Warsaw University of Technology

Faculty of Electronics and Information Technology

Ignacio Azagra Villares

Senior Design Project Report

EVALUATION OF STEGANOGRAPHIC COST FOR COVERT COMMUNICATION IN IP NETWORKS

Project done under the supervision of
Wojciech Mazurczyk, Ph.D, D.Sc.

Warszawa, June 2014

ABSTRACT

Network steganography encompasses the information hiding techniques that can be applied in communication network environments and that utilize hidden data carriers for this purpose. When describing a network steganography method despite the features like steganographic bandwidth, undetectability and robustness also steganographic cost should be considered. It is used as an indicator for the degradation or distortion of the carrier caused by the application of the steganographic method. In this master thesis we are going to evaluate how steganographic cost is affected in two different scenarios when using different steganographic methods, either separated or combined. We want to check the existence of two phenomena that can take place when combining two or more steganographic methods: superposition steganography and zero cost steganography.

TABLE OF CONTENTS

1. INTRODUCTION

Stolen data, information leaks, spying on the net: These and other similar terms and topics are subject to more and more debate. It is becoming apparent that studying and understanding all these topics has a huge importance, since they are being used more and more often. In fact, year 2011 was named by the media "the year of the hack" [1, 2], since there were a lot of data security breaches in different private companies and governments. The amount of stolen data was estimated in petabytes [3].

Most of these security breaches can be attributed to Operation Shady RAT [4, 2]. Numerous institutions around the world were targeted: in order to get the computers (or other electronic devices) infected, what they did was to deceive a lot of unaware users to open a specially crafted email (phishing), which was implanting a back door on their computers. Following this, the criminals were connecting to a website and downloading different files, which seemed to be legitimate (mainly HTML and JPEG files). But the truth is that the cybercriminals were encoding commands into these pictures and crafted web pages, hiding and making them invisible to unaware third-parties, smuggling them through firewalls into the different systems under attack. Then, these control commands were ordering victim's computer to obtain executable code from remote servers, allowing an outsider the access to local files on the compromised host [5]. These side channels in many cases remained open for months.

We are also witnessing in these last years the birth of a new kind of malware. In June 2010, Stuxnet [6], a computer worm was discovered. It was designed to affect Iranian nuclear power plants [7]. The next year, a new worm called Duqu was also found. It was pretty similar to Stuxnet, but unlike it, Duqu was used for extracting and stealing data, sending the captured information hidden into seemingly innocent pictures, which were traveling the network without raising any suspicion [8, 9, 2].

Nowadays, not only pictures are used as carriers. Music, videos, a packet traveling the net… All of these elements can be used as carriers for secret data.

This process of embedding secret information into an innocent looking carrier is not a recent invention – it has been known and used for ages by humankind. This is what we call steganography. The inverse of steganography is called steganalysis, which concentrates on the detection of covert communication.

The growing importance of steganography and steganalysis is what mainly motivated this Master thesis, in which we are going to implement different scenarios and make different trials in order to obtain some experimental results that will be helping us to prove the existence of two phenomena that can take place when combining two or more different steganographic methods: superposition steganography and zero cost steganography.

The results obtained in this thesis are important, since after confirming the existence of these two phenomena steganalyzers can take them into account, study them and start thinking about how to detect or counter them, or even use them (Since steganography has also some legal uses, like circumvention of web censorship and surveillance [10] or computer forensics). In fact, based on the experimental results provided in this thesis, a scientific paper for Security and Communication Networks Journal has been published [11].

The thesis is structured as it follows: since not everyone will be familiar with steganography and its related terms and characteristics, we will start by giving some background information about steganography history and basics. With this background, we should be able to understand the two pehenomena whose existence we want to prove: superposition steganography and zero cost steganography. After that, we will proceed to explain the considered steganographic scenarios for IP networks. Later, we will explain the experimental test-bed setup, and then we will continue explaining the steganographic scenarios and methods implementation. We will finish the thesis by commenting the obtained results and giving some final conclusions related with them. Two annexes including the programming code and the references and bibliography will also be included.

Let us start then by expanding on this steganography concept, explaining some cases of its usage, its characteristics and history.

2. Steganography history & basics

2.1 Characteristics of Steganography and its Relationship to Cryptography

Steganography is defined as the art of embedding secret messages (called steganograms) in a certain carrier, in order to communicate them in a covert manner [2]. With the use of steganography, we manage to hide the fact that we are sending a message. In other words, if not detected, both the sender and receiver are made "invisible".

Steganography is often mistaken with information hiding, because it isn't possible to establish a proper border between the two fields. There also exists a lack of a coherent classification for the newly invented clandestine methods for communication. It's not strange for example for a method to be classified as steganographic and for another method similar to the first one to be classified as an information hiding technique.

We can distinguish between what belongs to the spectrum of steganographic methods and what doesn't belong there just by looking at certain conditions that the method or technique must fulfill in order for it to be considered steganographic [2]:

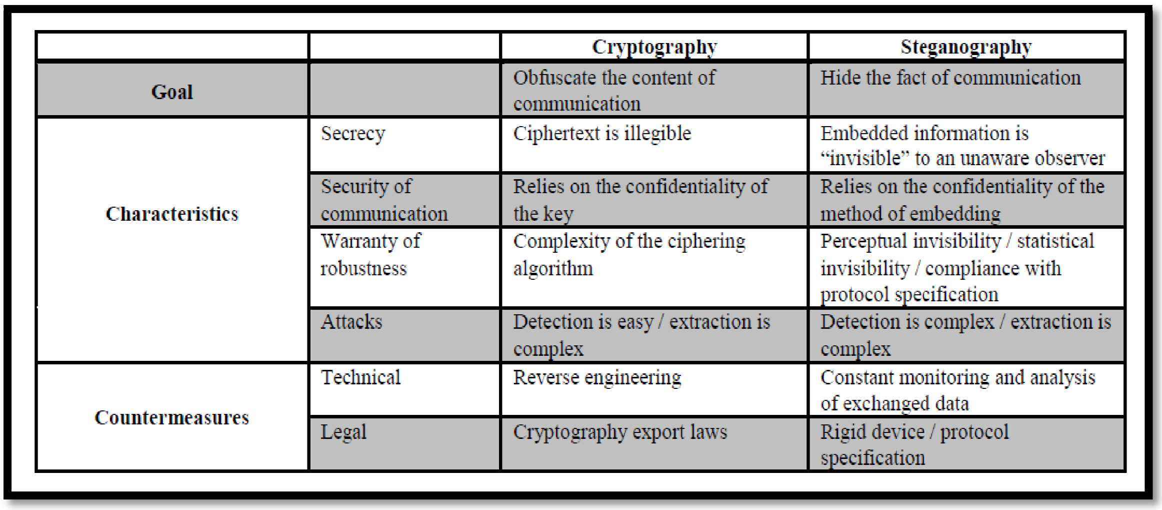
- The secret information must be embedded in a seemingly innocent carrier, which camouflages it.
- The purpose of applying a steganographic technique is to hide the information exchange from a third unaware party.
- The secrecy of the communication is guaranteed primarily by the way the algorithm applied to the carrier camouflages the secret information, which should be good enough to withstand detection attempts.

As for the carrier for the secret messages, we should try to find a carrier that fulfills the next two requirements:

- It should be popular.
- The steganogram insertion-related modifications of the carrier must be invisible to an unaware third party. Some degradation of the carrier caused by the steganographic modifications is allowed, but the severity of this degradation should be limited to a level low enough not to raise suspicion.

We must also note that steganography is confused a lot of times with cryptography, since they both provide confidentiality to the communication. However, they are not the same thing: steganography describes the techniques to create a hidden communication channel, whereas cryptography is a designation of ongoing overt message exchange, where all the information is unintelligible to unauthorized parties.

The table on Figure 1 [2] summarizes the differences between cryptography and steganography.



		Cryptography	**Steganography**
Goal		Obfuscate the content of communication	Hide the fact of communication
Characteristics	Secrecy	Ciphertext is illegible	Embedded information is "invisible" to an unaware observer
	Security of communication	Relies on the confidentiality of the key	Relies on the confidentiality of the method of embedding
	Warranty of robustness	Complexity of the ciphering algorithm	Perceptual invisibility / statistical invisibility / compliance with protocol specification
	Attacks	Detection is easy / extraction is complex	Detection is complex / extraction is complex
Countermeasures	Technical	Reverse engineering	Constant monitoring and analysis of exchanged data
	Legal	Cryptography export laws	Rigid device / protocol specification

Figure 1: Comparison of characteristics of steganography and cryptography [2]

2.2 Recent Cases of Steganography Usage

- Steganographic methods were utilized by Al Qaeda terrorists while planning the attack on USA on 11th of September, 2001 [12, 13]. The organization used images to hide all the instructions related with the attack. These images were posted on publicly available websites, where terrorists could access them (a d the secret information they were carrying) without anyone noticing it [13]. This communication could have passed unnoticed for as long as three years [14].

- In [15], the case of "Operation Twins" is recollected, an operation that ended in 2002 with the capture of the criminal group called "Shadowz Brotherhood", a pedophile organization involved in the distribution of child pornography. They were doing this with the aid of different steganographic methods.

- Steganographic methods have also been used for data exfiltration. In 2008, someone at the U.S. Department of Justice smuggled sensitive financial data out of the agency by embedding the data in several image files [16, 2].

- In 2010, a Russian spy ring (the so called "illegals") was revealed. This group used digital image steganography to leak classified information from USA to Moscow [17], and they did this for years until they were finally discovered.

- In March 2014, a new ZBOT malware appeared [18]. This malware was downloading into different computers what seemed to be an innocent picture of a sunset, but this picture had some steganographically hidden configuration files which main objective was to get bank account information from the victims.

2.3 The origins and evolution of steganography

Steganography is originally insipired in some of the skills displayed by either animals or plants for their survival that ancient people noticed: mimicry, to be more precise. Trying to imitate nature, the Ancient Greeks started to use steganography using ordinary objects to carry their secret messages.

The first written evidence of the usage of a steganographic technique is attributed to Herodotus, a Greek historian. According to him, the secret message was hidden and carried by a hare corpse acting as some kind of game trophy [19]. Other famous Greek techniques include the use of coated wooden tables, carrying carved messages only readable after eliminating the wax coating.

The Greek methods were pretty easy to implement and as the human civilization progressed, new methods started to arise. With the popularization of parchment, sympathetic inks appeared: some saps were used for writing texts that were apparently invisible, but revealed after applying heat to the parchment [20].

Romans brought more progress to steganography: they realized that the secret message didn’t have to necessarily take written form. Consequently, new methods were invented, like the Astragali [21], a small dice made of bone which holes could be used to send a secret message.

More progress was made in the Medieval Ages. With the invention of papper, it started to become apparent that they had to think about some way to differentiate between different manufactures products. In order to solve this problem, watermarking was invented [22]. We should note that, nowadays, digital image steganography base on the same principle.

Another of the inventions that became popular during these times is the acrostic, a textual steganographic method. Acrostic refers to pieces of writing, whose first letters or syllables spell out a message [2]. The most famous example of such textual steganography is attributed to a Dominican priest who, in 1499 hid in one of his books a love confession which could be spelled out from the first letter of each of the chapters of the book [23].

In the XVI century (The Renaissance), an Italian scientist discovered how to hide a message inside a hard-boiled egg: We just need to write it on the shell using ink made from a mixture of alum and vinegar. This solution penetrates the eggshell, being only readable once the shell is removed.

