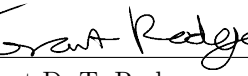
Submitted for the Degree of
B.Sc (Hons) in Computer Science 2020

Indistinguishable Steganographic Communication

Grant Rodgers (201607489)

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

I agree to this material being made available in whole or in part to benefit the education of future students.

Signature: 

Grant D. T. Rodgers

Date: *April 5, 2020*

0 Preface

0.1 Acknowledgements

I would like to thank my supervisor Shishir Nagaraja and his PhD students for their support and valuable insights. I would also like to thank my friends Iain and Teodora and my mother, for providing me with the strength to continue and the feedback to make this project a success.

I would also like to thank coffee, the finest organic suspension ever devised, for getting me through the last 8 years of academia.

0.2 Abstract

The aim of this project is to create a steganographic system that is tag scheme and compression standard independent. Research was conducted into the background and history of steganography, along with papers detailing existing systems. During this research, a feature of MPEG file formats was discovered that allows the exploitation of gaps of empty data present in many of them. A proof of concept program was developed that verified that 97.31% of files surveyed contained gaps larger than 1000 bytes, which is enough room to hide a short message. Another proof of concept program was then created to take advantage of these gaps and was successfully demonstrated to function at concealing and retrieving messages on a wide variety of sound files. The embedded data was demonstrated to survive transmission through messaging applications, cloud storage and general purpose file compression but did not survive re-encoding of any type or re-recording. In light of this, statistics were gathered to judge and compare the usability of various file metrics, specifically extension, bitrate, file size, bandwidth and duration, and analysis found that M4A files appear to have the largest gaps and most bandwidth available for this type of steganographic operation.

Contents

1 Introduction

MPEG steganography schemes currently under research rely on either modification before compression [1], or specific tagging scheme/compression standards being used [2]. This either severely restricts the number of files a given person can use to send messages or requires extensive re-encoding to allow for that specific scheme to work with a particular file.

The main aim of this investigation is to create a steganographic system that is independent of compression standard or tag scheme while taking advantage of blank spaces in the file after compression. This system should allow a given end user to guarantee that they will be able to take any given MPEG file, perform steganography on it and ensure that a given message may be sent and received in a manner unobservable by casual and global adversaries without suffering degradation or perceptible changes to the carrier file. It must also satisfy the Kerchoff principle [3] stating that, even if a hypothetical attacker were to gain full knowledge of the steganographic scheme and its exact implementation, they would still be unable to decode the message without the required key.

This report gives an overview of the environment surrounding encryption and the need for steganography. It also includes a hypothesis verification (GapChecker), the system itself, and a discussion of the strengths and weaknesses of it.

2 Background

Many countries in the world, developed and otherwise, implement censorship and control over the flow of information and news. This can range from simple matters of market driven political and social pressure influencing decisions made by individual publishing houses [4], [5], to wholesale suppression of anti-establishment sentiments, sensitive or embarrassing events, or any content that is deemed "obscene" by the governing party. This can be enforced with routine monitoring, and blocking platforms that refuse to capitulate provides incentive.

Even press environments deemed "free" by the Press Freedom Index and similar [6] are not immune to political pressure, and may be in fact be more sensitive to it. Publishing articles that criticise the government can invite scrutiny or pressure from said government. Their readership are also likely to have varying political views, and are likely to criticise what they disagree with or point out any inaccuracies, whether perceived or otherwise.

Media independence is seldom an all or nothing affair. Only rarely does a government implement a totalitarian approach on media, only allowing it to function as an arm of the governing party. Some governments implement variations to the boundaries of what is acceptable reporting depending on current social tensions and/or current anti-establishment sentiments [7].

There is also the matter of governments and intelligence services classifying and suppressing information that they consider damaging to national security or to the national interest and having mechanisms (such as the Official Secrets

Act) to criminalise those individuals who leak protected information to the wider world. In these environments, exchanging information without censure, oversight or censorship becomes more difficult and consequently more important, as communication without censure or general oversight is a fundamental freedom that is hard to find on an Internet largely governed by private companies [8].

2.1 General Protection

Using any communication network will always carry privacy risks and leave some kind of trace. In the case of the Internet, packets cross several pieces of infrastructure en route to their destination maintained by multiple entities which all have the opportunity to intercept the information being conveyed. Additionally, there is no way to reliably predict where your traffic is likely to be routed beyond the initial hop from your computer to your ISP due to the highly decentralised and meshed topology of the infrastructure. Not only can the data contained within the packet be sensitive, but the fact that a packet happened to pass a certain point at a certain time (traffic analysis) can also be leveraged to obtain information (e.g. an email was sent from this computer at this time) [9].

Widely implemented tools to protect packets in flight across untrusted networks (HTTPS encryption, for example), stop legitimate middlemen (caching and intrusion detection systems) from performing their tasks. This is generally enforced on sites that carry sensitive information but can be extended to include every website that supports it through various means (most commonly through browser extensions).

This can be a desirable trait to prevent these middlemen from gaining incidental insight into your browsing habits or undesirable, as it will increase bandwidth overhead. Caching can be employed at various levels of network topology: at the institution level to reduce the number of external requests made to the wider Internet by storing a version of the page on the internal cache, or at the "edge" of the network, physically or topographically nearer to the end user than the resource itself (for example, in an ISP data centre), reducing latency.

Some versions of intrusion detection work by monitoring the content of packets to detect the signature of protocol violations or direct attacks. This type of detection only works if the packets are directly "sniffable" (i.e. not encrypted).

2.2 Simple Anti-Censorship Mechanisms

Some simple mechanisms have been used in the past in attempts to mitigate against political or legal pressure brought against individual organisations or groups of publishers or news outlets, leading to suppression within a governing party's jurisdiction. WikiLeaks employs a "dead man's switch" system, which publicly and widely distributes an encrypted set of files in order to prevent a single point of takedown then releasing the decryption key automatically in response to some event (after a set time or a period of inactivity, or after an inciting event such as capture, death or takedown of the source) [10].

This allows a censorship resistant mechanism for distributing sensitive data, achieved by concealing the content of data until is until it has become too widespread to fully censor. This mechanism, however, does not disguise that a communication has taken place, nor does it allow for one to one communication.

2.3 Encrypted Messaging

The most widely available mechanism available to anyone who wishes to communicate in a manner that is not visible to outside observers is an encrypted messaging system. There are various methods and technologies available to facilitate this, all of which have their strengths and weaknesses.

The United States government lent support to the Escrowed Encryption Standard (EES) [11], an encryption scheme developed by the National Security Agency (NSA). In response to this, an open source protocol known as Pretty Good Privacy (PGP) was developed.

Under the EES system, the NSA certifies that any implementation of this technology will be unbreakable for several years in exchange for a copy of the key, allowing the NSA to intercept any messages sent by the user of that system. This raised constitutional concerns in relation to free speech and right to protection from unreasonable search and seizure [12]. It also came with the profound but unprovable risk that the NSA retains the ability to crack the closed source encryption protocol.

PGP, by contrast, is an open source standard, created in 1991 and defined (as OpenPGP by the IETF) in 1997 to allow end-to-end encrypted email [13], and transmission integrity verification. The standard has been implemented in many systems since then, including the GNU Privacy Guard system (GPG), and remains in use today under multiple supported implementations. It provided efficient (for the time) public key encryption and converts a message into an ASCII "armoured" (i.e. encoded in plain text) ciphertext that can be copied and pasted into any email client with no extra modifications required. It is also considered, with modern versions of PGP, to be "virtually impossible" for any law enforcement or intelligence agency to break into PGP encrypted content without obtaining the key through legal coercion or espionage, as the computational power required to derive the very large prime numbers involved is currently infeasible.

This, however, requires the user to have an understanding of cryptography and public-private key infrastructure [14], [15]. It is also considered relatively dated by modern cryptography standards and requires manual intervention and verification of keys while providing no forward security [16]. Forward security ensures the safety of any past usages of a private key regardless of whether it is compromised in the future by employing session or one time keys.

Standards such as PGP were designed for high-latency asynchronous operation. Messaging protocols such as eXtensible Messaging and Presence Protocol (XMPP) [17] or Internet Relay Chat (IRC) [18] were designed to facilitate real time low-latency communication between two or more parties and be open source standards alongside PGP.

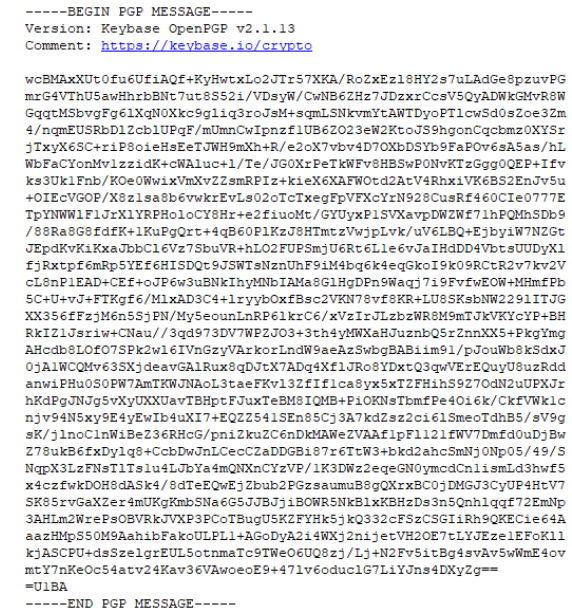


Figure 1: Example of a PGP encrypted message

IRC is a protocol that evolved from the Bulletin Board Service standard, that allows for groups of two or more people to exchange messages in "channels", (areas of congregation centred around a particular topic) [19], assuming an alias to give a veil of anonymity. It also provides powerful tools to allow private messaging, ignoring of messages from a particular user by another user, and the sending of invitations to join other channels, along with methods for administrators of servers or channels to ban or restrict access. Whilst messages intended for one particular user are possible and channels can be set up in order to require a key or password to join, the protocol does not provide methods of encryption and message security. This standard has fallen out of popularity with the rise of social media, but efforts to support and expand the protocol continue [20].

XMPP, a protocol that allows for asynchronous, end-to-end exchange of XML (eXtensible Markup Language) streams, is considered a well adopted standard as it has a globally distributed and addressable network of clients and servers. It does not support any special encryption as part of its standard, but has network security protocols (Simple Authentication and Security Layer and Transport Layer Security) built in to encrypt data in flight. As with IRC however, the service providers are still able to easily read the plaintext of the user's messages.

Off The Record (OTR) [21], [22] was one of the first security protocols, designed as an improvement over PGP and an extension to XMPP that allows forward security to be achieved as it does not have any long-term public keys to be compromised. It also provides deniable authentication, unlike the very public and verifiable communication and identification methods that underpin PGP. Perfect forward secrecy is achieved through the Diffie-Hellman key agreement [23], allowing the two communicating parties to agree on a shared secret to use as a one-time/session key without revealing it over a potentially insecure communication medium. This approach to key freshness became known as "ratcheting".

The Diffie-Hellman key agreement relies on the fact that that it is computationally infeasible to extract the generated secret from the information exchanged over the insecure channel. In order to ensure perfect forward secrecy is achieved, however, previous keys have to be forgotten after use, which mandates synchronous operation in order to prevent vulnerability windows exposed in asynchronous operation. These vulnerabilities take the form of the uncertainty over deleting previous keys that may be associated with messages in transit and the ability for any actor to steal those keys from an end user's machine and decrypt previous messages.

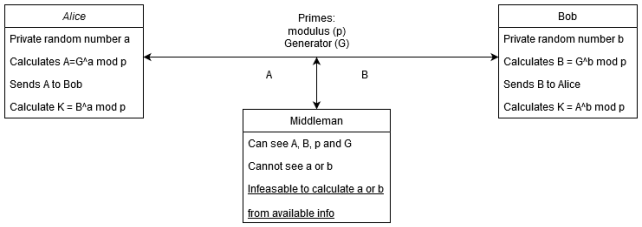


Figure 2: Illustration of Diffie-Hellman Key Exchange

In the defined client facing implementation of OTR, when an initial message is sent between two parties it is unencrypted, but contains an appended identifier to alert the receiving party that the sending party supports OTR. Then, during the key exchange process, the user is notified that secure communication is about to take place and the sender's public key fingerprint is displayed, allowing for out-of-band verification and reducing, but not entirely eliminating man-in-the-middle vulnerabilities as it still relies on proactive action from the user.

Deniable authentication is provided by message authentication codes (MACs), a hash function (HMAC [24] being a popular one) derived from a shared "MAC key", the result of which is exchanged alongside the message in the secure transmission. The MAC key is sent via the same mechanism as the session key, so only the end user has access to it. The receiving party computes the MAC using the exchanged key and compares it to the transmitted one, allowing a verifiable integrity and authentication mechanism, guaranteed to be private and deniable.

