the shortcomings and primitive user interface. The use of 256-bit AES keys to both encrypt messages and sensitive contact details (including secret keys) is computationally infeasible to break. The best relevant attack against AES 256-bit keys reduces the complexity of calculating the key from $2^{256}$ (for a brute force attempt) to $2^{254.4}$ keys after $2^{80}$ chosen-plaintexts have been encrypted [Algorithms, key size and parameters report – 2014]. While this classifies as a theoretical break, it is practically impossible given the current and likely future resources. Note that there are slightly better attacks, however these require related 256-bit keys, which would be highly unlikely given the use of the Diffie-Hellman protocol and the key derivation process used in both the implementation and the design.
The Diffie-Hellman key agreement on the other hand, does not provide as high a level of security as the AES encryption. The implementation uses the prime and generator from the 2048-bit mod group as specified in RFC3526. This is approximately equivalent to brute forcing a key between 110 bits and 160 bits long. Similarly, "Algorithms, key size and parameters report – 2014" recommends that new systems use a minimum prime of length 3072 bits. This was an oversight that I missed when selecting the values of the prime. The design does specify that the Diffie-Hellman key agreement should not be the weakest link. This means that the implementation should have used a prime of at least 8192 bits long, which is estimated as having equivalent strength as a key between 190 bits and 310 bits long [RFC3526].

If I were to continue developing this application, I would first improve the user interface. At the moment, it is very rudimentary. I would develop a colour scheme, perhaps similar to Image 3. I would also ensure that there are no bugs in the application. At the moment, a special character is used to separate the values for each contact in the encrypted file. If a user were to use this character when naming a new contact, the stored key would not be read into the right variable and would likely cause the application to crash. Depending on how the file was saved, it may be permanently corrupted. Obviously, this would be unacceptable to a legitimate user, but could be fixed by either preventing the use of the special character or by changing how the encrypted file is read. Next, I would publish the code to an online, public repository, such as GitHub. From there, it could be developed or criticised by the open source community, which I think is an important aspect of designing a secure cryptographic system.
6 Bibliography

6.1 Academic


Diffie-Hellman is as Strong as Discrete Log for Certain Primes, Bert den Boer, 1990

FIPS 180-0, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-0 (11 May 1993) by National Institute of Standards and Technology


FIPS 197, Announcing the advanced encryption standard (AES), Federal Information Processing Standards (FIPS), Publication 197 (November 2001) by, National Institute of Standards and Technology

FIPS 46, Data Encryption Standard (DES), Federal Information Processing Standard (FIPS), Publication 46 (January 1977) by National Institute of Standards and Technology

FIPS 46-1, Data Encryption Standard (DES), Federal Information Processing Standard (FIPS), Publication 46-1 (January 1988) by National Institute of Standards and Technology

FIPS 46-2, Data Encryption Standard (DES), Federal Information Processing Standard (FIPS), Publication 46-2 (23 December 1993) by National Institute of Standards and Technology
FIPS 46-3, Data Encryption Standard (DES); specifies the use of Triple DES, Federal Information Processing Standard (FIPS), Publication 46-3 (25 October 1999) by National Institute of Standards and Technology


Keying Hash Functions for Message Authentication, Mihir Bellare, Ran Canetti, Hugo Krawczyk, June 1996


NIST SP 800-108, Recommendation for Key Derivation Using Pseudorandom Functions (Revised), Special Publication 800-108 (October 2009) by National Institute of Standards and Technology

NIST SP 800-56C, Recommendation for Key Derivation through Extraction-then-Expansion, Special Publication 800-56C (November 2011) by National Institute of Standards and Technology

NIST SP 800-56Ar2, Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography Revision 2, Special Publication 800-56Ar2 (May 2013) by National Institute of Standards and Technology

Off-the-Record Communication, or, Why Not To Use PGP (October 2004), by Nikita Borisov, Ian Goldberg, Eric Brewer


Tunnels in Hash Functions: MD5 Collisions Within a Minute, Vlastimil Klima, April 2006

6.2 Images

Image 1 – from FIPS 197

Image 2 – retrieved April 21, 2015, http://upload.wikimedia.org/wikipedia/commons/2/2a/Attaque_Man_In_The_Middle.jpg and edited

Image 3 – designed in Photoshop

Image 4 – screenshot from application running on Android device

Image 5 – screenshot from application running on Android device

Image 6 – screenshot from application running on Android device

6.3 Open Source Code

jReddit – source code available on GitHub, https://github.com/karan/jReddit