In 1680 a new method was published. It consisted in hiding secret information using the different musical notes as carrier for sending the secret message: each note and their different combinations of the notes forming different chords corresponded to a letter [2]. This method had the problem of someone trying to play the obtained melodies, since they would sound pretty odd.

New steganographic were developed also during the Industrial Revolution. Newspapers became pretty popular, and they started using them as the new steganographic carrier, by making holes over selected letters, writing that way the secret message.

In the period of the two World Wars, and also in the period of the Cold War, many new techniques were developed, like the microdots: Punctuation marks with inserted microscopic negatives of images or texts [24, 2]. New spread spectrum steganographic techniques were also developed in the Second World War for guiding torpedoes [25]. These spread spectrum techniques were later used in the fields of digital media steganography. Some old techniques, like the invisible inks, made a comeback in this period also [26].

Thanks to the technological development, new techniques were developed in the 20th century. Among these inventions we could find the “subliminal channels”, which based on cryptographic ciphers for the embedding of steganograms [2]. Their main principle was to insert content into digital signatures [27].

2.4 Modern steganographic techniques

New modern steganographic techniques utilize the 20th century’s inventions: Mainly, computers and networking. Four main trends of development can be distinguished [2]:

- digital media steganography
- linguistic steganography
- file system steganography
- network steganography

Digital media steganography dates back to the 1970's. In these years, researchers focused on developing methods to secretly introduce some kind of invisible mark or signature into a digital picture. A lot of different methods were proposed [28]. We can classify the different algorithms they implemented for the embedding in digital pictures depending on the alterations introduced by them, being these alterations made on a bit level, or affecting the frequency domain characteristics, or exploiting specific file format intricacies [2]. It is also possible to use a mix of all the mentioned techniques. The transform domain provides for the most versatile medium of embedding. Affecting of the image processing algorithms may involve, among others, discrete cosine transform (DCT), discrete wavelet transform (DWT) or Fourier transform [29, 30].

When using digital image steganography, we are trying to trick our eyes so they perceive an altered image as an unaltered one [28]. The same principle can be applied to the whole field of digital media steganography: what we are doing when using it is to try to trick human senses.

Soon after starting to work with digital image steganography, experts noticed that it was also possible to trick the auditory system. So they stared to work with audio files and developing new techniques, like frequency masking, echo hiding, phase coding, patchwork or spread spectrum. They also noticed that error correction coding was a good supplemental carrier for audio steganography [28].

Next, steganographers took video files as target carrier. Most of the designed methods were just adaptations of the algorithms used for audio and image files, but some video-specific solutions were also designed, involving for example I-frames color space [31] as steganographic carrier or P-frames and B-frames motion vectors.

Alongside digital media steganography, information hiding in text was further developed, with methods that exploited various aspects of the written word, like word-spacing, a technique that was used at the times of Margaret Thatcher [32]. More advanced steganographic methods used syntactic and semantic structure of the text as a carrier: Displacement of punctuation marks, word order changes, etc. It has reached such a complexity that even apparently harmless SPAM messages may be a carrier of steganography nowadays [33]. It was also revealed that x86 machine code could also be used for carrying secret messages by using methods that exploited the same principle as linguistic steganography [34].

The invention of a steganographic file system was a true discovery [35]. It became apparent that information could be hidden even in isolated computing environments. The main principle of steganogram preparation is similar to invisible inks: if you know how to search for it, you will be able to reveal the encrypted files from a disk.

Alongside all the mentioned types of digital steganography, currently the target of increased interest is the fourth mentioned trend of development: Network steganography. We will now concentrate on this network steganography, since for this work we will be using some network steganography's techniques.

2.5 Network Steganography

Just like with the other methods, the main aim of network steganography is to hide secret data in legitimate transmissions of users without destroying the hidden data carrier used. The scope of the network steganography is limited to all information hiding techniques that: (a) can be applied in communication networks to enable hidden data exchange by crating convert communication channel; (b) are inseparably bounded to the transmission process; (c) do not destroy the hidden data carrier.

It must be emphasized that the main difference between "classic" steganography and the one utilized in networks is that the first relied on fooling human senses (as we have seen with the digital media steganography, for example), while the latter tries to deceive network devices (intermediate network nodes or end-user ones).

In network steganography a carrier is at least one network traffic flow. Typically, carrier can be multi-dimensional, i.e. it offers many opportunities for information hiding (called subcarriers). And a subcarrier is defined as a "place" or a timing of "event" (e.g. a header field, padding or intended sequences of packets) in a carrier where secret information can be hidden using single steganographic technique (Figure 2)

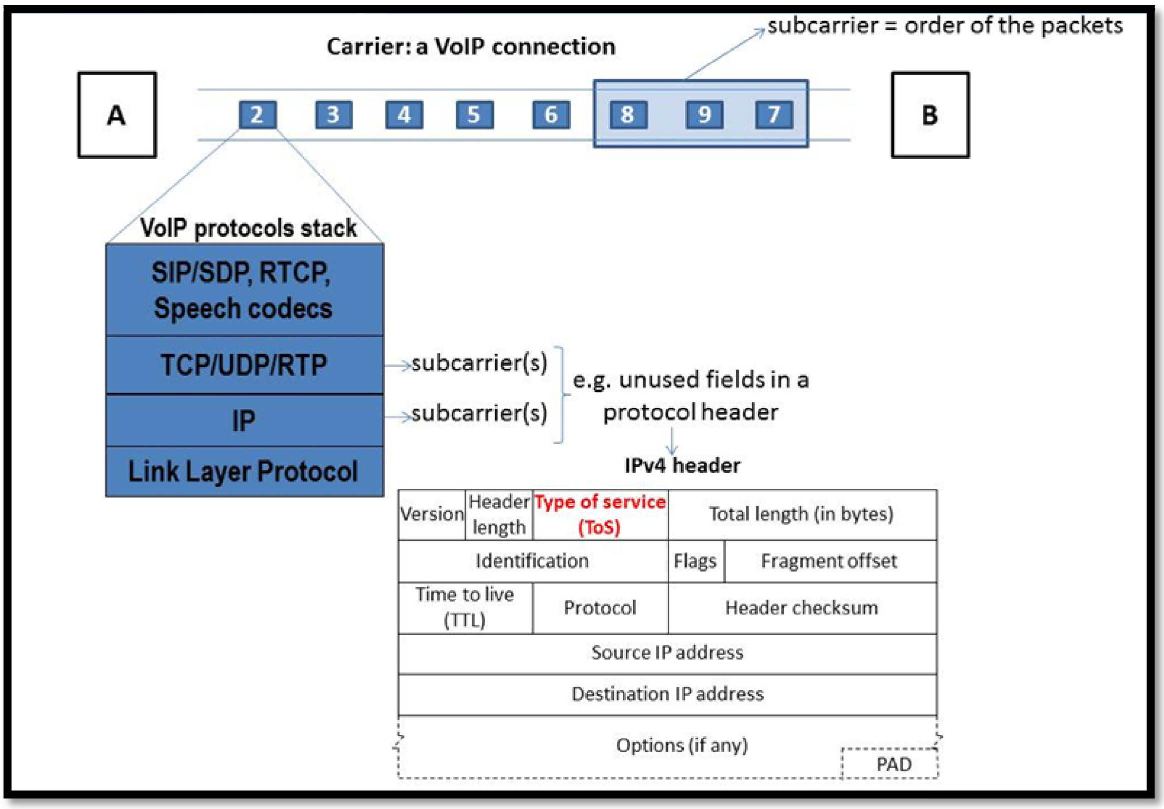


Figure 2: An Example of a carrier and subcarriers based on VoIP connection example.

The most favourable carriers for secret messages in communication networks must have two features:

- Carriers should be popular i.e. usage of such carriers should not be considered as an anomaly itself. The more popular carriers are present and utilized in a network the better, because they mask existence of hidden communication.
- Carrier modification related to embedding of steganogram should not be “visible” to the third party not aware of the steganographic procedure. Contrary to typical steganographic methods which utilize digital media (pictures, audio and video files) as a cover for hidden data, network steganography utilizes network connections i.e. communication protocols’ control elements and their basic intrinsic functionality.

Every network steganographic method can be described typically by the following set of characteristics: its steganographic bandwidth (also referred as capacity typically for media steganography), its undetectability (also referred as security in literature Fridrich[30]), and its robustness. The term “steganographic bandwidth” refers to the amount of secret data that can be sent per unit time when using a particular method. Undetectability is defined as the inability to detect a steganogram within a certain carrier. The most popular way to detect a steganogram is to analyze the statistical properties of the captured data and compare them with the typical values for that carrier. The last characteristic is robustness that is defined as the amount of alteration steganogram can withstand without secret data being destroyed. A good steganographic method should be as robust and hard to detect as possible while offering the highest bandwidth. However it must be noted that there is always a fundamental trade-off among these three measures necessary (Figure 3).

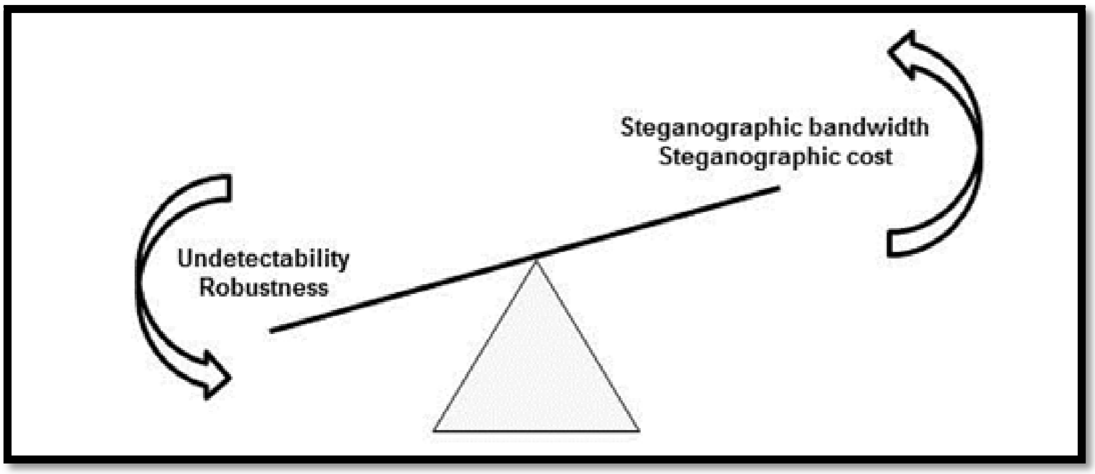


Figure 3: Relationship between characteristics of network steganography.

2.6 Steganographic cost

There is another characteristic that we must take into account when evaluating network steganography methods: the steganographic cost. This characteristic should be taken into account to indicate the degradation or distortion of the carrier caused by the application of the steganographic method (Figure 4). In digital media steganography, i.e. for hiding secret data in digital image, audio, video MSE (Mean-Square Error) or PSNR (Peak Signal-to-Noise Ratio) were utilized for this purpose. However, these parameters cannot be applied to dynamic, diverse carriers like network connections. For example, in the case of VoIP steganography methods, this cost can be expressed, for example, by providing a measure of the conversation quality degradation induced by applying a particular information hiding technique. If certain fields of the protocol header are used as the hidden data carrier, then the cost is expressed as a potential loss in that protocol's functionality. It is also possible that an information hiding method introduces steganographic cost that can be experienced in two different "planes", e.g. it introduces voice quality degradation as well as it adds additional delays to the overt traffic.