In response to the Snowden revelations in 2014 several companies began implementing their own end-to-end encrypted messaging services, such as Face-

book and WhatsApp [25]. Although they may be based on established or open source exchange and encryption protocols (such as variations of the Signal [26] protocol suite, implemented in the Signal app), since they are closed source the service itself is not standardised or decentralised, so anyone who wishes to use these protocols can only communicate with other users of that platform [27]. The nature of the closed source ecosystem means that it is difficult to analyse the security of the platform, but these systems provide an unprecedented level of accessibility to end-to-end encryption since keys are automatically generated and encryption is automatically or easily enabled. The Signal protocol leverages the Diffie-Hellman key agreement asynchronous ratcheting, and introduces synchronous ratcheting, applying a symmetric key derivation function to create a new key without incorporating fresh DH material. This approach allows the Signal suite to work in both synchronous and asynchronous environments.

Group messaging is also possible with these applications as they work by storing and forwarding messages from the always-on servers to group members as they come online. Group messages in the Signal protocol are treated identically to direct messages, so the managing servers cannot distinguish between them. The message is end-to-end encrypted for each group member and forwarded to each member with the group ID attached in the encrypted payload.

Administration messages for joining and leaving groups and acknowledgment messages for both detecting that sent messages were received by the recipient's client and read by that person are not end-to-end encrypted. This means they can easily be forged to disrupt the acknowledgement of messages (e.g. sending a forged read receipt when the recipient's client hasn't even received the message).

Given this flaw, all an attacker needs to "burgle" a group is find the group ID and the phone number of any member (preferably the group admin, who is most likely to send update messages). With this information they can send a forged group update message through the direct messaging channel between themselves and this group member who after receiving and validating this update message, will change their group description to include the attacker and begin sending their messages to the attacker. Once the targeted member sends another administration message to the group, every member will update their descriptions and, consequently, begin sending messages to the attacker.

2.4 Steganography

All the methods for ensuring message security detailed above have one fatal flaw: even if an observer cannot decipher the message's content, they can tell you are using an encrypted messaging platform through the structure of payloads sent and their destination. This may introduce scrutiny or censure in locations where such services are restricted, treated with suspicion or outright banned.

Digital encryption has been sought by the public and industry to protect secrets [28] for as long as it has existed. Many countries have wished to control all encrypted traffic in their jurisdiction or prevent its use entirely. This is officially meant to ensure that law enforcement or state security is able to access any content they wish to obtain (with varying degrees of judicial oversight) in a

timely and straightforward manner. This is usually justified under the guise of preventing terrorism and allowing various crimes to be easily investigated, thus allegedly discouraging them from being perpetrated [29], [30].

In environments where encryption technologies must either be state owned and patented, or the owners of said technology must "cooperate" with law enforcement or state security efforts to decrypt users' messages sent through their platform [31], [32] with varying degrees of punishment/censure for non-compliance, pressure for companies to either capitulate to demands (sacrificing their users' informational liberties), or make it functionally impossible for them to do so increases.

In order for these agencies to ban, restrict or access messages sent using encryption techniques, they must first be able to determine that they are in use, an effort which has been made for as long as steganographic messages perceived as innocuous to an outside observer have been in use [33].

Any medium that can convey information can be modified to carry secret information in an inconspicuous manner, with the most common methods in use on social media today involving manipulation of image, sound, document and video mediums, in varying forms.

2.4.1 Image Steganography

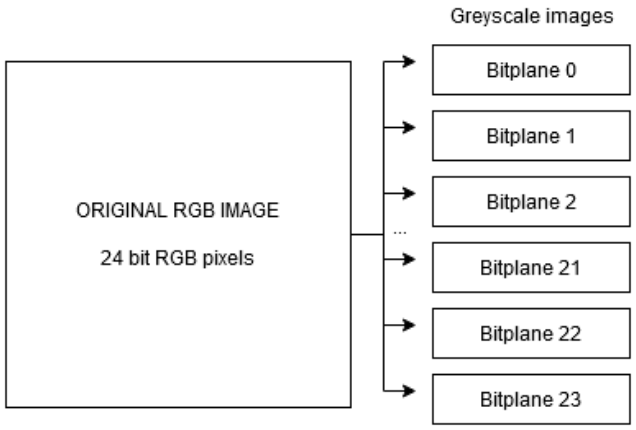


Figure 3: Bit Plane Complexity Segmentation

One of the simplest forms of steganography for digital images is Least Significant Bit manipulation (LSB) [34]. This involves taking the least significant bit of each pixel value (or each Red, Green and Blue value for each pixel) and modifying it to correspond with the bit of the information to be hidden. In its simplest form it is not very robust to any modification, nor is it resistant to detection [35] as sequential pixel values are likely to be inconsistent in a noticeable way when examined, especially in pictures which are meant to have consistent

consecutive colour values. The overall usage must be significantly lower than the potential available bandwidth to avoid tampering artefacts being visible in the image. Pseudorandom distribution of the altered pixels throughout the image can significantly reduce the chance of detection by eliminating the noticeable image banding common to linear LSB techniques [36].

Another, more complex method that is significantly less likely to introduce artefacts is Bit Plane Complexity Segmentation (BPCS) [37], which divides the carrier image into 24 black and white images. The "complexity" (i.e. how many adjacent changes between black and white are made along the whole plane, with a black and white checkerboard pattern being the optimum complexity) of the plane is calculated, and the one with the largest complexity is selected in order to allow larger amounts of data to be hidden without introducing any significant artefacts or apparent changes to the human eye. When this technique is combined with encryption algorithms such as AES, the information becomes computationally infeasible to retrieve even if detected without the decryption key. These methods combined with other compression methods such as EZW compression optimised for lossy image formats such as JPEG [38] allow for large amounts of data to be hidden relative to the size of the file.

Given JPEG is the most commonly used format for online image exchange, many steganography schemes have been developed for use on them. These can be broadly separated into two categories: spatial domain algorithms directly operate on the pixels themselves, while frequency domain algorithms (the most common method) exploit the JPEG compression methodology to imperceptibly hide data. During the JPEG compression process, 8x8 blocks in the image are converted using an algorithm based on Discrete Cosine Transform (DCT) [39] which converts these blocks from a spatial domain into a frequency domain.

The resolution of the chroma data is reduced, exploiting the fact that the human eye is less sensitive to sharp high-frequency brightness changes compared to smaller variations over a larger area [40]. The quality setting of the encoder directly affects the reduction in resolution of each component and, if a sufficiently low quality setting is used, high-frequency components will end up being entirely discarded instead of being stored less accurately. The resulting data from these blocks is then further compressed using a variant of Huffman encoding.

Early approaches for JPEG steganography involved hiding data in the LSB of the quantised DCT coefficients, avoiding ones where the value is 0 to prevent distortions being visible in the image. These approaches attempt to match the original DCT histogram as closely as possible in order to resist steganalysis. The JSteg and JHide algorithms [41], [42] apply this methodology using sequential and pseudorandom coefficients respectively.

Steganalysis schemes have been able to detect these methods and others that use similar techniques as some image features are modified by these techniques that are difficult to preserve. Blind steganalysis [43], [44] algorithms tend to perform self-calibration by removing a few pixel rows or columns then recompressing the image in order to desynchronise the original JPEG grid from the one used to embed the information. This allows detection of six common

steganography algorithms with around a 95% accuracy even at low embedding rates.

Yet Another Steganography Scheme (YASS) [45] was proposed as a method to resist blind steganalysis. This is achieved by embedding data in 8x8 blocks chosen randomly so as not to coincide with the 8x8 JPEG compression grid, normally performed by dividing the image into blocks larger than 8x8, then pseudorandomly selecting an 8x8 grid from within that, desynchronising the parameters computed by the analyst. The scheme also avoids modifying coefficients that quantise to zero and employs error correcting codes to provide protection against multiple active attacks (including recompression and additive noise). It cannot, however, escape detection from active steganographic techniques such as pixel and DCT coefficients dependency analysis [46].

2.4.2 Sound Steganography

MP3 steganography has been a relatively widely researched field [2], as it is one of the most common audio compression standards in use, in which data can be embedded either during or after compression.

Some techniques employed for image steganography can also be applied to sound steganography [47]. The most common use case for audio steganography is watermarking [48] in which the process of embedding a watermarking image or string into the sound file assists in the identification and tracking of copyright breaches.

As with image steganography, the simplest method of sound steganography involves substituting the LSB of each sample (the value of the sound waveform at a particular instant) in the file with the data bits to be hidden [49]. This has similar benefits to image LSB encoding, namely computational simplicity and a large embedding capacity. It also has similar problems to image LSB substitution, namely trivial discovery and noticeable distortion to the audible data.

Spread spectrum steganography involves spreading the encoding of the hidden data across as much of the frequency spectrum as possible [50]. This is a highly robust technique, but has similar noise introduction problems along with a vulnerability to time scale modifications.

An improvement to both of these techniques involves combining LSB coding with sub-band coding and shift register based scrambling [51]. The audio is divided into 32 equal width sub-bands, and the bands that are considered to have the lowest signal power (generally the highest sub-bands) are used to hide the data. Shift register based scrambling is employed to allow the data to be random enough so that it does not introduce harsh audio characteristics that are unpleasant to the ear. Encryption can also be employed, as with other techniques, to prevent recovery of the message even if the steganography is discovered.

Echo hiding steganography increases robustness and transfer rate compared to LSB and does not introduce any perceptible noise to the playback. Sound recorded naturally has both the original sound and one or more time delayed

echoes of that sound, the characteristics of which are dependent on the recording setup and the room the sound was recorded in. The hidden data can be added through embedded echoes with different offsets, one to represent binary one and another to represent binary zero. This process is designed to survive lossy MPEG compression but has a low embedding capacity alongside a low robustness to signal degradation (generally through re-recording).

Steganography after compression is generally less robust than steganography before compression, as relying on characteristics of the storage medium to embed information is inherently vulnerable to any re-encoding or degradation. Several techniques can be employed to leverage the underlying representation structures.

MP3 frame headers contain multiple fields (such as the original, private, copyright and emphasis bits) that are generally ignored by most players. These fields can be manipulated to embed steganographic data relatively easily, but with easy discoverability.

MP3 frame sizes also rarely contain equal amounts of data, so in some encoding schemes, padding bytes are used to keep their sizes consistent. One byte per frame can be used to embed data using a computationally simple algorithm, giving a relatively high capacity.

Alongside specific issues pertaining to each after-compression technique, they all rely upon quirks in specific formatting schemes or the MP3 formatting scheme as a whole. Although the MP3 format is widely used, relying upon it as your primary transmission method is still limiting, especially when multiple other widely used and open source formats are in use.

3 Related Work

The BlindSpot system was investigated as a basis of understanding of work already undertaken, as was StegoBot. A small experiment (GapChecker) was also performed to determine whether gaps exist in MPEG files with different sample rates, schemes and encoding.

3.1 StegoBot

This paper sets out a botnet system which communicates over probabilistically unobservable communication channels. It builds on the weaknesses of traditional botnet systems, such as centralised command and control creating a single point of failure and scalability chokepoint, and that communications between the bot and the controllers are observable. It also introduces JPEG steganography as the mode of communication through a given social network (in this case Facebook).

This paper gives an overview of the threat model the system is designed to combat, the steganographic technique proposed, and the design, construction and testing of the given components.

The threat model is assumed to be a global passive adversary, i.e. someone who has a global view of all traffic going in and out of a given node and who is likely to cooperate with other global observers (e.g. ISPs, enterprises,

governments etc.). YASS [45] is selected as the best scheme to use for hiding the payload, as it was proven to be most resistant to image processing by Facebook with a few caveats to minimise interference, such as resizing images to the maximum resolution allowed (720x720 px) and converting it to the JPEG format.

3.1.1 Design

The deployment of the botnet is achieved through the exploitation of the same social links the system will use to communicate once deployed. Genuine emails and attachments, stolen from a given connection and modified to carry the malicious payload, are considered one of the best social malware distribution methods [52].

This method can be deployed to recruit a reasonably sized botnet in a relatively short period of time, especially when the volume of traffic passing between links is high (such as in an enterprise). The bot itself can contain a pre-programmed list of actions to perform. If it is to be used for malicious purposes, this could include keylogging or targeted harvesting of credentials or financial information. Alternatively, it can wait to receive commands from the command and control channel.

Two message types are handled by the bot: bot-commands, which are broadcast messages from the Botmaster that can include search terms or commands to delete itself, and Bot-cargo, which is information either generated by the local client or forwarded through the botnet on a multihop route to its destination. Information is only transferred through steganographic channels. This, combined with limits on payload size, can ensure indistinguishability from normal communication when examining the communication channel alone [53].