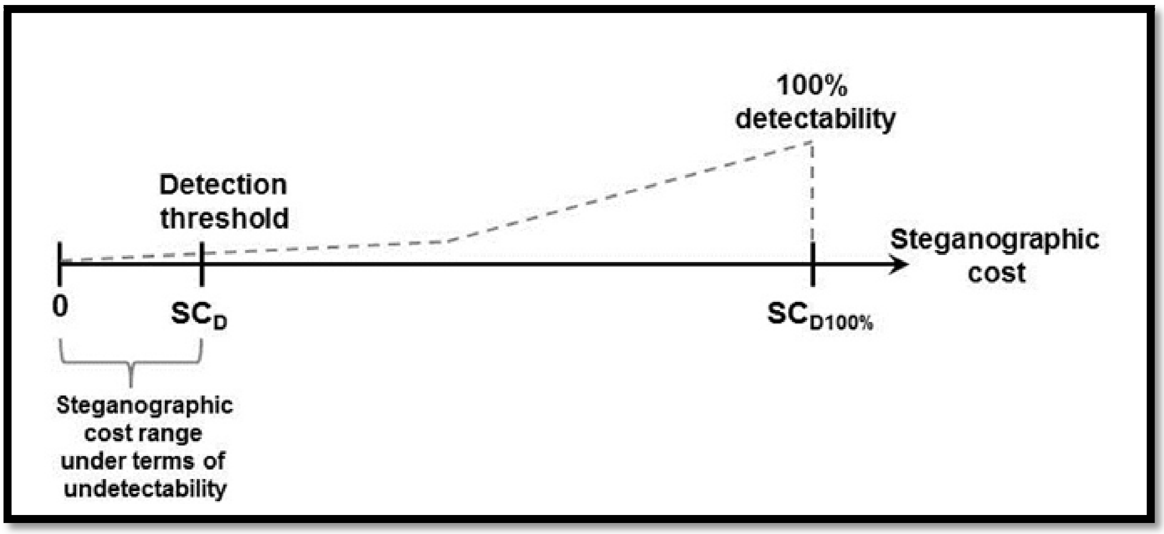


Figure 4: Relationship between steganographic cost and undetectability.

Therefore in general we can conclude that steganographic cost affects undetectability and may be responsible for loss of carrier's functionality or loss of carrier's performance (e.g. it results in longer connection or increased resources usage). The relationship between steganographic cost and undetectability is explained in Fig. 3. One can imagine a steganographic cost as a "zip" as it provides a view on how exactly the carrier was affected by applying steganographic method. On the other hand undetectability can be imagined as a "switch" i.e. when the certain level of steganographic cost is exceeded (SC_D) then the steganographic method becomes detectable up to the point where the detection is trivial ($SC_{D100\%}$).

The effects of steganographic cost are threefold and form a vector for each steganographic method. Some steganographic methods affect the detectability, while others affect the feature spectrum or the performance of the carrier. Others affect multiple aspects simultaneously but in different extent. Besides splitting performance and feature cost, both could be combined to functionality cost in order to achieve a twofold view of steganographic cost.

Steganographic cost can also be important when more than one method is applied to the same hidden data carrier. In this case, steganographic cost allows observing the relationships between steganographic methods applied to the same hidden data carrier, and some of these relationships might be really interesting. For example, despite general considerations, we think that is possible that two or more methods applied simultaneously to the same carrier affect each other in such a way that the resulting total cost is lower than the overall cost of these two methods when applied alone. We will be calling this situation superposition steganography. Moreover, we think that there can also be a special case of superposition steganography (that we will be calling zero cost steganography) which occurs if at least one steganographic method is applied to another steganographic method in a way that it results in no additional cost.

In this work, what we are going to do is to design two different steganographic scenarios we thought about that will help us to prove if these superposition steganography and zero cost steganography cases are feasible or not. Let us proceed to explain this in more detail.

3. CONSIDERED STEGANOGRAPHIC SCENARIOS FOR IP NETWORKS

In this thesis we intend to implement two different scenarios that will be helping us to prove the existence of the two terms we mentioned before: superposition steganography and zero cost steganography.

3.1 Scenario 1: IP Fragmentation

Let us consider a case where two simple steganographic methods F1 and F3 as defined in [36] are applied to an IP-based traffic flow.

F1 method relies on the parity of the number of fragments that the packet was divided into. SS (Steganogram Sender) is the source of the fragmentation and controls this process. SS inserts single bit of hidden data by dividing each of IP packets into the predefined number of fragments. For example, if the number of fragments is even then it means that binary 0 is transmitted and in other case binary 1 (Figure 5). The hidden data extraction is obvious as after the fragments reception SR utilizes the number of the fragments of each received IP packet to determine what hidden data was sent.

Of course if the statistical steganalysis based on number of fragments is performed to detect irregularities in number of each packet's fragments the F1 method is not hard to detect.

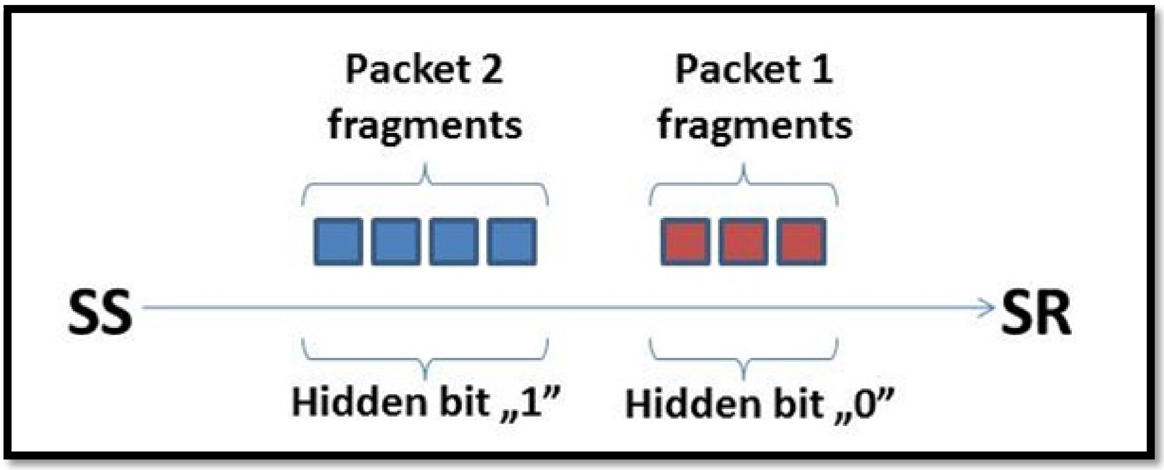


Figure 5: F1 Steganographic method example.

The second method (F3) utilizes legitimate fragments with steganogram inserted into payload for higher steganographic bandwidth and harder detection. SS is the source of the fragmentation and controls the process. During the fragmentation SS inserts secret data instead of inserting user data into the payload of selected fragments.

To make the steganographic fragments distinguishable from others yet hard to detect the following procedure was introduced. If SS and SR share a secret Steg-Key (SK) then for each fragment chosen for steganographic communication the following hash function (H) is used to calculate Identifying Sequence (IS):

$$IS = H(SK\ ||\ Fragment\ Offset\ ||\ Identification)$$

Where Fragment Offset and Identification denote values from these IP fragment header fields and ||bits concatenation function. For every fragment used for hidden communication the resulting IS will have different value due to the values change in a Fragment Offset field. All IS bits or only selected ones are distributed across the payload field in predefined manner. Thus, for each fragment SR can calculate appropriate IS and verifies if it contains secret or user data. If the verification is successful then the rest of the payload is considered as hidden data and extracted. Then SR does not utilize this fragment in reassembly process of original IP packet.

Figure 6 illustrates an example of the proposed steganographic method. IP packet with ID 345 is divided into four fragments (FR1-FR4). Fragment FR2 is used for steganographic purposes, so inside its payload secret data is inserted together with correct IS. Values in Fragment Offset and identification fields remain the same as in other legitimate fragments. While reassembling original packet, SR merges payloads P1, P2 and P3, omits fragment F2 and use it only to extract secret data.

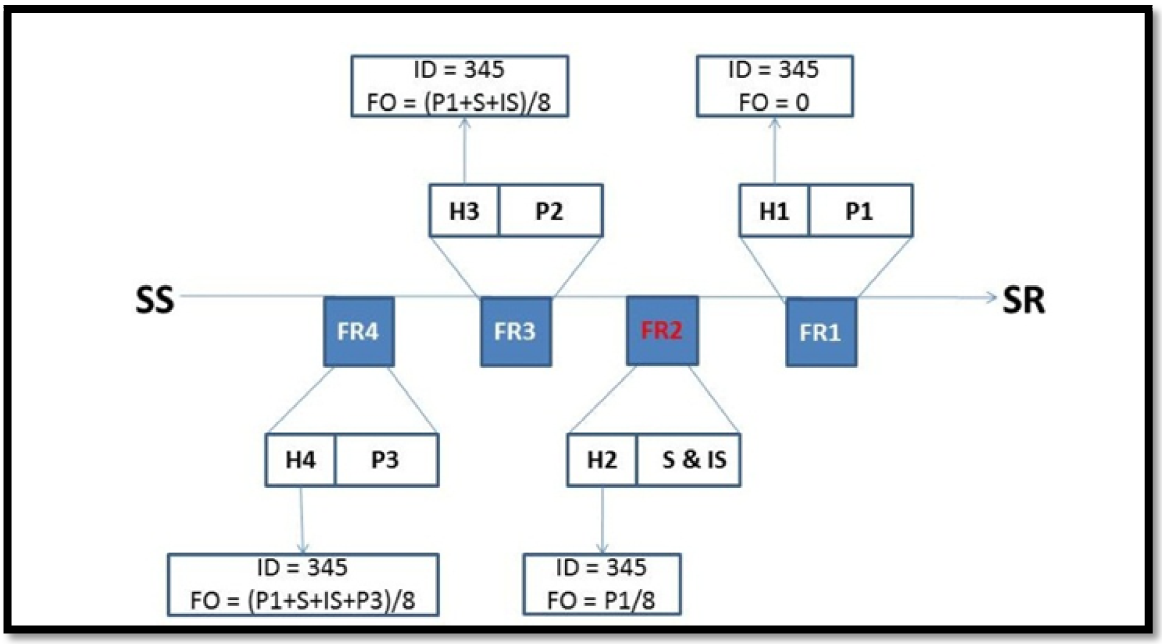
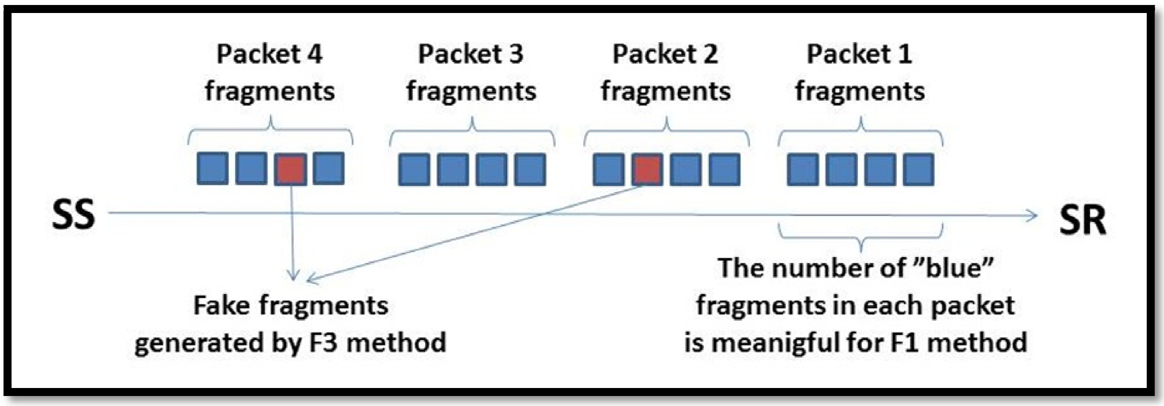


Figure 6: F3 Steganographic method example (H-header, P-payload, S-secret data).

Now let us consider the case when combined F1 and F3 methods are applied simultaneously to the same hidden data carrier – IP-based traffic flow (Figure 7). Since F1 modulates a number of fragments that the packet is divided into and F3 inserts fake fragments then if used combined the total steganographic cost will decrease and undetectability. This is what we call superposition steganography.



Packet 4 fragments
Packet 3 fragments
Packet 2 fragments
Packet 1 fragments
SS
SR
Fake fragments generated by F3 method
The number of "blue" fragments in each packet is meanigful for F1 method

Figure 7: Simultaneous utilization of F1 and F3 methods.

So, for this first scenario, we will have two virtual machines, PC-A and PC-B. They will be connected between them, with PC-B acting as the steganogram sender (SS), and PC-A as the steganogram receiver (SR).PC-B will be sending IP traffic (we will be specifying what kind of traffic later) directed to PC-A, and alongside this normal IP traffic we will be sending some hidden information making use of IP fragmentation. Then, we will be studying how steganographic cost is affected in four different cases:

- Case 1: No steganography case, where we will be applying no steganographic method over the information transference that happens between both PCs (in other words, we will not be sending secret information in this case).
- Case 2: F1 method case, where we will use F1 steganographic method over the IP traffic in order to send some secret information from PC-B to PC-A.
- Case 3: F3 method case, where we will use F3 steganographic method over the IP traffic in order to send some secret information from PC-B to PC-A.
- Case 4: F1+F3 methods case, where we will use simultaneously both F1 and F3 steganographic methods over the IP traffic in order to send some secret information from PC-B to PC-A.

We will compare the obtained results for each of the cases in order to prove the existence of the superposition steganography phenomena.

3.2 Scenario 2: HTTP Headers

We will now consider a case where three simple steganographic methods F6, F7 and F8 are applied to an HTTP-based traffic flow.

In order to steganographically send the secret information, method F6 changes the case of the different HTTP header fields, F7 changes the order of the header fields, and F8 changes the total number of header fields. Only method F8 introduces steganographic cost by increasing the header's size and thus the available space for the remaining payload. Methods F6 and F7 modify the already created header elements of F8 without degrading the functionality of the protocol or the performance of the request. Thus, methods F6 and F7 add no additional steganographic cost, or are, in other words zero cost methods in combination with F8. We illustrate the example in Figure 8:

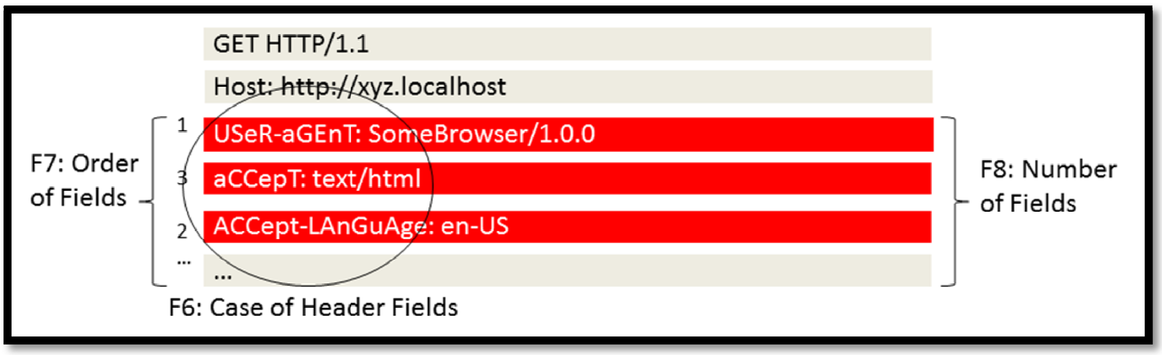


Figure 8: Three steganographic methods applied to the HTTP Request header.

In this second scenario, we will have again 2 virtual machines (PC-A and PC-B) directly connected between them. This time, PC-A will be acting as the steganogram sender (SS) and PC-B as the steganogram receiver. PC-B will be acting as a WEB Server, with PC-A as a client machine that will constantly send HTTP Requests to the server. Of course, steganographically hidden information will be sent on these HTTP Requests' headers. Here, we will be analyzing how steganographic cost is affected in five different cases:

- Case 1: No steganography case, where we will be applying no steganographic method over the information transference that happens between both PCs (in other words, we will not be sending secret information in this case).
- Case 2: F6 method case, where we will use F6 steganographic method over the HTTP traffic in order to send some secret information from PC-A to PC-B.
- Case 3: F7 method case, where we will use F7 steganographic method over the HTTP traffic in order to send some secret information from PC-A to PC-B.
- Case 4: F8 method case, where we will use F8 steganographic method over the HTTP traffic in order to send some secret information from PC-A to PC-B.
- Case 5: F6+F7+F8 methods case, where we will use simultaneously F6, F7 and F8 steganographic methods over the HTTP traffic in order to send some secret information from PC-A to PC-B.

We will compare the obtained results for these five cases in order to prove the existence of the zero cost steganography phenomena.

3.3 Procedure

The total work for the realization of this thesis can be divided into four clear different parts:

- 1) Preparation of the scenarios: installation and preparation of the virtual machines.

  The first thing that we must do is to correctly install the virtual machines we are going to use for the implementation of the thesis's scenarios, as well as all of the programs and instruments we are going to need in each virtual machine. We will be using virtual machines working under Ubuntu's Linux operating system. We will also be needing to install some applications like Wireshark (required for traffic analysis), g++ (because we are going to be working with C++ and we will be needing the compilator), and at least in one of the virtual machines we will have to install and correctly configure one HTTP server, so we can have HTTP traffic between the virtual machines. With the correct configuration of the virtual network that connects our machines, we will have ended the first part of the master thesis.

- 2) Programming of each of the 9 cases.

  Once we have our scenarios correctly implemented, the next thing we will have to do is to prepare each of the nine study cases (previously mentioned in Sections 3.1 and 3.2). In other words, we will have to implement the different steganographic methods. Since we will have to manipulate some of the packet's information (like the headers in HTTP, or the number of fragments when working with IP fragmentation) before sending them from one computer to the other in order to send the hidden information, we will need to use IP-Tables, a tool which will allow us to change the packets' information before sending them. Combining IP-Tables with C++ programming, we will implement all of the study cases.

- 3) Simulation, analysis and evaluation of the results.

  Once we have each case programmed, we will have to try them (around 10-20 times each, so we can calculate mean values and standard deviations) and then to analyze the obtained results with the help of Wireshark, looking how steganographic cost is affected on each case. We will then proceed to compare the results in order to prove our hypothesis about super-position steganography and zero-cost steganography.

- 4) Conclusions.

  After analyzing all the results and comparing the cases between them, we will conclude the work giving our final conclusions.

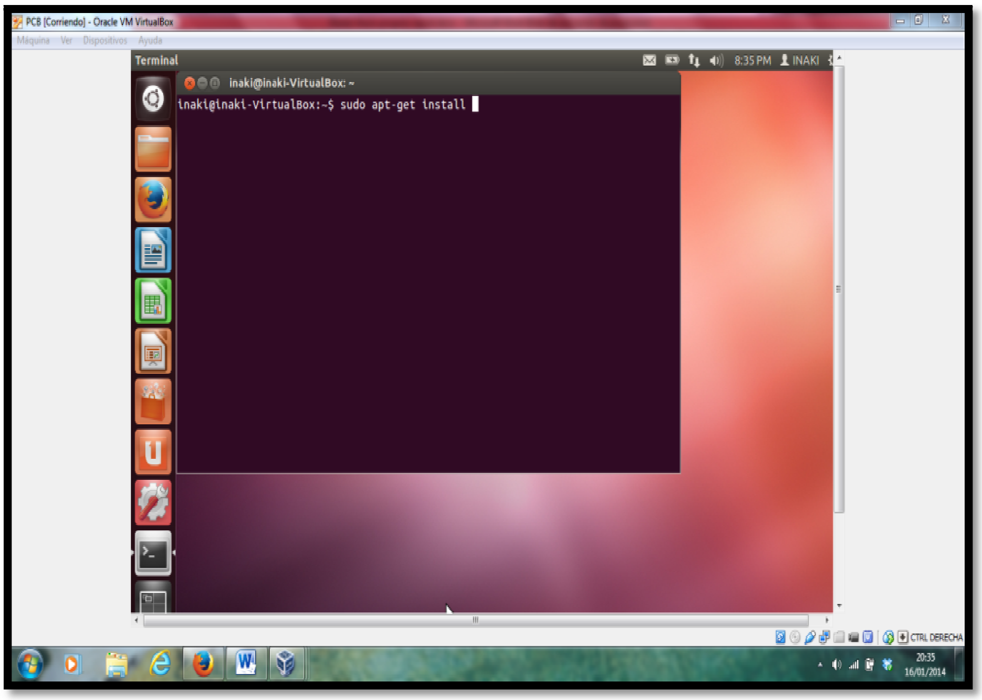
We will now proceed to describing total work in detail.

4. EXPERIMENTAL TEST-BED SETUP

4.1 Installation of the machines

The first thing we have to do is to install the two virtual machines we are going to work with. We decided to use free software VirtualBox, a program that allows the user to easily create virtual machines, giving them the choice to control things such as the memory dedicated to the virtual machine, the operative system running in the machine or the different connections and interfaces that the machine has.

After installing the program, we proceed to create the two virtual machines. In our case, we named them PC-A and PC-B. As for the operative system, we chose to use Ubuntu (Linux) in both virtual machines. The main reason for it is that we find it easier to work with programs like Wireshark, or to download programming libraries when using Linux than when using Windows for example. We chose Ubuntu because it is the Linux operative system that we are more familiar with.



```
inaki@inaki-VirtualBox:~$ sudo apt-get install
```

Figure 9: Virtual machine PCB using Ubuntu operating system, version 12.04.

After installing Ubuntu, we will also have to install the programs and libraries we are going to need, like Wireshark or g++ compiler. This is easily done by using *apt-get* command.