3.1.2 Results

The viability of the transmission technique was tested by uploading a series of images embedded with YASS encoded data to a given Facebook account. The images were then downloaded from another Facebook account and decoded. Increasing the amount of data stored in the image (controlled in the YASS algorithm by the value Q) naturally increased the error rate, while increasing the redundancy decreased the error rate. The optimal value which minimises the error rate for the given compression/transmission medium varies based on the medium of transmission used and its compression schemes. The viability of the network itself was also examined in detail using real-world social graph data. With a naive routing algorithm, the Botmaster saw around 10% of information sent but, with a slightly more advanced algorithm that exploits high-volume nodes along with smarter sending and receiving through hubs to reduce congestion, the delivery rate and throughput were significantly increased.

3.1.3 Commentary

This experiment reinforces the need for both a message encryption-decryption system and the algorithm to manage distribution between nodes. The YASS algorithm, under reasonable constraints, has been proven to be viable for use in the given scenario with minimal errors.

This experiment presumes transmission from many nodes to a single point of contact (the Botmaster), however, it may be viable to employ a "many to many" approach, or, in the case where a single command and control point is still required, a "round robin" passing of responsibility to prevent a single point of failure.

The system also assumes social engineering being used to unwillingly co-opt nodes into participating in the network. While this may be desirable for malicious purposes, some persons may voluntarily participate in such a network for the purposes of routing messages (if no malicious code is contained in the system, of course).

3.2 Blindspot

This paper seeks to investigate methods of providing practical anonymous communications through means indistinguishable from normal communications. This avoids viewable chains of communication [54] between parties and prevents observation of message content; information which can be used against them to justify actions such as increased surveillance, censoring or sanctions.

This paper sets out and addresses issues with two overarching methodologies: resisting surveillance by emulating innocuous protocols [54] and by using anonymous communication networks [55]. With emulation the system can never, by design, fully emulate the protocol it seeks to mimic as it may introduce unusual changes to the standard protocol. With anonymous communication networks, however, it does not hide traffic volumes or protect against traffic confirmation attacks (injecting load patterns to determine communication endpoints) and cannot hide whether a user is using an anonymous channel or not - a problem when the very act of using anonymous communication may invite sanction.

It poses the following questions: How might a user's given social network be leveraged to send messages outwith it in a way that does not leave direct evidence of communication and that cannot be read by intermediaries? How might said network perform this communication in a way that is (relatively) low latency and, with a given secret communication, indistinguishable from normal communication? The paper seeks to investigate these questions by designing a system to solve the problems with existing methodologies.

3.2.1 Results

Each user encrypts their messages with a public-private encryption pair, exchanged out of band. Each node in the system also generates a second encryption key pair to be shared with all its neighbours [56], to allow delivery only

directly to that node for forwarding purposes. The core of the system is a decentralised distribution algorithm. Messages are routed between a sender and receiver through intermediaries without revealing either identity to any node other than the exit node, which then delivers to the receiver. The delivery network relies on a pull-broadcast model to prevent traffic analysis and confirmation attacks. In this system, each node maintains an input and output queue of messages to be uploaded/distributed when the user next makes a post to their social media. This message is embedded into the innocuous delivery medium (such as an image or video for sharing through a social network) using already established algorithms such as YASS [45] for JPEG steganography, with suitable levels of redundancy to resist compression by the social media platform.

The routing algorithm was tested through a simulation built from the social graph and monthly uploading behaviour of 7200 Flickr users and was proven to maintain delivery rates and relatively high speed in the face of active attacks and loads, such as congestion and black holing (where nodes in the system are taken offline, either deliberately or accidentally). In congestion scenarios, the simulated system maintained a delivery rate above 85%. In black holing scenarios, in which around 50% of key nodes (that provide crucial steps between a given set of nodes) were removed, there was a drop to a rate of 52% in the Flickr model. When random nodes were removed the delivery rate was slightly higher at 56%, however the delivery time remained unchanged at approximately 1 day. The system was also tested under a Barabasi-Albert model (with the Flickr users and social graphs integrated as a 1:1 mapping) to simulate a more closed network such as Facebook. Under congestion this performed similarly to the Flickr model but, under the black holing attacks, the network appeared more resilient to removal of random nodes, even under a random degree attack.

3.2.2 Commentary

This simulated system as it stands is more than feasible to justify the real-world construction of both a message encryption-decryption system and the algorithm to manage distribution between nodes. The simulation itself appears to be well constructed since it effectively emulates real user behaviour by drawing from real social graphs. A usability study would appear to be the primary goal of this working prototype both in a technical and user/client sense. This would lay the groundwork for a fully functional system, preferably in the form of a browser plugin, which handles message encryption and the routing automatically. Some further questions to investigate would include whether the YASS algorithm [45] would be the most suitable steganographic technique for this solution, whether or not the best algorithm depends on which social network is to be used, and what method is best for acquiring the users' public keys.

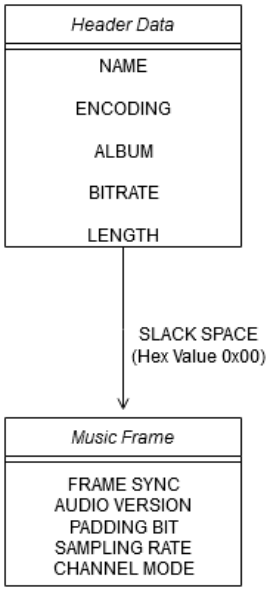


Figure 4: Illustration of the gap between Header Data and Frame Data

4 GapChecker

A hypothesis was proposed involving tagging and compression standard agnostic encoding based on the observation that the unused space seemingly present in some MPEG files could be used to store encrypted data. In order to verify this hypothesis two questions had to be answered: whether or not every MPEG file (regardless of age, quality or tagging scheme) possesses a blank space larger than a kilobyte (1000 bytes), and whether or not any modification to this area can withstand transmission or recompression.

A program was devised [57] to check a large number of MPEG files in a consistent way (hereafter referred to as GapChecker). GapChecker was written in Java and will, when given a particular file path, walk through all folders and subfolders on that path searching for files that fit a given parameter (in this example, MP3, M4A and OGG files).

It will open that file as a byte array and scan through it looking for the hex value "00". If it finds this value, it will note its position and start counting the consecutive empty bytes. Once a non-zero value is encountered, the end point is counted and, if the difference between these two points is greater than 1000 bytes, it is reported as a gap to the summariser.

Once the walking is complete, an average gap size is calculated and a summary is presented detailing the average gap size, the number of valid files seen which contain gaps, and the total number of gaps seen.

4.1 Design

4.1.1 Program

This class creates an instance of the FileWalker class, sets the directory to start the file walker from, then calls the walkFileTree method of the Files class with the start directory and the instance of FileWalker. Once the walking is complete, the Program class calls the summariseStats method of the instance of GapChecker within the instance of FileWalker.

4.1.2 FileWalker

This class creates an instance of the GapChecker class then extends and overrides the SimpleFileVisitor Java class, specifically the visitFile and visitFileFailed methods. If the file is a "regular file" (i.e. not a directory) and, if it is an MP3, M4A or OGG file, it calls the getStats method of the GapChecker module. It also prints the size of the file in bytes regardless of whether it is a music file or not.

4.1.3 GapChecker

The GapChecker class attempts to obtain basic stats about a given set of files passed to it. It stores data about the size of every gap in the files it parses then summarises them to obtain average sizes, total numbers of gaps, valid files seen within the directory and how many of these valid files contain gaps larger than 1000 bytes. This value was chosen based on the size of a small unencrypted text file containing a short message being around a kilobyte in size on average.

The getStats method opens the passed in file as a byte array then increments a counter of valid files seen, as it is assumed that if the file path has been passed to GapChecker then it is a valid music file. For each byte in this array, it will check to see if this byte is equal to "00" (a null value). It will then proceed to count the number of consecutive null values until it reaches a non-null value. If the number of consecutive null values is greater than 1000, it will print stats about the gap (the number of consecutive empty bytes, the start point and the end point) and add that value to an ArrayList of gap sizes. The module will then increment a counter of files seen with gaps. This is only incremented once, regardless of how many gaps are present, as this increment is located out of the for loop iterating through each byte.

The summariseStats method calculates the metrics mentioned above and writes them out to a file named "averageAll.txt". It provides some basic error checking, as it will only proceed if the number of gaps found (i.e. the size of the gapSizes array) is not zero.

5 GapFiller

This program was devised to take advantage of the tagging and compression scheme agnostic gaps found to be available in a large number of MPEG files by the GapChecker test program. It seeks to provide a reasonable user the ability to encrypt short messages using existing public/private key distribution systems, embed them in a steganographic target, then send said target through any existing social media link. This program assumes the end user has some knowledge of command line interfaces, Python IDEs, Ubuntu and public-private key architecture. For receiving messages as well as signing sent messages, the user is also required to sign up for a Keybase account, and either create a new keypair through the Keybase system or upload a locally generated key to their profile.

5.1 Deployment Requirements

The recommended way to run this program is through the PyCharm IDE, but any IDE that has Python integration will potentially be suitable, and the virtual environment requires the following packages to be installed:

- os
- pathlib
- gnupg
- time

The program also requires you to run your program in an environment that has GnuPG accessible via the command line. Ubuntu and most flavours of Linux have GnuPG available from the command line. In order to allow access to your keyring, the program requires initialisation through the GnuPG module to gain permission to write keys into your GnuPG home directory. For the testing modules, extra libraries are required:

- mutagen
- soundfile
- math

5.2 Architecture

- GapFiller - Controller class. Gives command line interface to user and facilitates the use of underlying components
- GnuPG - Wrapper class for command line GnuPG operations
- GapChecker - Analyses files for gaps larger than the prescribed size
- GapFillerStego - Places the given ciphertext directly into the steganographic target

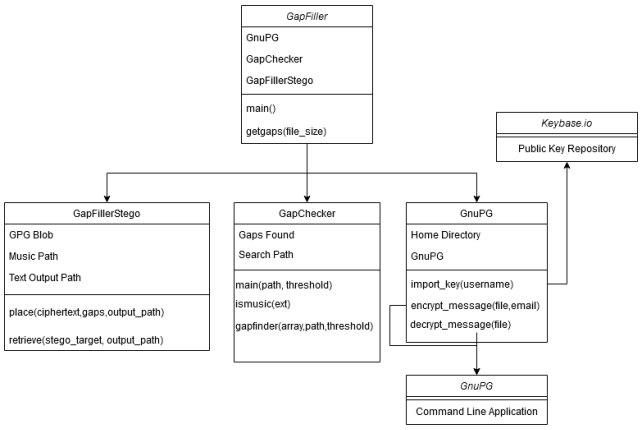


Figure 5: GapFiller Class Diagram

6 Implementation Specification

6.0.1 GapFiller

Provides an interface and controller for the rest of the components. The interface is command line based, and provides basic error checking for certain input lines, including the initial choice of encryption, decryption, or exiting. If the user chooses to encode a message, the module prompts for a Keybase username. This corresponds to the Keybase username of the person you wish to send the message to. This will be passed to the GnuPG module. It then asks for the person's email that corresponds to the one associated with their public key. This allows GnuPG to find the key after it has been downloaded and added to your keychain, as the GPG system was initially created to encrypt emails, and Keybase simply acts as a repository for these keys. It then asks for the relative path to the file you wish to encrypt, and finally asks for the directory in which you wish to place outputted files (with the option to leave it blank for the program directory).

The GapFiller module then finds the real (full) path to the file, calls the import_key module of GnuPG to import the key, then calls encrypt_message to encrypt and sign the file. It then calls the getGaps module of GapChecker to find suitable gaps corresponding to the size of the encrypted and signed GPG blob, then calls upon the place module of GapFillerStego to perform the steganographic operation. If the user chooses to decode a message, the module prompts for the relative path to the music file that is suspected to have a steganographic target embedded. The GnuPG module will find the real (full) path to the file, which is then passed to the retrieve method of the GapFillerStego module, then, if a GPG blob exists in the file, it will extract it, then pass the path of this new file to the decrypt_message module of GnuPG.

6.0.2 GnuPG

A wrapper class that shells out to the command line GnuPG application to handle encryption and signing, decryption and importing of keys taken from the Keybase application. This module was the reason why development shifted from Java for the GapChecker viability check, to Python for the final program. Java's methods for shelling out to the system's command line are needlessly complex, and available modules require asynchronous operation. Some require "joining" the process thread for a fixed amount of time, and most have no way to tail the standard out or standard error of your command without extra steps.