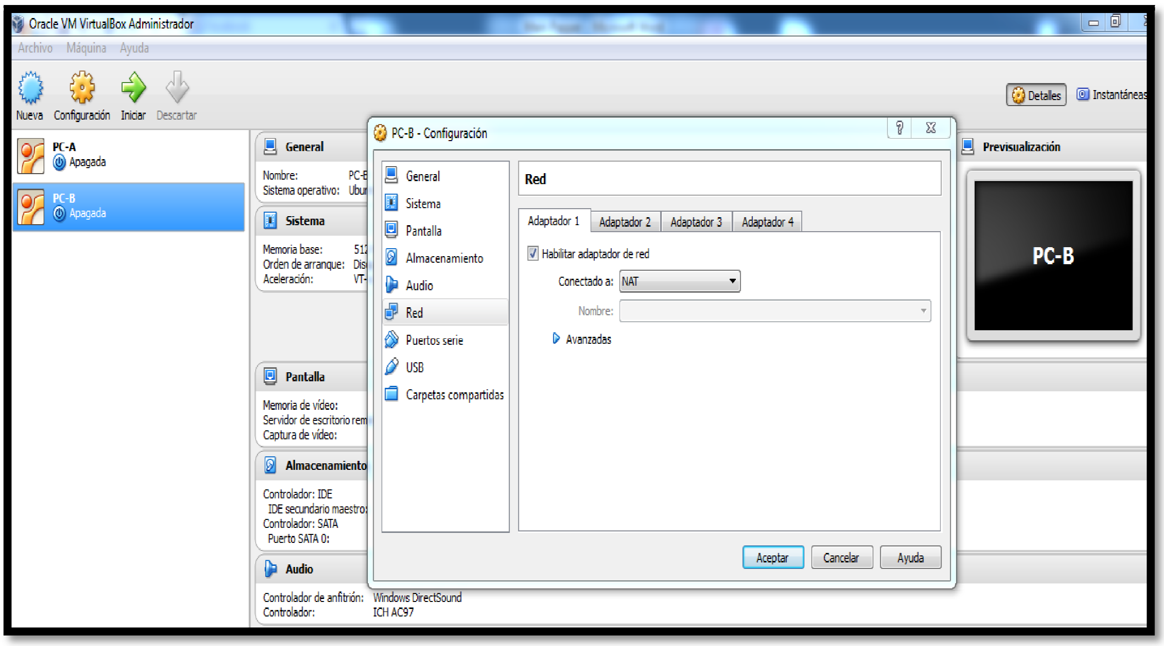
We must note one important thing here: not every Ubuntu version is valid for this work. At first we tried to use version 10.10, but this version is no longer supported by Cannonical Ltd. Enterprise, which means that we were trying to work with a version that had no longer technical support nor updates. Because of this, for example, something as simple as to use command “apt-get” to install new programs wasn’t working, since it was no longer possible to correctly access and to download some files and programs from Ubuntu’s servers. We have to work with a supported version in order to avoid this kind of problems, like the one we ended up choosing: version 12.04.

In the future, if someone wants to repeat this work he will have to take into account that the version we ended up using (12.04, supported until 2017) might not be suitable for the work by then.

4.2 Installation of the virtual network

After preparing the virtual machines, the next thing we have to do is to make the virtual network connecting them. Each of the virtual machines has two interfaces: One of these interfaces will be directly connected to the internet, and will be obtaining its IP address via DHCP protocol. The other interface will be connected to an intern network common to both machines, providing a direct communication channel between them. This interfaces connected to the intern network will be having fixed IP addresses, masks and gateways.

This entire network configuration is pretty easy to do: after initiating VirtualBox program, we have to select one of the virtual machines and then, without initiating the machine, we access to the configuration menu. Inside this menu, going to the network section, we are able to habilitate as many interfaces as we want: We have to habilitate two of them, one connected to the internet (NAT) and another one connected to an intern network. We must do this again with the other virtual machine.



☑ Habilitar adaptador de red
Conectado a: NAT

Controlador de anfitrión: Windows DirectSound
Controlador: ICH AC97

Figure 10: Activation of the interfaces in VirtualBox.

After activating both interfaces on each machine, we must initiate the machines and correctly change the interfaces' configuration: we will be fixating IP addresses for the interfaces connected to the intern network. The ones connected to the internet will be using DHCP protocol to obtain their IP address. This is illustrated in the next two figures (Figure 11 and Figure 12).

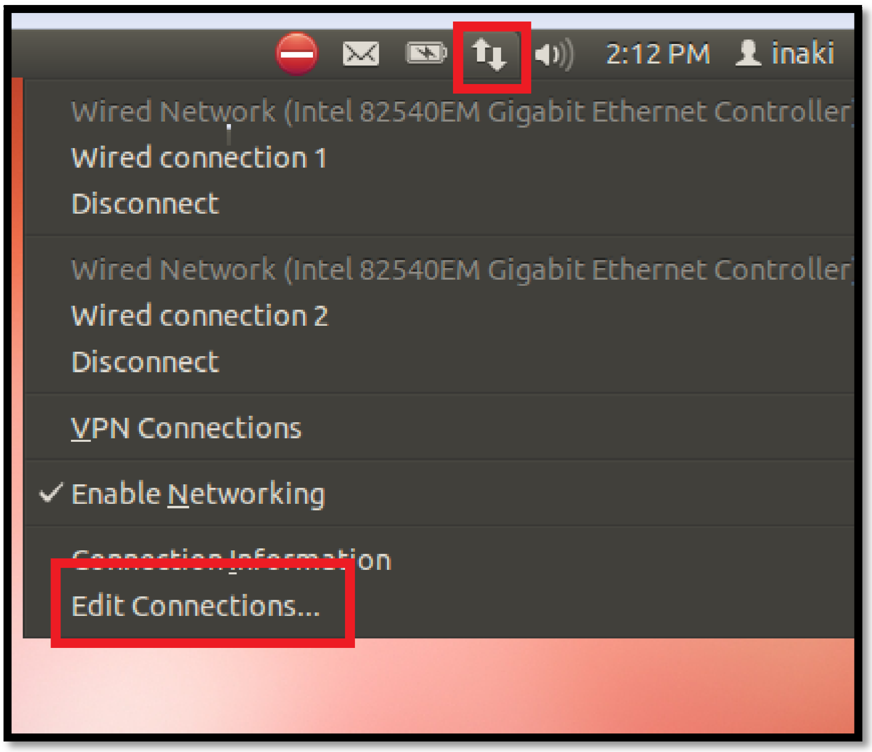
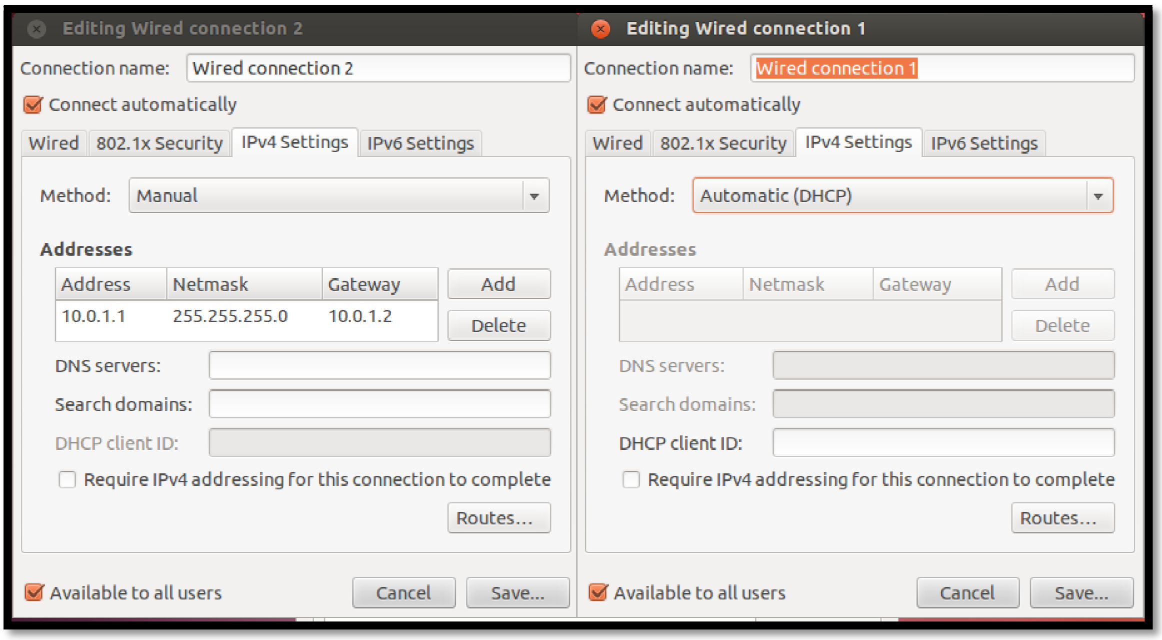


Figure 11: Network interface's configuration (I).



Editing Wired connection 2

Connection name: Wired connection 2

✓ Connect automatically

Wired | 802.1x Security | IPv4 Settings | IPv6 Settings

Method: Manual

Addresses

Address	Netmask	Gateway
10.0.1.1	255.255.255.0	10.0.1.2

DNS servers:

Search domains:

DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

✓ Available to all users

Editing Wired connection 1

Connection name: Wired connection 1

✓ Connect automatically

Wired | 802.1x Security | IPv4 Settings | IPv6 Settings

Method: Automatic (DHCP)

Addresses

Address	Netmask	Gateway

DNS servers:

Search domains:

DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

✓ Available to all users

Figure 12: Network interface's configuration (II).

5. STEGANOGRAPHIC SCENARIOS AND METHODS IMPLEMENTATION

We will now proceed to describe how each of the steganographic methods used in this Master thesis was implemented. In this section, we will be fairly descriptive: we will focus on describing how the methods were implemented and how the problems encountered were tackled, without making any reference to the implemented programming code. Related programming code can be found in Annex 1.

5.1 IP TABLES

In order to capture and manipulate the packets, we are going to use netfilter/IP tables.

IP tables is a user space application program that allows a system administrator to configure the tables provided by the Linux kernel firewall (implemented as different Netfilter modules) and the chains and rules it stores. Different kernel modules and programs are currently used for different protocols; iptables applies to IPv4, ip6tables to IPv6, arptables to ARP, and ebtables to Ethernet frames.

iptables requires elevated privileges to operate and must be executed by user root, otherwise it fails to function. On most Linux systems, iptables is installed as /usr/sbin/iptables and documented in its man pages which can be opened using man iptables when installed. It may also be found in /sbin/iptables, but since iptables is more like a service rather than an "essential binary", the preferred location remains /usr/sbin.

All packets passing through a host are traversing what we call iptables chains (see figure below). There are three main types of these chains:

• INPUT – chain for incoming (received) packets that are intended for a process running on the local machine

• OUTPUT – chain for packets that are being sent from a process running on the local machine

• FORWARD – chain for packets that are being forwarded through the host (from one network interface to another)

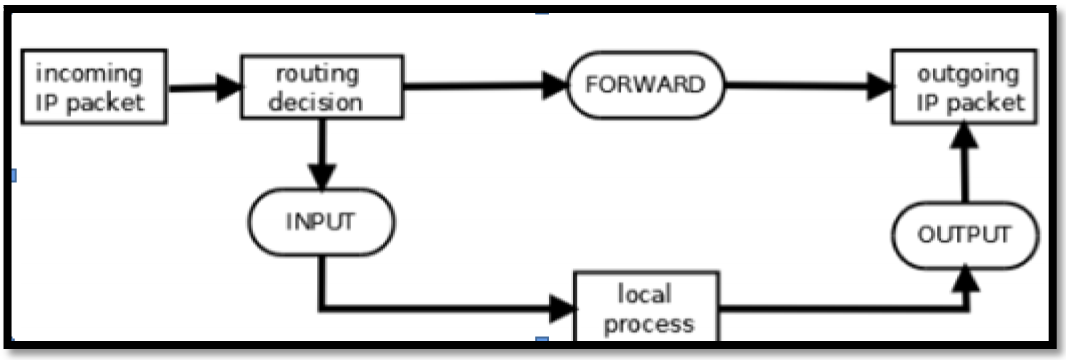


Figure 13: IP Tables Chains.