With the move to Python came a built in GPG library to directly encrypt, decrypt and import keys. This came with its own problems, however, as not only is it a completely separate system from your built-in keychain (this was discovered when attempting to import keys through the command line then use it through the GPG module), but it does not allow selection of keys with anything other than the key's fingerprint. So, Python's os.system() commands (which automatically tail stdout and stderr to the console and return a status code depending on whether the command threw any errors or not) were used to send commands to the GnuPG program installed with most Linux distributions and available through the OS command line. The "trust-always" flag is set to suppress untrusted key warnings, but as it is assumed that keys obtained from Keybase are trusted for the purpose of encrypting messages, it removes an extra step for an end user. The move to Python also came with it a dependency on Linux, as most Linux distributions have GnuPG by default, so the program only functions on Python under Linux as a whole, although certain components (GapChecker, for example) will work on other operating systems.

6.0.3 GapChecker

Translated from Java to Python, this module is developed from the initial testing program and modified to function in the wider GapFiller program. It provides the means of checking whether a given file contains a gap large enough to accommodate the size of the GPG blob passed to it (in bytes). In order to ensure that any data either side of the gap is not compromised, the module automatically

adds a 100 byte "buffer" to the threshold value.

When passed a threshold and a path, it will walk through each file and subdirectory available to check its extension. For each file it comes across, it checks to see if its extension matches the predetermined list of acceptable ones (music file extensions, in this implementation). If this check is true, it will open the file as a byte array. For each byte in this array, it will check to see if it equals the hex value "00", corresponding to a null value. It will then proceed to count the number of consecutive null values until it hits a non-null value. At this point, if the number of consecutive null values hits the given threshold, it adds an entry into the array list of gaps to pass back to the controller, in the format [Path to file, start of gap, end of gap, consecutive empty bytes]. It then resets the counter and "start values" and continues to iterate through the file looking for other gaps.

For the purposes of this implementation, each gap is stored as a separate entry, regardless of how many gaps occur in each carrier file. This is for ease of later parsing and to potentially simplify future multi-gap/multi-file improvements to this program.

6.0.4 GapFillerStego

Where the actual steganographic operations take place. This module handles both retrieval and placement of data into steganographic targets. For placement of data into steganographic targets, the path to the ciphertext (in this case, the GPG blob) and the list of gaps is required. An output path is also (optionally) required, but a value must be passed in, even if it is blank. The program iterates through the list of gaps to find the largest value (simple logic, which can be expanded in future with improved logic such as maintaining a list of already used files/gaps) and opens the file with the largest gap as a byte array. A "to_write" array is initialised with the magic header "DE AD BE EF" to allow for easy detection of embedded data later, as initial research into GnuPG built-in magic headers was inconclusive.

The ciphertext is then opened as a byte array and appended to this "to_write" array, then appended with the magic footer value "BA DC 0F FE" to indicate where data retrieval should stop. A copy of the steganographic target is created. Next, starting from the location 50 bytes inside the gap, the contents of the "to_write" array are written into the byte array containing the target file.

The contents of the target file byte array are written to the copy of the steganographic target, then closed. The name of the target is prepended with the current UNIX epoch value to allow for the same target file to be used more than once without risk of overwriting any previous steganographic targets. The directory the steganographic target is located in (i.e. its containing folder) is also created to allow ease of location in the event of multiple files being present in the chosen output directory.

For retrieval of data from steganographic targets, the file containing potential steganographic data is opened as a byte array, and the magic header is searched for. If this value is found, the location of the potential start of data and the

byte array is passed to a retrieval method, which appends data bytes to an array to return until the magic footer is found. If the magic footer is not found, the retrieval method informs the user that no data was found and errors out, returning an empty array. If no header is found, the program similarly informs the user and terminates gracefully. If data is located and successfully extracted, the program then writes out the byte array to a file (message.txt.gpg) as it is assumed in this program that a text file was encoded as a GPG blob), and passes the path to that file back to the controller.

7 Evaluation

7.1 GapChecker

7.1.1 Testing and Verification

The sample data analysed was 5717 MP3 and M4A files in a personal music file collection, collated from around 2010 to 2019 from various sources (CD ripping and online sources like iTunes and AmazonMP3). A smaller subset was also tested (2757 files) along with four albums (encoded in 2012, 2014, 2015 and 2018).

7.1.2 Results and Discussion

See 1 for the table of results.

Of 5717 music files analysed, 5563 (97.31%) contained one or more gaps larger than 1000 bytes. Of the smaller subset of 2757 (the "Compilations" folder, containing larger than average albums), 2715 (98.48%) contained one or more gaps larger than 1000 bytes. On a singular album level (the "EDM", "Now 80" and "Now 99" albums), 100% of these had gaps larger than 1000 bytes, suggesting that, on an album level, if one of the songs contains a gap, it is likely that all of them will.

The album "Ministry Of Sound - Drive" was anomalous in that it seemed to contain no files with valid gaps but, when examined further, it was found to contain many small gaps of approximately 998 and 999 bytes each, lending credence to the consistency of format between songs on a particular album, as they are likely to have been encoded at the same time, with the same bitrate, from the same source, with the same algorithm.

In general, files that do not contain gaps of sufficient size are low bitrate, small file size rips or, as above, contain gaps slightly lower than 1000 bytes.

This experiment validates the hypothesis that, statistically, a given music file of reasonable quality is very likely to contain a gap sufficient to encode a small amount of text or a small file.

7.2 GapFiller

7.2.1 Evaluation Setup

In order to test the end-to-end experience, two virtual machines were set up with the GapFiller program, a random song was chosen as the file carrier, and a text file containing a quote from an internet video and some loriem ipsum text was created. A public key was published to a Keybase profile for the receiving computer to receive an encrypted message, and a keypair was generated locally on the sending computer for the purposes of signing a message for transmission. The files were exchanged through a shared network folder.

For multi-gap/multi-file, code was written to call GapFiller for each gap identified and place the above test message in it. This is intended to perform a "stress test" on the GapFillerStego portion of the program. The same was also done for the decryption portion of the program, to ensure that retrieval is possible across non cherry-picked samples. A modification was made to the base program to place timestamps at the start of filenames (both embedded files and retrieved messages) to cope with multiple gaps being available in one file, and to allow for independent retrieval verification. The test system will automatically calculate the ratio of placed messages vs retrieved messages and display it.

For data gathering, code was written to leverage GapFiller and music management libraries to gather statistical data about every file and gap larger than 1000 bytes, in order to identify correlations between average and maximum gap sizes, overall bandwidth and number of gaps available for steganographic operations, bitrate, file size and song duration.

For compression and re-encoding tests, several files with embedded steganographic data were converted to multiple other formats through Audacity (M4A, MP3, WMA, OGG and WAV) and separately had their bitrate lowered progressively. They were also compressed by general purpose algorithms (ZIP, 7Z, TAR and RAR), both individually and in groups.

For transmission tests, several files were sent and re-downloaded through the messaging/file sharing facilities of Facebook Messenger, Telegram, WhatsApp, Google Drive, OneDrive and Dropbox, both in ZIP archives and unzipped forms.

7.3 Testing Modules

7.3.1 MultiFileTest

This module is a modification of the GapFiller module, built to semi-automatically place a message into every gap found in a given directory and its subdirectories, then retrieve it back again.

The module initialises the GnuPG engine, then initialises values for the gaps placed during this session, and the gaps retrieved. It allows the specification of auto-completed text for the Keybase username, key email, file path and output path to reduce the amount of typing required over multiple runs (activated through changing the values specified, then typing "auto1" when prompted for a Keybase username). The class will then use the GnuPG module to import

the specified key, encrypt the message file and sign it. The class will then prompt for the directory to locate file gaps from and search that directory with GapChecker. Upon receipt of the gaps located, the class will take the length of the array as the number of gaps placed during the session. For each gap located, the "place" method of GapFillerStego is called with the specified output path, that gap information, and the GPG blob created previously.

Once every gap has been utilised, the user is then prompted for a location to place the retrieved message blobs. The module will then, for every file in the previously specified embedded file output directory, attempt to retrieve a message and place it in the specified message output path. If this retrieval is successful (i.e. If the "retrieve" method returns a path and not None), the gaps_retrieved counter is incremented.

Once all files in the embedded file output path are processed, the program presents the values of the gaps_retrieved and gaps_placed variables and calculates the percentage recovered. If the test is sound and GapFiller is fit for purpose, these values should be identical, and consequently the percentage recovered should be 100%.

7.3.2 DataGatherer

This module builds on the GapChecker module to gather more statistics about files in a given directory.

For each file in the given directory, DataGatherer will attempt to gather statistics from it. If the file is of a type with ID3 or APEv2 tagging (e.g. MP3, M4A and WMA files), the Mutagen library is used to retrieve the song's duration and bitrate. If the file has a different tagging scheme (in this case, WAV files) the SoundFile library is used to extract the sample rate, bit depth and number of channels in order to calculate duration and bitrate. Once this data gathering is complete, the file is opened as a byte array, the size of the byte array is used to calculate the file size in bytes, then is passed to GapChecker. If GapChecker locates gaps, DataGatherer creates an array in the format [bandwidth, extension, duration, bitrate, file_size, max_gap, avg_gap, no_gaps] and places the already gathered data within it. Then, for every gap found, it increments the no_gaps value in the array, and calculates the largest gap. It will then calculate the average gap size for that file, and add this array to the list of aggregated information to pass to the starter method.

Once the main method is complete, the aggregated file data is passed to the dump method, which reads in the array and writes it out to a file named "data_dump.dat" which is used by statistics programs for the purposes of analysis and correlation spotting.

8 Testing Results

8.1 End To End Experience

See this video for a demonstration of the end to end experience, along with an overall summary of the project.

8.2 Multiple Gaps and Multiple Files

For the set of 4291 MP3 files, 1246 M4A files, 1002 WAV files, and 34 WMA files, 8119 gaps large enough to hold a 1410 byte file (i.e. greater than 1510 bytes) were located. Of those 8119 gaps, 100% of them successfully had data placed into them, and 100% of them successfully had the data retrieved from them. This has been independently verified by comparing the hash value of a random subset of the extracted GPG files to the original and by counting the number of carrier files generated vs the number of GPG files generated.

For the set of 4291 MP3 files, 4173 gaps were identified, giving an average number of gaps per file of 0.97. For the set of 1246 M4A files, 3094 gaps were identified, giving an average of 2.48 gaps per file. For the 1002 WAV files and 34 WMA files, 784 and 68 gaps were identified, giving an average of 0.78 and 2.00 respectively. The overall average was 1.20 gaps per file.

8.3 Compression and Re-Encoding

Increasing and decreasing the bitrate of several MP3, M4A, WAV and WMA files did not retain any of the embedded data. Format shifting between these music formats produced similar results. The lack of embedded data was verified by using a hex editor to search for the magic header and footer ("DE AD BE EF" and "BA DC 0F FE" respectively).

Compressing and decompressing files through general purpose archive formats (ZIP, RAR and 7Z) retained the embedded data.

8.4 Transmission Medium

Transmission tests were conducted through WhatsApp and Telegram. Of the compressed and uncompressed files sent, 100% of them retained the embedded data through both mediums. This was verified through ensuring the checksums of the original and transmitted files matched, and through searching for the magic header and footer from a hex editor.

Uploading and downloading tests were conducted through cloud providers Dropbox, OneDrive and Google Drive. Downloading the files one at a time, and downloading them as a group (as web portals to cloud providers automatically compress multiple files into an archive if they are requested simultaneously) through all providers preserved the embedded data. This was verified as with the above transmission tests.

9 Discussion

9.1 MultiFileTest

See 2 for table of results.

This test has proven fit for purpose as it demonstrates that, irrespective of the underlying encoding format, there is a very high chance you will find a suitable gap to embed a short message and retrieve it again without any loss or corruption. The test itself however has some shortcomings. It can only process one file through one action at a time, and each gap is handled three times at different stages of the program (location, placement and retrieval). This, coupled with the fact that a file is likely to contain more than one gap, means that processing of any large dataset can easily take hours. Embedding messages in all 8119 gaps identified in the data set took approximately 6.5 hours and retrieving messages from these gaps took approximately 3 hours. It should be noted that MP3 files make up 63.5% of the test data, with the other three formats taking up 36.5%. As such, any conclusions reached are heavily biased in favour of this format. The WAV data, for example, is mostly comprised of short sound effect clips instead of full songs, therefore the average number of gaps per file is consequently lower. In order to more efficiently perform this test, some parallelisation could be introduced, but care must be taken to avoid the possibility of collisions. For example, linear processing means timestamps are guaranteed to be unique, but with parallelisation, there is a non-zero possibility of two gaps from the same file being processed at the same instant and being overwritten. The test also does not allow for the mass decryption of files. This decision was made as the current implementation of GnuPG requires you to re-enter your secret key password periodically as a security measure, which, given the timescales involved, severely hampers the automated, unsupervised nature of this test.