All our programs have a common part, which consists in a GTK graphic interface that allows us to control the different chains with which IP Tables work. While traversing the iptables chains, packets are in the kernel space, which is inaccessible from the user space. Because of that, to perform modifications, packets from selected chains are put to a special QUEUE chain. This chain does not appear in the normal/usual packet traversing paths. Subsequently, the libnetfilter_queue module (the netfilter queue library mentioned in the previous section) is used to send packets one by one to the user space, where packet modifications take place. Following the modifications, packets are returned to the QUEUE and continue traversing the iptables chains. We illustrate all this in the next figure.

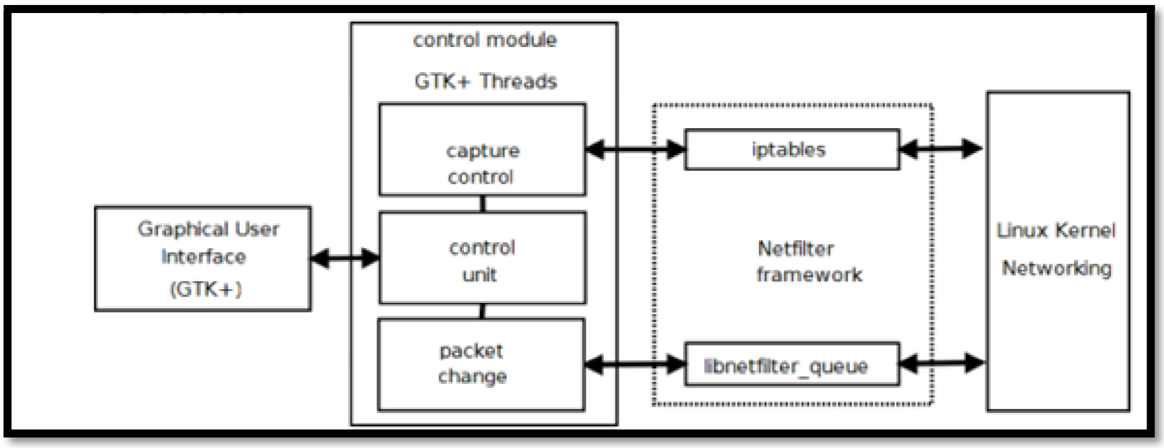


Figure 14: IP Tables working scheme.

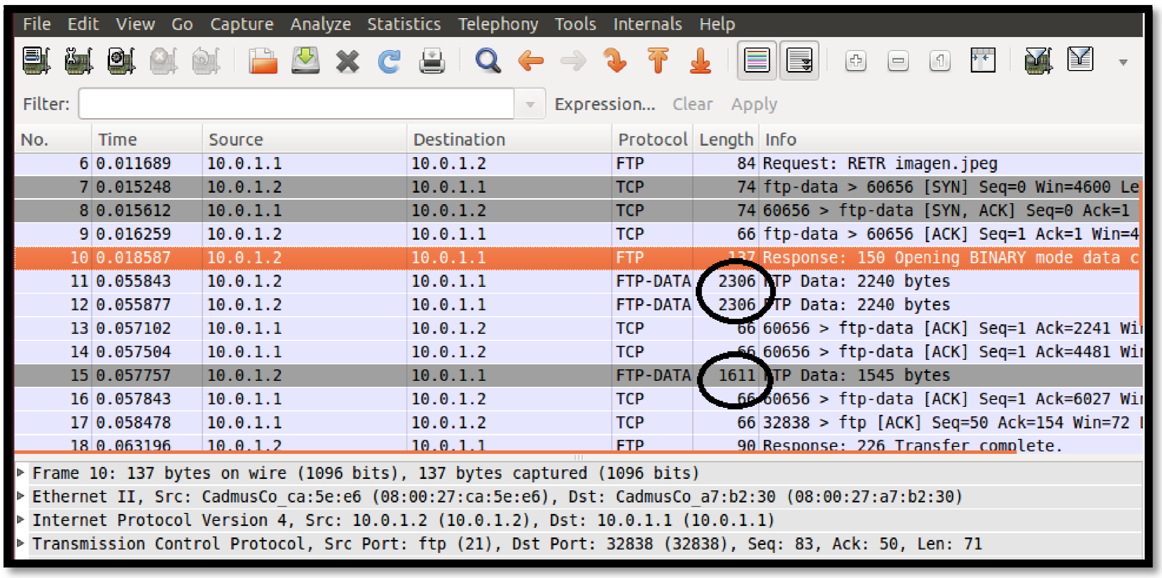
Lets explain now how must we modify the packets for each steganographic case.

5.2 IP FRAGMENTATION-RELATED METHODS

We will start by explaining all the methods related with the IP- fragmentation scenario. Before explaining each one of them, we will be clarifying the kind of IP traffic that we are going to work with.

We decided to use ICMP echo requests-replays carrying random data for this experiment.

This might seem to be a strange decision, since ICMP isn't even normally used for sending user information. The use of other kind of traffic, like for example FTP traffic, seems like a better option. And this was actually our first idea: to use an FTP connection to send a file over which we would be introducing the hidden information steganographically. The main reason for using ICMP echo requests-replays instead of an FTP connection, as originally planned, is that when using an FTP connection, for unknown reasons it wasn't possible to analyze how IP fragmentation was affected, since Wireshark only presents the full FTP-Data packets sent before any kind of fragmentation, as you can see in the figure bellow.

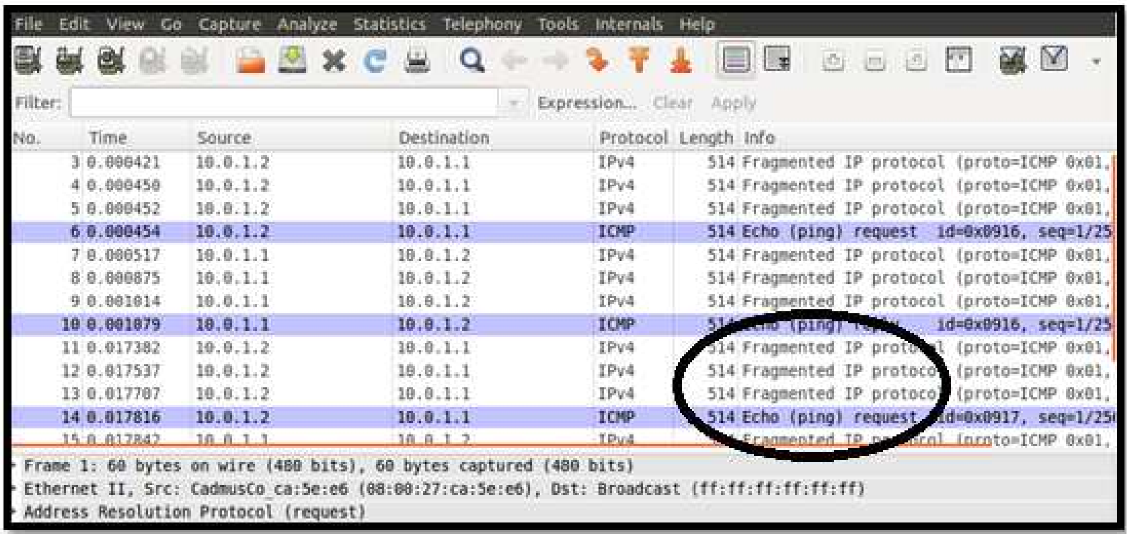

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

```
No.   Time      Source    Destination Protocol Length Info
    6 0.011689  10.0.1.1  10.0.1.2    FTP          84 Request: RETR imagen.jpeg
    7 0.015248  10.0.1.2  10.0.1.1    TCP          74 ftp-data > 60656 [SYN] Seq=0 Win=4600 Le
    8 0.015612  10.0.1.1  10.0.1.2    TCP          74 60656 > ftp-data [SYN, ACK] Seq=0 Ack=1
    9 0.016259  10.0.1.2  10.0.1.1    TCP          66 ftp-data > 60656 [ACK] Seq=1 Ack=1 Win=4
   10 0.018587  10.0.1.2  10.0.1.1    FTP         137 Response: 150 Opening BINARY mode data c
   11 0.055843  10.0.1.2  10.0.1.1    FTP-DATA   2306 FTP Data: 2240 bytes
   12 0.055877  10.0.1.2  10.0.1.1    FTP-DATA   2306 FTP Data: 2240 bytes
   13 0.057102  10.0.1.1  10.0.1.2    TCP          66 60656 > ftp-data [ACK] Seq=1 Ack=2241 Wi
   14 0.057504  10.0.1.1  10.0.1.2    TCP          66 60656 > ftp-data [ACK] Seq=1 Ack=4481 Wi
   15 0.057757  10.0.1.2  10.0.1.1    FTP-DATA   1611 FTP Data: 1545 bytes
   16 0.057843  10.0.1.1  10.0.1.2    TCP          66 60656 > ftp-data [ACK] Seq=1 Ack=6027 Wi
   17 0.058478  10.0.1.1  10.0.1.2    TCP          66 32838 > ftp [ACK] Seq=50 Ack=154 Win=72 
   18 0.063196  10.0.1.2  10.0.1.1    FTP          90 Response: 226 Transfer complete.

▸ Frame 10: 137 bytes on wire (1096 bits), 137 bytes captured (1096 bits)
▸ Ethernet II, Src: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6), Dst: CadmusCo_a7:b2:30 (08:00:27:a7:b2:30)
▸ Internet Protocol Version 4, Src: 10.0.1.2 (10.0.1.2), Dst: 10.0.1.1 (10.0.1.1)
▸ Transmission Control Protocol, Src Port: ftp (21), Dst Port: 32838 (32838), Seq: 83, Ack: 50, Len: 71
```

Figure 15: Wireshark capture of an FTP file transfer with interface MTU=500: Full packets are presented, and it's impossible to analyze fragmentation.

This problem disappears when using ICMP traffic: when using ICMP traffic, it's possible to see all of the packet's fragments on Wireshark, and because of it we decided to use ICMP traffic for the experiments (since it's vital for our thesis to be able to capture the fragments for their analysis).


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

```
No.   Time      Source     Destination  Protocol Length Info
    3 0.000421  10.0.1.2   10.0.1.1     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
    4 0.000450  10.0.1.2   10.0.1.1     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
    5 0.000452  10.0.1.2   10.0.1.1     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
    6 0.000454  10.0.1.2   10.0.1.1     ICMP        514 Echo (ping) request  id=0x0916, seq=1/25
    7 0.000517  10.0.1.1   10.0.1.2     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
    8 0.000875  10.0.1.1   10.0.1.2     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
    9 0.001014  10.0.1.1   10.0.1.2     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
   10 0.001079  10.0.1.1   10.0.1.2     ICMP        514 Echo (ping) reply    id=0x0916, seq=1/25
   11 0.017382  10.0.1.2   10.0.1.1     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
   12 0.017537  10.0.1.2   10.0.1.1     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
   13 0.017707  10.0.1.2   10.0.1.1     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
   14 0.017816  10.0.1.2   10.0.1.1     ICMP        514 Echo (ping) request  id=0x0917, seq=1/25
   15 0.017842  10.0.1.1   10.0.1.2     IPv4        514 Fragmented IP protocol (proto=ICMP 0x01,
▸ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▸ Ethernet II, Src: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)
```

Figure 16: ICMP Packets captured: It’s possible to analyze IP Fragmentation.

5.2.1 No Steganography

As we already explained, we decided to work with ICMP Echo Request/Replies for this scenario, sending packets that carry a total amount of 1432 bytes of data (random data generated by ourselves in this case).

This might seem to be a strange choice. We chose to send packets carrying 1432 bytes of data because of the network MTU and the way we want it to divide our packets: if we add to this data the 8-byte ICMP header and the 20-byte IP header, we have a packet which total size equals 1460 bytes. Because of the MTU of 500 bytes we decided to fixate (it is fixated to that number when starting the programs), this packet is divided into three fragments, and each one of the fragments carries their own 20-byte IP-Header. As a result, we end up having 3 fragments equally sized, with a size equal to 500 bytes (not taking into account the Ethernet headers in any moment). We illustrate this in the figure below.

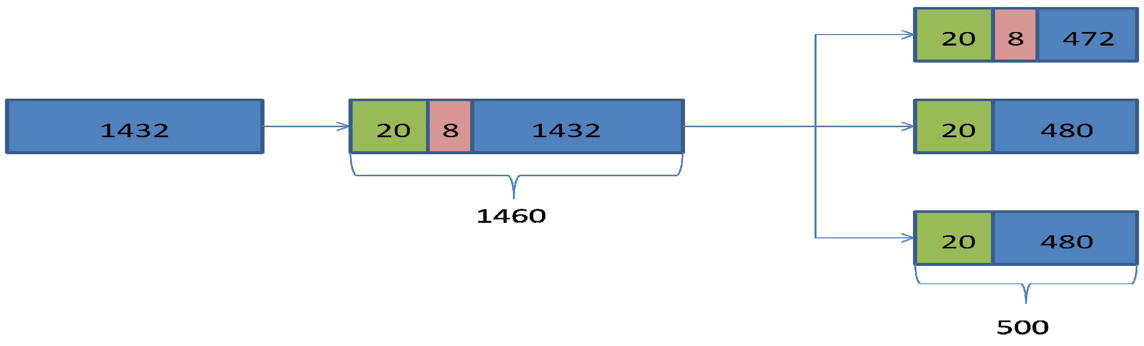
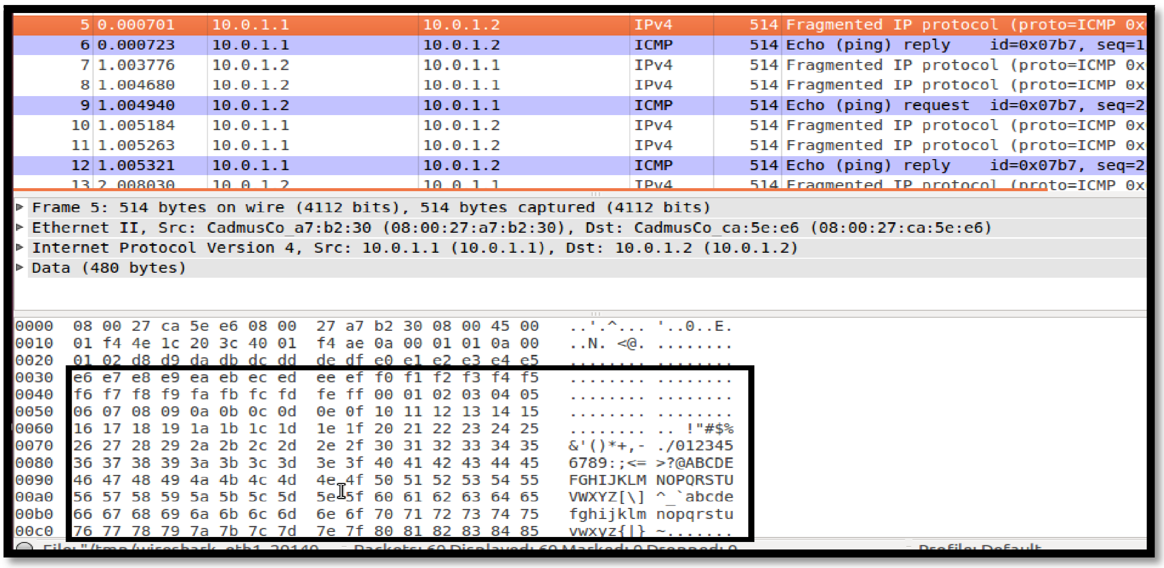


Figure 17: Fragmentation of a 1432 byte-length packet.

So, for the No Steganography case (described in Section 3.1), we have a script prepared that directly send ICMP ping requests with this packet size of 1432 bytes. We capture the pings with IP tables before sending them into the network and then we just change all the data they are originally carrying by the random data stream generated by ourselves. This is done in order for the traffic to look closer to what real traffic would look like, since the data carried originally by the ICMP packets consists just in a cyclic numeric progression (0x00, 0x01, 0x02, …, 0xFF; repeated as many times as needed), which is a pretty unrealistic representation of what real data would look like. We can see the difference in the next 2 pictures.

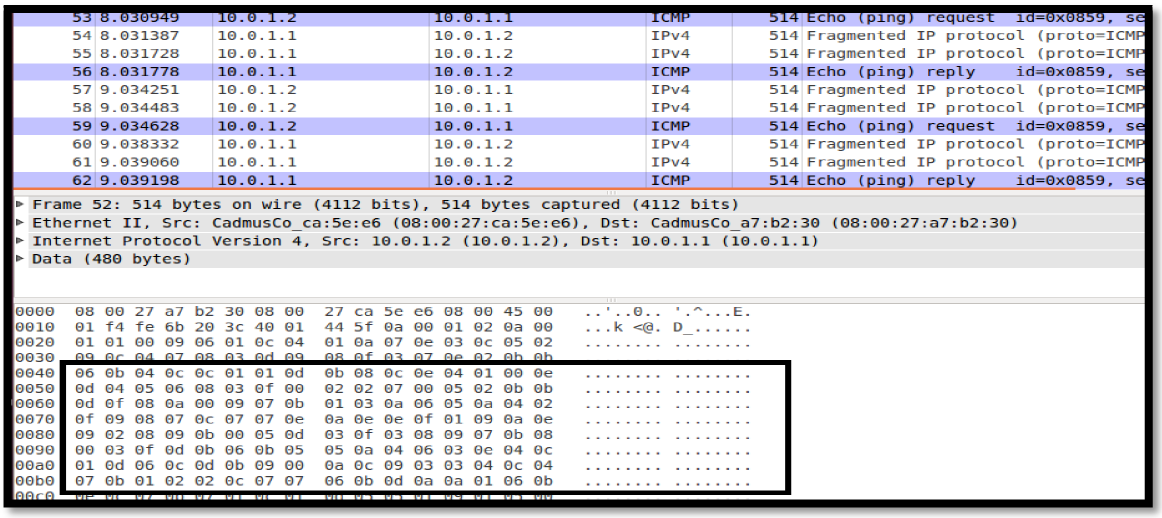


```
 5 0.000701   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP 0x
 6 0.000723   10.0.1.1   10.0.1.2   ICMP   514 Echo (ping) reply    id=0x07b7, seq=1
 7 1.003776   10.0.1.2   10.0.1.1   IPv4   514 Fragmented IP protocol (proto=ICMP 0x
 8 1.004680   10.0.1.2   10.0.1.1   IPv4   514 Fragmented IP protocol (proto=ICMP 0x
 9 1.004940   10.0.1.2   10.0.1.1   ICMP   514 Echo (ping) request  id=0x07b7, seq=2
10 1.005184   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP 0x
11 1.005263   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP 0x
12 1.005321   10.0.1.1   10.0.1.2   ICMP   514 Echo (ping) reply    id=0x07b7, seq=2
13 2.008030   10.0.1.2   10.0.1.1   IPv4   514 Fragmented IP protocol (proto=ICMP 0x

▶ Frame 5: 514 bytes on wire (4112 bits), 514 bytes captured (4112 bits)
▶ Ethernet II, Src: CadmusCo_a7:b2:30 (08:00:27:a7:b2:30), Dst: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6)
▶ Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.1.2 (10.0.1.2)
▶ Data (480 bytes)

0000  08 00 27 ca 5e e6 08 00  27 a7 b2 30 08 00 45 00   ..'.^... '..0..E.
0010  01 f4 4e 1c 20 3c 40 01  f4 ae 0a 00 01 01 0a 00   ..N. <@. ........
0020  01 02 d8 d9 da db dc dd  de df e0 e1 e2 e3 e4 e5   ........ ........
0030  e6 e7 e8 e9 ea eb ec ed  ee ef f0 f1 f2 f3 f4 f5   ........ ........
0040  f6 f7 f8 f9 fa fb fc fd  fe ff 00 01 02 03 04 05   ........ ........
0050  06 07 08 09 0a 0b 0c 0d  0e 0f 10 11 12 13 14 15   ........ ........
0060  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0070  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0080  36 37 38 39 3a 3b 3c 3d  3e 3f 40 41 42 43 44 45   6789:;<= >?@ABCDE
0090  46 47 48 49 4a 4b 4c 4d  4e 4f 50 51 52 53 54 55   FGHIJKLM NOPQRSTU
00a0  56 57 58 59 5a 5b 5c 5d  5e 5f 60 61 62 63 64 65   VWXYZ[\] ^_`abcde
00b0  66 67 68 69 6a 6b 6c 6d  6e 6f 70 71 72 73 74 75   fghijklm nopqrstu
00c0  76 77 78 79 7a 7b 7c 7d  7e 7f 80 81 82 83 84 85   vwxyz{|} ~.......
File: "/tmp/wireshark_eth1_20140   Packets: 60 Displayed: 60 Marked: 0 Dropped: 0   Profile: Default
```

Figure 18: ICMP packet with data generated by the ICMP protocol



```
53 8.030949   10.0.1.2   10.0.1.1   ICMP   514 Echo (ping) request  id=0x0859, se
54 8.031387   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP
55 8.031728   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP
56 8.031778   10.0.1.1   10.0.1.2   ICMP   514 Echo (ping) reply    id=0x0859, se
57 9.034251   10.0.1.2   10.0.1.1   IPv4   514 Fragmented IP protocol (proto=ICMP
58 9.034483   10.0.1.2   10.0.1.1   IPv4   514 Fragmented IP protocol (proto=ICMP
59 9.034628   10.0.1.2   10.0.1.1   ICMP   514 Echo (ping) request  id=0x0859, se
60 9.038332   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP
61 9.039060   10.0.1.1   10.0.1.2   IPv4   514 Fragmented IP protocol (proto=ICMP
62 9.039198   10.0.1.1   10.0.1.2   ICMP   514 Echo (ping) reply    id=0x0859, se

▶ Frame 52: 514 bytes on wire (4112 bits), 514 bytes captured (4112 bits)
▶ Ethernet II, Src: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6), Dst: CadmusCo_a7:b2:30 (08:00:27:a7:b2:30)
▶ Internet Protocol Version 4, Src: 10.0.1.2 (10.0.1.2), Dst: 10.0.1.1 (10.0.1.1)
▶ Data (480 bytes)

0000  08 00 27 a7 b2 30 08 00  27 ca 5e e6 08 00 45 00   ..'..0.. '.^...E.
0010  01 f4 fe 6b 20 3c 40 01  44 5f 0a 00 01 02 0a 00   ...k <@. D_......
0020  01 01 00 09 06 01 0c 04  01 0a 07 0e 03 0c 05 02   ........ ........
0030  09 0c 04 07 08 03 0d 09  08 0f 03 07 0e 02 0b 0b
0040  06 0b 04 0c 0c 01 01 0d  0b 08 0c 0e 04 01 00 0e   ........ ........
0050  0d 04 05 06 08 03 0f 00  02 02 07 00 05 02 0b 0b   ........ ........
0060  0d 0f 08 0a 00 09 07 0b  01 03 0a 06 05 0a 04 02   ........ ........
0070  0f 09 08 07 0c 07 07 0e  0a 0e 0e 0f 01 09 0a 0e   ........ ........
0080  09 02 08 09 0b 00 05 0d  03 0f 03 08 09 07 0b 08   ........ ........
0090  00 03 0f 0d 0b 06 0b 05  05 0a 04 06 03 0e 04 0c   ........ ........
00a0  01 0d 06 0c 0d 0b 09 00  0a 0c 09 03 03 04 0c 04   ........ ........
00b0  07 0b 01 02 02 0c 07 07  06 0b 0d 0a 0a 01 06 0b   ........ ........
00c0
```

Figure 19: ICMP packet with the random data we generated.