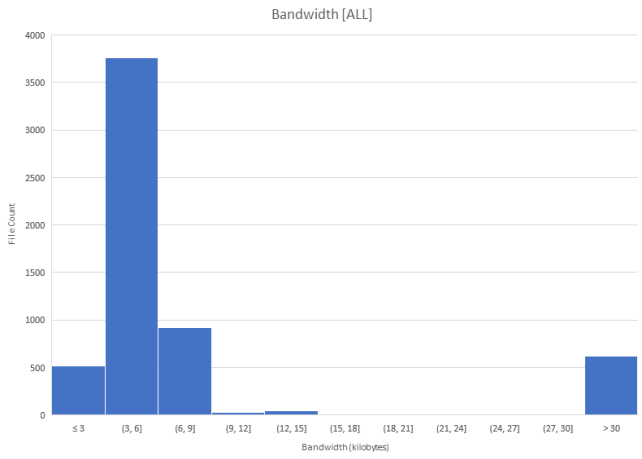


Figure 6: Histogram of bandwidth distribution

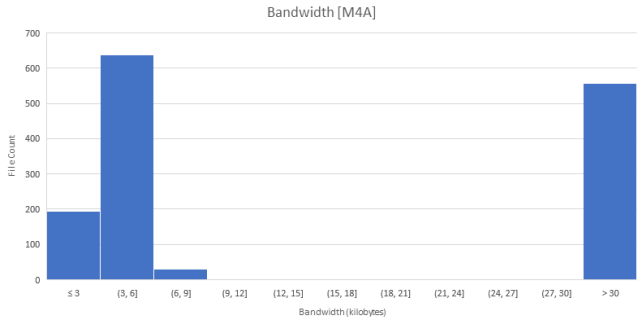


Figure 7: Histogram of bandwidth distribution [M4A]

9.2 DataGatherer

In order to draw reasonable correlations, statistical outliers in the data set were removed. The majority of files in this dataset have durations ranging from less than a second to 15 minutes, and file sizes ranging from 8 kilobytes to 61 megabytes 6. These removals consisted of three "continuous mixes" from a compilation album, with durations exceeding an hour and file sizes exceeding 100MB. While they were useful in the initial testing stages, the bounding on any graphs involving duration, bandwidth and file size would be too large to draw any meaningful data from with the inclusion of these data points.

Initially, a histogram to show the distribution of bandwidth overall and by extension was created. It showed that the majority of all files have a distribution between 3 and 6 kilobytes. When broken down by extension, this majority is comprised of M4A 7 and MP3 8 files, with WAV files 9 and WMA files 10 contributing the rest. The M4A files are shown to have a large distribution of bandwidths greater than 30kb 7, as do WMA files 10, which did not meet initial assumptions given their comparatively short average duration.

All kilobyte values are taken to be calculated as 1000 bytes per kilobyte. The first analysis undertaken was a comparison of average and maximum gap sizes by file extension. It was hypothesised that M4A files would have the greatest average gap score, and therefore the greatest maximum gap score, as, during data gathering, it appeared that M4A files had significantly larger gaps than their counterparts. Lowest gap size was not considered a relevant statistic, as there is a hard lower bound of 1000 bytes set by the data gathering process. Any gap smaller than this value is deemed to be less useful for steganographic purposes due to GNUPG encrypted and signed blobs having a lower size boundary of around a kilobyte, regardless of message size.

This hypothesis was confirmed in analysis 11, as the M4A format was shown to have an average gap size of 51.8 kilobytes (compared to the 4.6kb, 7.4kb and 3.4kb of MP3, WAV and WMA respectively), and an average maximum gap size of 140.1kb (compared to 4.7kb, 12.7kb and 3.9 kb of MP3, WAV and WMA respectively). This hypothesis was also supported by an analysis of average

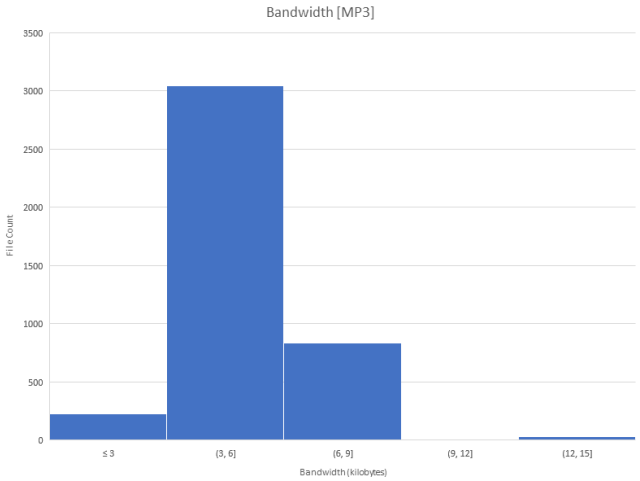


Figure 8: Histogram of bandwidth distribution [MP3]

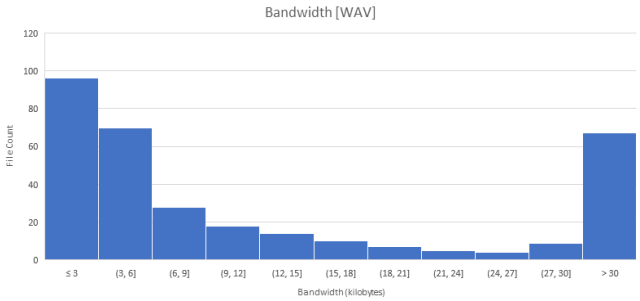


Figure 9: Histogram of bandwidth distribution [WAV]

bandwidth (i.e. the amount of useful space in every gap in the file) 12 by extension.

This analysis showed that M4A has an average bandwidth of 15.4kb per file (compared to the 2.1kb, 0.7 and 0.4kb of WAV, WMA and MP3 respectively), confirming that, for a potential later iteration of GapFiller that can take advantage of multiple gaps in a file, M4A is still the optimum choice to maximise useful bandwidth.

This conclusion gave rise to another hypothesis, namely, given that M4A has the highest average and highest maximum gap sizes, it may also have the greatest average number of gaps per file. This hypothesis was disproven 13, as WAV files have the highest average gap count at 2.7, compared to M4A at 2.3, and WMA and MP3 at 2.1 and 1.0 respectively. This data set would seem to suggest that, even though M4A formatted files contain, statistically, a slightly lower number of gaps, these gaps are of a larger size and, therefore, of greater use

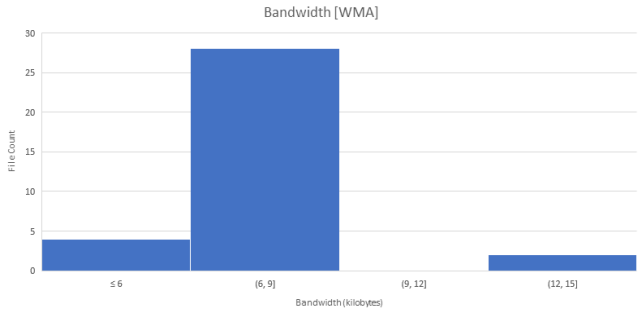


Figure 10: Histogram of bandwidth distribution [WMA]

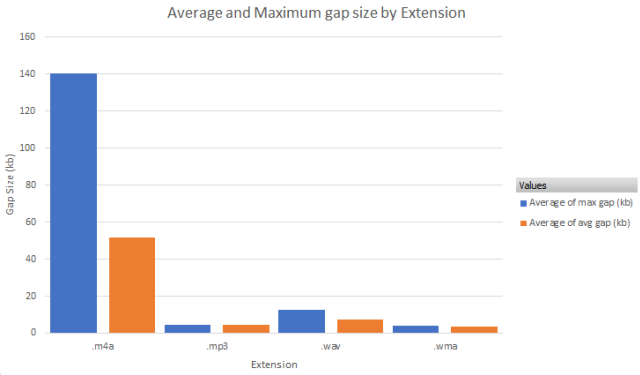


Figure 11: Average and Maximum gap size by extension

to this iteration of GapFiller. These results differ from those obtained during the MultiFileTest testing, as the threshold for gap sizes was set at 1000 bytes, rather than 1510 bytes (the size of the encrypted blob used for that test plus 100 bytes).

An analysis was undertaken to determine if there is any correlation between the available bandwidth in a file and its duration. It was hypothesised before data gathering began that there would be a positive correlation between duration and bandwidth, as, intuitively, it was considered that as the duration of the song increased, the size of the file would increase, increasing the amount of unused space, increasing the available bandwidth. It was also hypothesised that there would be some correlation between bitrate and bandwidth, but it was unclear what this would be given the relationship between bitrate and file size is clear for equally encoded and equal duration files. This relationship is less clear when it comes to slack space.

The positive correlation between bandwidth and duration was not proven 14, as the Pearson correlation coefficient value of average bandwidth and duration is *0.116584885*, indicating that there is no clear correlation between song duration

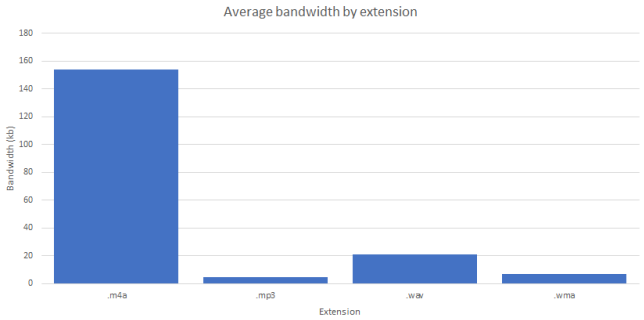


Figure 12: Average bandwidth by extension

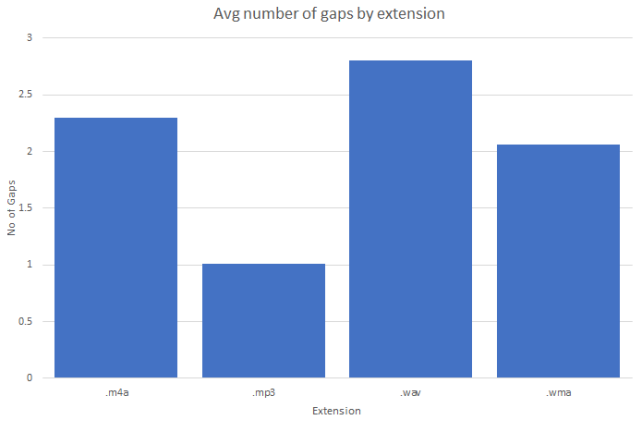


Figure 13: Average Number of gaps by extension

and the available bandwidth for steganographic operations. This may be due to a multitude of factors. The most likely explanation is that the variation between the exact amount of song data available at any particular time and the final set bitrate of the file is less predictable, making the amount of available slack space (and bandwidth available to GapFiller) is equally unpredictable. It could also be due to differences in how each format handles sound sampling.

Equally, there is no proven correlation between bandwidth and bitrate 15, as the Pearson correlation coefficient value of average bandwidth and bitrate is *-0.075751799*. There are a few anomalous spikes in the data around common bitrates (256 kbps and 1411 kbps) that would seem to suggest that these formats are optimal for bandwidth. However, these spikes are expected, as 256kbps and 1411kpbs are the most common bitrates of the M4A and WAV formats in this dataset respectively, and are not reflected in the graph for the sake of clarity. These file formats have been independently proven to have the highest bandwidth by format, so consequently cannot be considered a reasonable correlation.

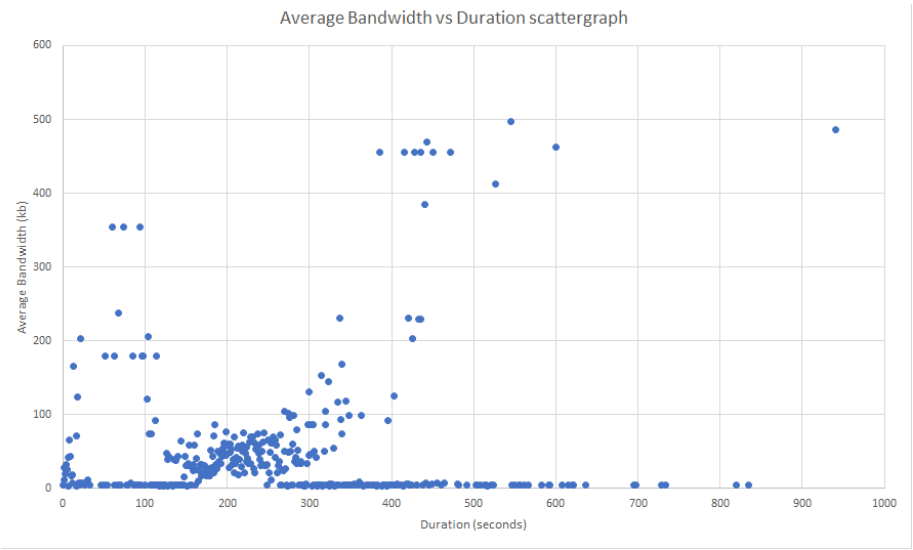


Figure 14: Average bandwidth by duration

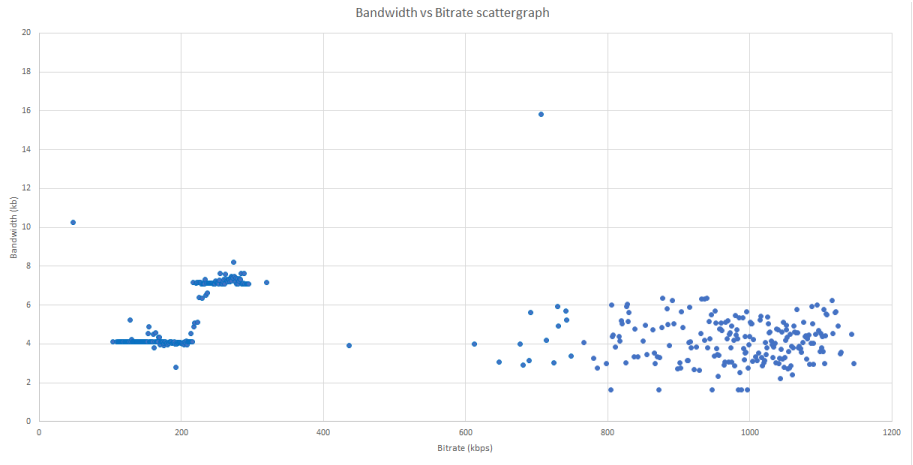


Figure 15: Average bandwidth by bitrate

9.3 Compression and Re-Encoding

See 3 for a full breakdown of conversions undertaken.

This test has proven the original hypothesis; that any modification to the underlying music file, be it format conversion or bitrate modification, will result in the destruction of the underlying data. This is likely due to the conversion algorithms not recognising the embedded data as either tag or music data, and thus discarding it during the remastering process. General purpose file compression, however, does not have these limitations, as these algorithms are lossless, and therefore do not make any determinations about what data can be lost. Therefore, it can be assumed any embedded data would be safe through general purpose compression.

10 Conclusion

This system relies upon modification of the underlying file structure, but does not rely upon any specific tagging scheme or compression standard as with existing schemes of this type. It allows an end user to take advantage of blank spaces available in multiple different file schemes. MP3, M4A, WMA and WAV files have been tested in this paper, but any tagging scheme that has blank spaces as a result of gaps between frames or has gaps between the header and the start of the music data can have steganographic operations performed on it with this system.

M4A files have been shown to be the optimal encoding scheme with regard to bandwidth, average gap size available, and maximum gap sizes. In fact, an M4A file had the largest gap of any file tested. Testing could not conclude if there is a correlation between the available bandwidth in a file and either its duration or its bitrate.

This system has been tested, both by user testing and automated testing, to reasonably guarantee that it will be able to take an MPEG file, a text message, a recipients Keybase username and email, retrieve that recipient's public key, encrypt the message, check to see if there is enough space in this carrier file to hide the message, and if so perform the steganographic operation. This process will provide the end user with the steganographically modified file, that is still recognised as a music file and has no audible changes when listened to.

The system also satisfies the Kerchoff principle, as, even if a hypothetical attacker has full knowledge of the steganographic scheme and its exact implementation details, they will be able to extract the embedded data but not the message itself, as that requires the recipient's private key. Using existing infrastructure and encryption programs reduces the program's complexity and increases the program's security.

The steganographic data has been proven to survive being sent through the messaging facilities of Facebook Messenger, Telegram and WhatsApp. It has also been proven to survive through the file sharing facilities of Google Drive, OneDrive and Dropbox, both through general purpose compression and through

individual, uncompressed downloading.

If re-encoding, format shifting or re-recording of the sound file is likely to be an issue during transmission, then the steganographic data will not survive, so another steganographic embedding scheme should be considered. However, this program, for the purposes of providing a computationally simple, relatively easy to use and secure method of passing information though existing messaging or social media links, is more than satisfactory.

This program will be made available as an open source, free to use and implement system for the benefit of the wider community.

11 Appendix A - User Guide

11.1 Prerequisites

- A Python IDE (PyCharm is strongly recommended).
- A Python virtual environment with the following packages:
  - os
  - pathlib
  - gnupg
  - time
  - mutagen
  - soundfile
  - math
- A Ubuntu/Linux distribution with GNUPG and Curl accessible from the command line (packages gnupg and curl if your distro doesn't have them available by default).
- An account at keybase.io.
  - You and your recipient must have a keypair (generated with Keybase or GNUPG), with the public key published to Keybase. (see these instructions for how to generate a keypair from the command line with GNUPG)
  - You and your recipient must also know your respective username, and the email tied to your public keys.
- A text file containing the message you wish to send.
- A music file (or ideally a set of music files) to perform steganography on.
- A mechanism to send the file (most established messaging systems will do).

11.2 Encryption

1. Run the "GapFiller.py" file to start the program.
2. At the first prompt, type "e" to start the encryption process.
3. At the second prompt, type the Keybase username of your recipient, then at the third prompt type the recipient's email (this must match the email associated with the public key).
4. Finally, enter the (relative or full) path to the file containing your message.
5. The program will then fetch the recipient's key from their Keybase profile, encrypt the text file with it, then attempt to sign the message with your private key. If prompted, enter the password to unlock your private key.
6. Once the encryption process is complete, the program will prompt you for a directory to search for suitable carrier files. It will then search every music file in this directory for gaps of a suitable size to place the message in.
    - If this directory contains a lot of files, this may take some time. If the directory doesn't contain any suitable files or does not exist, the program will reprompt you for a new path.
7. Once the search process is complete, the program will select the file with the largest available gap, and prompt you for a location to save the carrier file to. You may leave this blank if you wish to place the carrier file in the program directory.
8. The program will then perform the steganographic operation, and place the resulting file in the location [SpecifiedDirectory]/[OriginalDir]/[Epoch]-[OriginalMusicName].[ext], then displays the path to you. You may now send this file through any messaging means you wish.
    - The program prepends the current time (in Unix epoch) to the start of the file to prevent any overwriting resulting in reuse of carrier files. This can be renamed before sending.

11.3 Decryption

1. Run the "GapFiller.py" file to start the program.
2. At the first prompt, type "d" to start the decryption process.
3. At the second prompt, type the path to the file you wish to check for encrypted content.
4. At the third prompt, type the path to the location you wish to place the encrypted file (which will be named as [Epoch]message.txt.GPG).

5. The program will print “dead beef located”, “bad coffee located” and the message itself if a message is embedded, and will print “no message encoded” if not. It will then save the message to “out.txt” in the program directory.

    - The GPG module will give you information on the message’s signature and verify it as good if you have access to the sender’s public key.
    - The GPG module will also warn you that you are using an untrusted key. Please disregard this, as GNUPG will automatically distrust any key you haven’t explicitly “trusted and signed” through its system. It can be assumed any keys acquired from Keybase can be trusted.

12 Appendix B - Source Code

12.1 GapChecker

The source code is available at [58]

12.1.1 Program.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Program {

    public static void main(String[] args) throws IOException {
        FileWalker fw = new FileWalker();
        Path startDir = Paths.get("D:\\Public\\Music");
        Path fv = Files.walkFileTree(startDir, fw);

        fw.gc.summariseStats(startDir);
    }

}
```

12.1.2 FileWalker.java

```
import java.io.IOException;
import java.nio.file.FileVisitResult;
import java.nio.file.Path;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;

import static java.nio.file.FileVisitResult.*;

public class FileWalker
        extends SimpleFileVisitor<Path> {
    public GapChecker gc = new GapChecker();

    @Override
    public FileVisitResult visitFile(Path file,
                                     BasicFileAttributes attr)
    throws IOException {

        if (attr.isRegularFile()) {

            System.out.format("Regular file: %s \n", file.
    getFileName());
            if (file.getFileName().toString().endsWith("mp3") ||
    file.getFileName().toString().endsWith("m4a")) {
                gc.getStats(file);
            }
            System.out.println(" (" + attr.size() + "bytes)");
            System.out.println();
        }

        return CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file,
                                           IOException exc) {
        System.err.println(exc);
        return CONTINUE;
    }

}
```

12.1.3 GapChecker.java

```
1  import java.io.*;
2  import java.math.BigInteger;
3  import java.nio.file.Files;
4  import java.nio.file.Path;
5  import java.util.ArrayList;
6
7  public class GapChecker {
8      ArrayList<Integer> gapSizes = new ArrayList<>();
9      Integer average = 0;
10     Integer total = 0;
11     Integer validFilesSeen = 0;
12     Integer filesWithGaps = 0;
13
14     public void getStats(Path filePath) throws IOException {
15
16         byte[] musicBytes = Files.readAllBytes(filePath);
17         validFilesSeen++;
18         int consecEmpty = 0;
19         int start = 0;
20         int end = 0;
21         boolean gap = false;
22         for (int i = 0; i < musicBytes.length - 1; i++) {
23
24             if (musicBytes[i] == 0) {
25                 if (start == 0) {
26                     start = i;
27                 }
28                 consecEmpty++;
29             } else {
30                 end = i;
31                 if (consecEmpty >= 1000) {
32                     gap = true;
33                     gapSizes.add(consecEmpty);
34                     System.out.println("has " + consecEmpty + "
    bytes available from " + start + " to " + (end - 1));
35                 }
36                 start = 0;
37                 consecEmpty = 0;
38             }
39         }
40         if (gap) {
41             filesWithGaps++;
42         }
43     }
44
45     public void summariseStats(Path sd) throws IOException {
46
47         for (int g : gapSizes) {
48             total = total + g;
49         }
50
51         if (gapSizes.size() != 0) {
52             average = total / gapSizes.size();
53         }else{
```

```
54              System.out.println("No gaps larger than 1000 bytes
    found");
55          }
56          try (Writer writer = new BufferedWriter(new
    OutputStreamWriter(
57                  new FileOutputStream("averageAll.txt"), "utf-8")))
    {
58              writer.append(sd.toString() + ": \t");
59              writer.append("Average size of gaps is " + average);
60              writer.append("\t Valid files seen : " + validFilesSeen
    );
61              writer.append("\t Valid files with gaps : " +
    filesWithGaps);
62              writer.append("\t Total Gaps : " + gapSizes.size());
63              writer.append("\n");
64
65          }
66      }
67
68  }
```

12.2 GapFiller

The source code is available at [58]

12.2.1 gapfiller.py

```
1  import GnuPG
2  import GapChecker
3  import GapFillerStego
4  import os
5
6
7  def main():
8      GnuPG.main()  # initialises the GnuPG engine
9
10     print("Do you wish to (e)ncrypt a message or (d)ecrypt a file?"
       )
11     choice = input("> ")
12
13     if choice.lower() == "e":
14         print("Please enter the Keybase username of the recipient")
15         username = input(">")
16
17         print("Please enter the email of the recipient")
18         email = input(">")
19
20         print("Please enter the path to the file you wish to
       encrypt and sign")
21         path_to_file = input(">")
22
23         print("%s %s %s" % (username, email, os.path.realpath(
       path_to_file)))  # DEBUG
24         GnuPG.import_key(username)
25         GnuPG.encrypt_message(path_to_file, email)
```

```
26          file_size = os.path.getsize(path_to_file + '.gpg')
27
28          gaps = getGaps(file_size)
29          print(gaps)
30
31          print("Please enter the directory that you wish to place
        the outputted files (leave blank for this dir)")
32          path_to_output = input(">")
33          if path_to_output == "":
34              path_to_output = "./"
35
36          GapFillerStego.place(path_to_file + '.gpg', gaps,
        path_to_output)
37
38      elif choice.lower() == "d":
39          print("Please enter the path to the file you wish to check
        for encrypted content")
40          path_to_file = input(">")
41          print("Please enter the directory that you wish to place
        the outputted file (leave blank for this dir)")
42          path_to_output = input(">")
43          if path_to_output == "":
44              path_to_output = "./"
45
46
47          print("%s" % (os.path.realpath(path_to_file)))
48          ciphertext_path = GapFillerStego.retrieve(path_to_file,
        path_to_output)
49          if ciphertext_path is not None:
50              GnuPG.decrypt_message(ciphertext_path)
51      elif choice.lower() == "exit":
52          exit(0)
53      else:
54          print("Invalid input, please try again")
55          main()
56
57
58  def getGaps(file_size):
59      print("Please enter the directory you wish to find a file
        target from")
60      stego_directory = input(">")
61
62      gaps = GapChecker.main(os.path.realpath(stego_directory),
        file_size)
63      if not gaps:
64          print("No suitable files found in selection, please try
        again")
65          getGaps(file_size)
66      return gaps
67
68
69  if __name__ == "__main__":
70      main()
```

12.2.2 GnuPG.py

```
import os
import gnupg

homedir = ""
gpg = ""


def encrypt_message(file, email):  # takes given full real path, encrypts with Public key and signs
    print(file)  # DEBUG
    result = os.system("gpg --trust-model always --yes -se -r %s %s" % (email, file))
    if result != 0:
        print("Encryption failure")
        exit(1)


def decrypt_message(file):  # takes given full real path and decrypts it with private key
    result = os.system("gpg --trust-model always --decrypt %s" % file)
    if result != 0:
        print("Decryption failure")
        exit(1)


def import_key(uname):  # curls out to Keybase with given username and imports their pgp keys
    r = os.system("curl https://keybase.io/%s/pgp_keys.asc >> %s/%s.asc" % (uname, homedir, uname))
    os.system("gpg --import %s/%s.asc" % (homedir, uname))


def main():
    global homedir
    global gpg
    os.system("cd /")
    homedir = ".gnupg"
    homedir = os.path.join(os.getenv("HOME"), homedir)
    print(homedir)
    gpg = gnupg.GPG()


if __name__ == "__main__":
    main()
```

12.2.3 GapChecker.py

```
import os
import pathlib

def gapfinder(array, path, threshold):  # parses through file to count gaps of "00"
    found = []
    # print("File size %s bytes" % threshold)  # DEBUG
    threshold += 100  # threshold
    # print("Safety threshold %s bytes" % threshold)  # DEBUG

    i = 0
    start = -1
    consec_empty = 0

    for b in array:  # for each byte in the array
        if b == 0:
            if start == -1:  # if byte empty, and no start has been identified
                start = i
            consec_empty += 1
        else:  # if b is not empty, start end action
            end = i - 1
            if consec_empty >= threshold:  # if consec_empty hits threshold
                print("File %s has %s bytes available from %s to %s" % (path, (consec_empty - 1), start, end))
                # print(array[start], array[end])  # DEBUG
                found.append([path, start, end, consec_empty - 1])  # add gap info to array to send back
            start = -1  # reset start counter and empty count
            consec_empty = 0
        i += 1  # iterate through counter
    return found


def ismusic(ext):
    return ext == ".mp3" or ext == ".m4a" or ext == ".wav" or ext == ".ogg" or ext == ".aac" or ext == ".wma"


def main(path, threshold):
    gaps = []
    files_found = 0
    for root, dirs, files in os.walk(path):
        for name in files:  # for each file in path
            path = os.path.join(root, name)
            ext = pathlib.Path(path).suffix
            print("FILE: %s EXTENSION: %s" % (path, ext))  # DEBUG
            if ismusic(ext):  # if passes as music file

                files_found += 1
                in_file = open(path, "rb")  # [r]ead as [b]yte array
                data = in_file.read()
                in_file.close()
```

```
                found = gapfinder(data, path, threshold)
                for f in found:
                    gaps.append(f)

        for name in dirs:
            print("DIR: %s" % os.path.join(root, name))

    return gaps


if __name__ == "__main__":
    print("Please enter the directory/file you wish to search")
    path_to_file = input("> ")
    x = main(os.path.realpath(path_to_file), 900)
    print(x)
```

12.2.4 GapFillerStego.py

```
import os
import pathlib
import time


def get_blob(stego, start):
    blob = []
    for i in range(start, len(stego) - 1):
        if stego[i] == b'\xBA':
            if stego[i + 1] == b'\xDC' and stego[i + 2] == b'\x0F'
    and stego[i + 3] == b'\xFE':
                print("Bad coffee located")
                return blob
            else:
                blob.append(stego[i])
        else:
            blob.append(stego[i])
    print("ERROR = no footer found")
    return []


def place(ciphertext, gaps, output_path):
    largest = ['', 0, 0, 0]  # [path, start, end, consec_empty - 1]
    for g in gaps:  # simple logic to take the largest gap
    available
        if g[3] >= largest[3]:
            largest = g
    # print(largest) # DEBUG

    target_bytes = []  # bytes to write out to new file

    to_write = [b'\xDE', b'\xAD', b'\xBE', b'\xEF']  # bytes of
    ciphertext with magic header

    with open(largest[0], "rb") as file:  # read in stego target as
     byte array
        for byte in iter(lambda: file.read(1), b''):
            target_bytes.append(byte)
```

```
35
36      with open(ciphertext, "rb") as file:  # read in gpg blob as
        byte array
37          for byte in iter(lambda: file.read(1), b''):
38              to_write.append(byte)
39
40      to_write.append(b'\xBA')  # magic footer
41      to_write.append(b'\xDC')
42      to_write.append(b'\x0F')
43      to_write.append(b'\xFE')
44
45      file_name = str(time.time()) + " - " + largest[0].split("/")
        [-1]
46
47      dir = largest[0].split("/")[-2]
48
49      print(file_name)
50
51      file_path = os.path.join(output_path, dir)
52      pathlib.Path(file_path).mkdir(parents=True, exist_ok=True)
53      name = os.path.join(file_path, file_name)
54
55      print(name)
56      target_copy = open(os.path.realpath(name), "wb+")  # create
        output music file
57
58      j = 0  # stego inner counter
59      for i in range(largest[1] + 50, (largest[1] + 50 + len(to_write
        ))):  # from the location 50 bytes into the gap
60          target_bytes[i] = to_write[j]  # write each byte of
        ciphertext to stego target
61          j += 1
62
63      for b in target_bytes:  # write out to copy of stego target
64          target_copy.write(bytes(b))
65      target_copy.close()
66
67
68  def retrieve(stego_target, output_path):
69      stego_bytes = []
70      output_file = str(time.time()) + "message.txt.gpg"
71      file_path = os.path.join(output_path, output_file)
72      with open(stego_target, "rb") as file:  # read in stego target
        as byte array
73          for byte in iter(lambda: file.read(1), b''):
74              stego_bytes.append(byte)
75
76      for i in range(0, len(stego_bytes) - 1):
77          if stego_bytes[i] == b'\xDE':
78              if stego_bytes[i + 1] == b'\xAD' and stego_bytes[i + 2]
         == b'\xBE' and stego_bytes[i + 3] == b'\xEF':
79                  print("Dead beef located")
80                  blob = get_blob(stego_bytes, i + 4)
81                  target_copy = open(file_path, "wb")
82                  for b in blob:
83                      target_copy.write(b)
84                  target_copy.close()
```

```
85
86                  return os.path.join(output_path, output_file)
87      print("No message encoded")
88      return None
89
90
91  if __name__ == "__main__":
92      gaps = [['/media/grant/DATA/Grant/Documents/Uni/CS408/Practical
        /gapchecker/17-Cant-Sleep-Love.m4a', 40340, 45048,
93              4708]]
94
95      text = os.path.realpath('out.txt.gpg')
96      place(text, gaps, "/media/grant/DATA/Grant/Documents/Uni/CS408/
        Practical/gapfiller/")
97
98      retrieve(os.path.realpath('output.m4a'))
```

12.2.5 [Test] MultiFileTest.py

```
1  # TEST FOR MULTIPLE FILES/MULTIPLE GAPS
2  import os
3
4  import GapChecker
5  import GapFillerStego
6  import GnuPG
7
8
9  def main():
10     GnuPG.main()  # initialises the GnuPG engine
11     gaps_placed = 0
12     gaps_retrieved = 0
13
14     print("Please enter the Keybase username of the recipient")
15     username = input(">")
16
17     if username == "":  # Use this to automate multiple runs and
       reduce the amount of typing
18         username = ""
19         email = ""
20         path_to_file = ""
21         path_to_output = ""
22     else:
23         print("Please enter the email of the recipient")
24         email = input(">")
25
26         print("Please enter the path to the file you wish to
       encrypt and sign")
27         path_to_file = input(">")
28
29         print("Please enter the directory that you wish to place
       the outputted files")
30         path_to_output = input(">")
31
32     print("%s %s %s" % (username, email, os.path.realpath(
       path_to_file)))  # DEBUG
33     GnuPG.import_key(username)
```

```
34      GnuPG.encrypt_message(path_to_file, email)
35      file_size = os.path.getsize(path_to_file + '.gpg')
36
37      gaps = getGaps(file_size)
38      print(gaps)
39      gaps_placed = len(gaps)
40
41      for x in gaps:
42          GapFillerStego.place(path_to_file + '.gpg', [x],
        path_to_output)
43
44      print("Please enter the directory that you wish to place the
        outputted files")
45      msg_output = input(">")
46
47      for root, dirs, files in os.walk(path_to_output):
48          for name in files:  # for each file in path
49              success = GapFillerStego.retrieve(os.path.join(root,
        name), os.path.realpath(msg_output))
50              if success is not None:
51                  gaps_retrieved = gaps_retrieved + 1
52
53      print("Gaps Placed - %s, Gaps Retrieved - %s, Percentage
        Recovered - %s" % (
54          gaps_placed, gaps_retrieved, ((gaps_retrieved / gaps_placed
        ) * 100)))
55
56
57  def getGaps(file_size):
58      print("Please enter the directory you wish to find a file
        target from")
59      stego_directory = input(">")
60
61      gaps = GapChecker.main(os.path.realpath(stego_directory),
        file_size)
62      if not gaps:
63          print("No suitable files found in selection, please try
        again")
64          getGaps(file_size)
65      return gaps
66
67
68  if __name__ == '__main__':
69      main()
```

12.2.6 [Test] DataGatherer.py

```
import math
import os
import pathlib
import mutagen
import GapChecker
import soundfile as sf


def dump(data):
    f = open("data_dump1.dat", "w+")

    f.read()
    f.write("# [bandwidth, ext, duration, bitrate, file_size,
    max_gap, avg_gap, no_gaps]\r")

    for n in range(len(data)):
        for i in range(len(data[n])):
            f.write(str(data[n][i]))
            f.write(" ")


def main(path, threshold):
    per_file = []

    files_found = 0

    for root, dirs, files in os.walk(path):
        for name in files:  # for each file in path
            path = os.path.join(root, name)
            ext = pathlib.Path(path).suffix
            print("FILE: %s EXTENSION: %s" % (path, ext))  # DEBUG
            if GapChecker.ismusic(ext):  # if passes as music file
                if ext == ".wav":  # special way of calculating
    duration and bitrate for WAV files
                    f = sf.SoundFile(path)
                    duration = (len(f) / f.samplerate)
                    word = f.subtype
                    word = int(word.split("_")[-1])
                    bitrate = str((f.samplerate * word * f.channels
    ) / 1000)


                else:
                    # get file metadata
                    cur_file = mutagen.File(path)
                    if cur_file is not None:
                        duration = math.floor(cur_file.info.length)
                        bitrate = math.floor(cur_file.info.bitrate
    / 1000)

                files_found += 1
                in_file = open(path, "rb")  # [r]ead as [b]yte
    array

                data = in_file.read()
```

```
51                  size = len(data)
52                  in_file.close()
53                  found = GapChecker.gapfinder(data, path, threshold)
54
55                  # aggregates all useful bandwidth in the file
56                  aggregate = [0, ext, duration, bitrate, size, 0, 0,
      0]
57                  # [0=bandwidth, 1=ext, 2=duration, 3=bitrate, 4=
    file_size, 5=max_gap, 6=avg_gap, 7=no_gaps]
58
59                  # [path, start, end, consec_empty - 1]
60                  for f in found:  # for all gaps found
61                      aggregate[7] += 1  # increment number of gaps
62                      aggregate[0] += f[3]  # add to bandwidth value
63                      if f[3] > aggregate[5]:  # if current size is
    bigger than max_size, replace
64                          aggregate[5] = f[3]
65                  if aggregate[0] != 0 and aggregate[7] != 0:
66                      aggregate[6] = math.floor(aggregate[0] /
    aggregate[7])  # avg = bandwidth/no_gaps
67                      per_file.append(aggregate)
68
69          for name in dirs:
70              print("DIR: %s" % os.path.join(root, name))
71
72      return per_file
73
74
75  if __name__ == "__main__":
76      print("Please enter the directory/file you wish to search")
77      path_to_file = input("> ")
78      x = main(os.path.realpath(path_to_file), 900)
79
80      print(x)
81
82      dump(x)
```

13 Appendix C - Figures and Tables

GapChecker Results					
Containing Folder	MPEG Files	Files with Gaps	%	Avg Gap Size	Total Gaps
Music	5717	5563	97.31	5563	7425
Compilations	2757	2715	98.48	24220	3716
EDM	50	50	100	7118	50
Now 80	43	43	100	2096	78
Now 99	45	45	100	8193	45
Drive	42	0	0	0	0

Table 1: GapChecker Results

MultiFileTest Results				
File Extension	Number of Files	Gaps Found	Avg Gaps per file	% Gaps Successfully Used
MP3	4291	4173	0.97	100
M4A	1246	3094	2.48	100
WAV	1002	784	0.78	100
WMA	34	68	2.00	100
TOTAL	6753	8119	1.20	100

Table 2: MultiFileTest Results

Compression/Reencoding Results				
File Name	Bitrate (kbps)	Modified Bitrate (kbps)	Modified Extension	Data Retained?
Lego House.M4A	260	144	MP3	FALSE
Lego House.M4A	260	56	MP3	FALSE
Lego House.M4A	260	N/A	OGG	FALSE
Lego House.M4A	260	1411	WAV	FALSE
Lego House.M4A	260	195	M4A	FALSE
Lego House.M4A	260	128	WMA	FALSE
Attack Jet.WAV	1411	1411	WAV	FALSE
Attack Jet.WAV	1411	320	MP3	FALSE
Attack Jet.WAV	1411	N/A	OGG	FALSE
Castle on The Hill.MP3	278	149	MP3	FALSE
Castle on The Hill.MP3	278	64	MP3	FALSE
Castle on The Hill.MP3	278	1411	WAV	FALSE
Castle on The Hill.MP3	278	N/A	OGG	FALSE
Castle on The Hill.MP3	278	128	WMA	FALSE
Cheerleader.WMA	128	128	MP3	FALSE
Cheerleader.WMA	128	128	MP3	FALSE
Cheerleader.WMA	128	195	M4A	FALSE
Cheerleader.WMA	128	N/A	OGG	FALSE
Eraser.MP3	320	N/A	ZIP	TRUE
Eraser.MP3	320	N/A	7Z	TRUE
Eraser.MP3	320	N/A	RAR	TRUE
Divide (Deluxe)	N/A	N/A	ZIP	TRUE
Divide (Deluxe)	N/A	N/A	7Z	TRUE
Divide (Deluxe)	N/A	N/A	RAR	TRUE

Table 3: Compression/Re-encoding Results

References

[1] N. Cvejic and T. Seppanen, "A wavelet domain lsb insertion algorithm for high capacity audio steganography," in *Proceedings of 2002 IEEE 10th Digital Signal Processing Workshop, 2002 and the 2nd Signal Processing Education Workshop.*, IEEE, 2002, pp. 53–55.

[2] M. S. Atoum, S. Ibrahimn, G. Sulong, A. Zeki, and A. Abubakar, "Exploring the challenges of mp3 audio steganography," in *2013 International Conference on Advanced Computer Science Applications and Technologies*, IEEE, 2013, pp. 156–161.

[3] K. Rabah, "Steganography-the art of hiding data," *Information Technology Journal*, vol. 3, no. 3, pp. 245–269, 2004.

[4] L. Eko and L. Hellmueller, "One meta-media event, two forms of censorship: The charlie hebdo affair in the united kingdom and turkey," *Global Media and Communication*, vol. 0, no. 0, p. 1 742 766 519 899 118, 0. DOI:

10.1177/1742766519899118. eprint: https://doi.org/10.1177/1742766519899118.

[5] P. Sadler, *National security and the D-notice system*. Routledge, 2018.

[6] J. D. Martin, D. Abbas, and R. J. Martins, "The validity of global press ratings: Freedom house and reporters sans frontières, 2002–2014," *Journalism Practice*, vol. 10, no. 1, pp. 93–108, 2016.

[7] P. Lorentzen, "China's strategic censorship," *American Journal of Political Science*, vol. 58, no. 2, pp. 402–414, 2014. DOI: 10.1111/ajps.12065. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/ajps.12065. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/ajps.12065.

[8] D. C. Nunziato, *Virtual freedom: Net neutrality and free speech in the Internet age*. Stanford University Press, 2009.

[9] D. Naylor, "Architectural support for managing privacy tradeoff s in the internet," Ph.D. dissertation, ETH Zürich.

[10] F. Brunton, "Wikileaks and the assange papers," *Radical Philosophy*, vol. 166, pp. 8–20, 2011.

[11] A. M. Froomkin, "The metaphor is the key: Cryptography, the clipper chip, and the constitution," *University of Pennsylvania Law Review*, vol. 143, no. 3, pp. 709–897, 1995.

[12] C. E. Torkelson, "The clipper chip: How key escrow threatens to undermine the fourth amendment," *Seton Hall L. Rev.*, vol. 25, p. 1142, 1994.

[13] K. Ermoshina, F. Musiani, and H. Halpin, "End-to-end encrypted messaging protocols: An overview," in *International Conference on Internet Science*, Springer, 2016, pp. 244–254.

[14] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0.," in *USENIX Security Symposium*, vol. 348, 1999, pp. 169–184.

[15] S. Ruoti, J. Andersen, D. Zappala, and K. Seamons, "Why johnny still, still can't encrypt: Evaluating the usability of a modern pgp client," *arXiv preprint arXiv:1510.08555*, 2015.

[16] M. Bellare and B. Yee, "Forward-security in private-key cryptography," in *Cryptographers' Track at the RSA Conference*, Springer, 2003, pp. 1–18.

[17] P. Saint-Andre *et al.*, "Extensible messaging and presence protocol (xmpp): Core," 2004.

[18] J. Oikarinen and D. Reed, "Internet relay chat protocol," 1993.

[19] C. C. Werry, "Internet relay chat," *Computer-mediated communication: Linguistic, social and cross-cultural perspectives*, pp. 47–63, 1996.

[20] *Ircv3 working group*, 2016. [Online]. Available: https://ircv3.net/.

[21] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record communication, or, why not to use pgp," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, ser. WPES '04, Washington DC, USA: Association for Computing Machinery, 2004, pp. 77–84, ISBN: 1581139683. DOI: 10.1145/1029179.1029200.

[22] M. Di Raimondo, R. Gennaro, and H. Krawczyk, "Secure off-the-record messaging," in *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, ser. WPES '05, Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 81–89, ISBN: 1595932283. DOI: 10.1145/1102199.1102216.

[23] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[24] H. Krawczyk, M. Bellare, and R. Canetti, *Hmac: Keyed-hashing for message authentication*, 1997.

[25] K. Ermoshina and F. Musiani, "Standardising by running code: The signal protocol and de facto standardisation in end-to-end encrypted messaging," *Internet Histories*, vol. 3, no. 3-4, pp. 343–363, 2019.

[26] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2017, pp. 451–466.

[27] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, Apr. 2018, pp. 415–429. DOI: 10.1109/EuroSP.2018.00036.

[28] E. Franz, A. Jerichow, S. Möller, A. Pfitzmann, and I. Stierand, "Computer based steganography: How it works and why therefore any restrictions on cryptography are nonsense, at best," in *Information Hiding*, R. Anderson, Ed., Springer Berlin Heidelberg, 1996, pp. 7–21, ISBN: 978-3-540-49589-5.

[29] A. Brantly, "Banning encryption to stop terrorists: A worse than futile exercise," *CTC Sentinel*, vol. 10, no. 7, pp. 29–33, 2017.

[30] C. Everett, "Should encryption software be banned?" *Network Security*, vol. 2016, no. 8, pp. 14–17, 2016.

[31] A. Segal, "China, encryption policy, and international influence," *Hoover Institution, Beyond Privacy and Security series paper*, pp. 1–4, 2016.

[32] P. McLaughlin, "Crypto wars 2.0: Why listening to apple on encryption will make america more secure," *Temp. Int'l & Comp. LJ*, vol. 30, p. 353, 2016.

[33] I. Cox, M. Miller, J. Bloom, J. Fridrich, and T. Kalker, *Digital watermarking and steganography*. Morgan kaufmann, 2007.

[34] R. Chandramouli and N. Memon, "Analysis of lsb based image steganography techniques," in *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, IEEE, vol. 3, 2001, pp. 1019–1022.

[35] S. Dumitrescu, X. Wu, and Z. Wang, "Detection of lsb steganography via sample pair analysis," in *International workshop on information hiding*, Springer, 2002, pp. 355–372.

[36] K. J. Devi, "A secure image steganography using lsb technique and random pseudo random encoding technique," Ph.D. dissertation, 2013.

[37] E. Kawaguchi and R. O. Eason, "Principles and applications of bpcs steganography," in *Multimedia Systems and Applications*, International Society for Optics and Photonics, vol. 3528, 1999, pp. 464–473.

[38] J. Spaulding, H. Noda, M. N. Shirazi, and E. Kawaguchi, "Bpcs steganography using ezw lossy compressed images," *Pattern Recognition Letters*, vol. 23, no. 13, pp. 1579–1587, 2002.

[39] H. Ochoa-Dominguez and K. R. Rao, *Discrete Cosine Transform*. CRC Press, 2019.

[40] E. A. Fedorovskaya, H. de Ridder, and F. J. Blommaert, "Chroma variations and perceived quality of color images of natural scenes," *Color Research & Application: Endorsed by Inter-Society Color Council, The Colour Group (Great Britain), Canadian Society for Color, Color Science Association of Japan, Dutch Society for the Study of Color, The Swedish Colour Centre Foundation, Colour Society of Australia, Centre Français de la Couleur*, vol. 22, no. 2, pp. 96–110, 1997.

[41] D. Upham, "Steganographic algorithm jsteg," *Software available at http://zooid.org/~ paul/crypto/jsteg*, 1993.

[42] N. Provos and P. Honeyman, "Hide and seek: An introduction to steganography," *IEEE security & privacy*, vol. 1, no. 3, pp. 32–44, 2003.

[43] T. Pevny and J. Fridrich, "Multi-class blind steganalysis for jpeg images," in *Security, Steganography, and Watermarking of Multimedia Contents VIII*, International Society for Optics and Photonics, vol. 6072, 2006, 60720O.

[44] T. Pevny and J. Fridrich, "Merging markov and dct features for multi-class jpeg steganalysis," in *Security, Steganography, and Watermarking of Multimedia Contents IX*, International Society for Optics and Photonics, vol. 6505, 2007, p. 650 503.

[45] K. Solanki, A. Sarkar, and B. Manjunath, "Yass: Yet another steganographic scheme that resists blind steganalysis," in *International Workshop on Information Hiding*, Springer, 2007, pp. 16–31.

[46] X. Yu and N. Babaguchi, "Breaking the yass algorithm via pixel and dct coefficients analysis," in *2008 19th International Conference on Pattern Recognition*, IEEE, 2008, pp. 1–4.

[47] H. Dutta, R. K. Das, S. Nandi, and S. M. Prasanna, "An overview of digital audio steganography," *IETE Technical Review*, pp. 1–19, 2019.

[48] H. Karajeh, T. Khatib, L. Rajab, and M. Maqableh, "A robust digital audio watermarking scheme based on dwt and schur decomposition," *Multimedia Tools and Applications*, vol. 78, no. 13, pp. 18 395–18 418, 2019.

[49] H. Dutta, R. K. Das, S. Nandi, and S. R. M. Prasanna, "An overview of digital audio steganography," *IETE Technical Review*, vol. 0, no. 0, pp. 1–19, 2019. DOI: 10.1080/02564602.2019.1699454. eprint: https://doi.org/10.1080/02564602.2019.1699454. [Online]. Available: https://doi.org/10.1080/02564602.2019.1699454.

[50] R. Tanwar, K. Singh, M. Zamani, A. Verma, and P. Kumar, "An optimized approach for secure data transmission using spread spectrum audio steganography, chaos theory, and social impact theory optimizer," *Journal of Computer Networks and Communications*, vol. 2019, 2019.

[51] T. M. Cedric, R. W. Adi, and I. Mcloughlin, "Data concealment in audio using a nonlinear frequency distribution of prbs coded data and frequency-domain lsb insertion," in *2000 TENCON Proceedings. Intelligent Systems and Technologies for the New Millennium (Cat. No. 00CH37119)*, IEEE, vol. 1, 2000, pp. 275–278.

[52] S. Nagaraja and R. Anderson, "The snooping dragon: Social-malware surveillance of the tibetan movement," University of Cambridge, Computer Laboratory, Report, 2009.

[53] J. Gardiner and S. Nagaraja, "Blindspot: Indistinguishable anonymous communications," *arXiv preprint arXiv:1408.0784*, 2014.

[54] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg, "Skypemorph: Protocol obfuscation for tor bridges," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 97–108.

[55] L. Invernizzi, C. Kruegel, and G. Vigna, "Message in a bottle: Sailing past censorship," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ACM, pp. 39–48, ISBN: 1450320155.

[56] P. Golle, M. Jakobsson, A. Juels, and P. Syverson, "Universal re-encryption for mixnets," in *Cryptographers' Track at the RSA Conference*, Springer, pp. 163–178.

[57] G. Rodgers, *Gapchecker*, 2019. [Online]. Available: https://github.com/grantr98/GapChecker.

[58] ——, *Gapfiller*, 2020. [Online]. Available: https://github.com/grantr98/GapFiller.