We are using IP Tables to capture and modify the packets, and this gives us some kind of synchronization problems that we couldn't solve: the 2 first packets we send are lost. In order to solve this problem, we prepared the script so it works as it follows: it starts by generating these 2 pings that are going to get lost, then it waits 20 seconds (so the user can start the wireshark capture, for example), and then it starts launching the pings.

This script solves the problem because the programs we are using for modifying the packets have an int-type variable called “trigger”, which is used for those 2 initial pings needed for synchronization: this variable works as a trigger that makes sure that no information is sent until the third ping is sent, moment in which the trigger value is reached. So, despite those 2 initial pings, no data (normal or hidden) is lost.

Of course, after changing all the data, our program recalculates both IP and ICMP checksum before sending the packet back into the network. These checksums are calculated by forming the ones’ complement of the ones’ complement sum of the header's 16-bit words, and this is directly what our program does after fixating the initial checksum values to 0. Check annex 1 if you want to look at the exact programming code implemented for it.

We must note that these operations have a cost: we have to introduce the data and recalculate the checksums, which takes some time, and so we have an increment in the total connection time if we compare it with the case in which we don’t change the packet’s data. There is no problem with it, since we are doing exactly the same operation in all of the fragmentation cases, so the overtime for introducing the randomly generated data will be affecting equally all of them.

5.2.2 F1 Method

For the second case of this scenario, we will have to implement F1 steganographic method. This method relies on the parity of the number of fragments that the packet was divided into.

For its implementation, we decided to send:

- 4 fragments if we want to send a secret binary 0.

- 3 fragments if we want to send a secret binary 1.

So, if we compare it with the no steganography case, we are sending an extra fragment (which is useful, since it carries normal information) every time we are sending a secret 0.

All of the sent fragments have the same size (500 bytes each, excluding Ethernet header), since we think is more realistic to find similar sized fragments when you study a real-life fragmentation scenario. Consequently, the full packets we are sending will be having different sizes, depending on if a secret 0 or 1 is being sent.

Of course, we won't be changing the packet size by ourselves each time we send a packet depending on the secret information we are sending: we will send equally sized packets, and our program is the one that will be changing their size.

If you check the script prepared for this and the other steganographic cases (included in the CD annex to this paper), you will see that it's exactly the same as in the previous case (trigger issue and all included). The only difference resides in the chosen ping size: 1912 bytes, so we obtain 4 equally sized fragments after the packet's fragmentation.

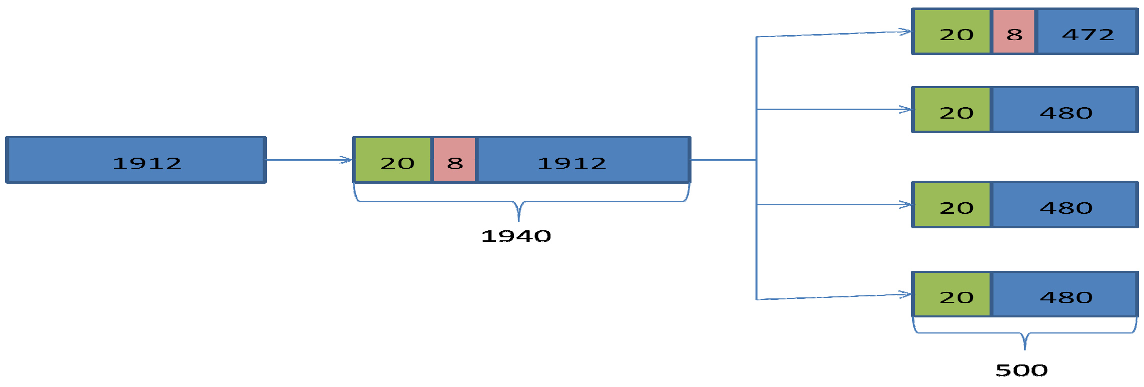


Figure 20: Fragmentation of a 1912 byte-length packet.

We are always generating and sending packets that carry 1912 bytes of data, independently of the secret information we want to send. We are sending 1912 bytes of dat instead of 1432 because using my program and IP Tables, we found it easier to make a smaller packet from a big one than the other way around, so a packet of 1912 bytes (which is divided into 4 equally sized fragments) is generated and then, depending on if we need the packet to be divided into 4 or 3 fragments, it's total length is altered or not, as we can see in the figure below:

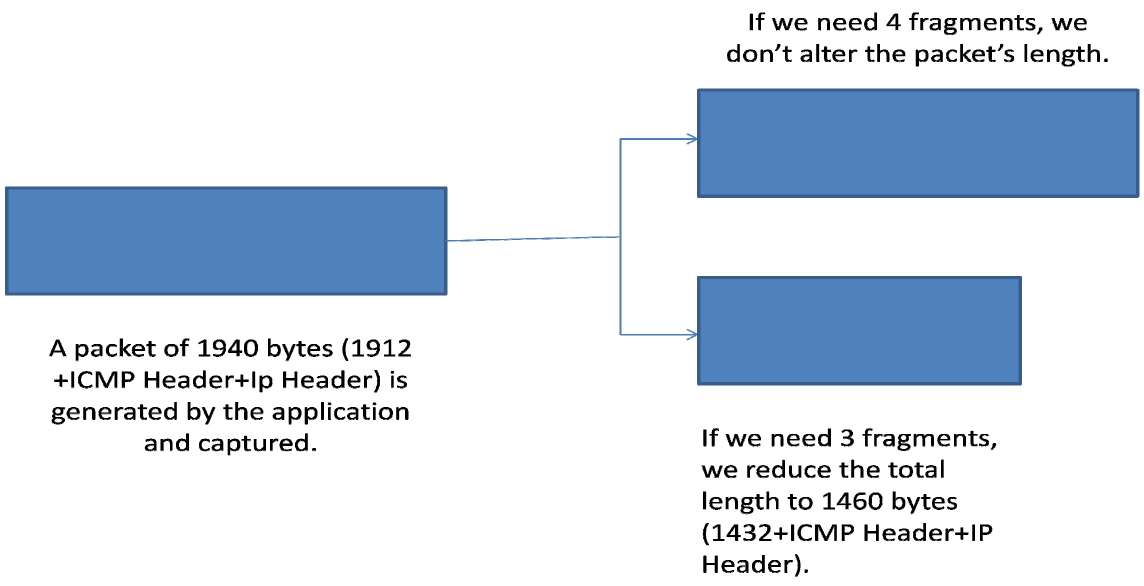

If we need 4 fragments, we don't alter the packet's length.
A packet of 1940 bytes (1912 +ICMP Header+Ip Header) is generated by the application and captured.
If we need 3 fragments, we reduce the total length to 1460 bytes (1432+ICMP Header+IP Header).

Figure 21: Resizing of the packets, F1 method.

Of course, when we do this a new issue appears: now, we are able to change the packet's total size, so we can have the number of fragments we desire, and so we can send the secret information without any problem. But we must also think about what happens with the not hidden information, the one carried by the IP packet.

Unfortunately, if we cut the packet, part of its information is lost, and we can't afford that. So we had to come up with another way to solve this problem. Since we do all this in the PC that is going to send both the normal file and the secret file, and so we have access to both files, what we decided to do is to reintroduce the normal, not hidden information again into the packet after it has already been prepared to be divided into 3 or 4 fragments.

We will try to illustrate it with an example: let's imagine we want to send secret data 00100 using F1 method. Let's suppose we are using a file that, when sent, need to be divided into 20 fragments. If we just send 1912 length packets with information and we cut the total length of the packet when we receive a 1 so it is divided into 3 fragments, we would be seeing that, while the secret information would be correctly received, we would be losing one of the fragments transporting not-secret information, as we illustrate in the next figure:

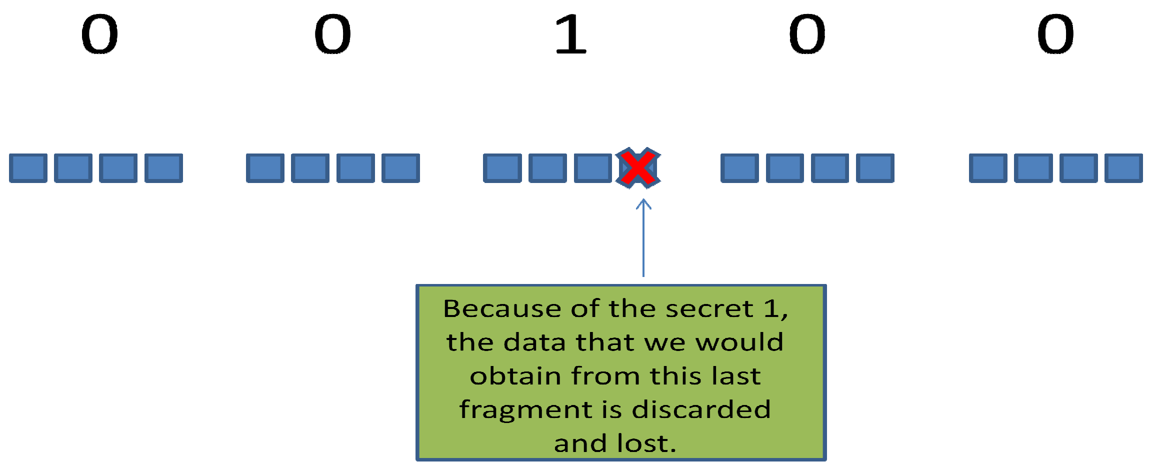
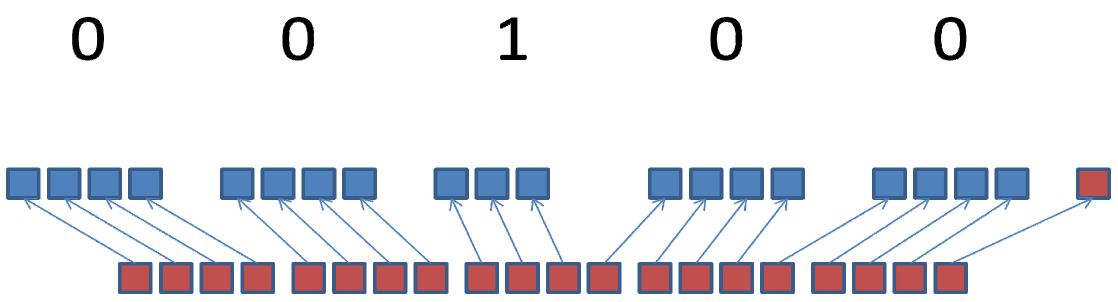

0
0
1
0
0
Because of the secret 1, the data that we would obtain from this last fragment is discarded and lost.

Figure 22: Fragment lost because of F1 method proposed implementation.

This problem is avoided when the program has direct access to the file we are sending: let's think of the same example as above: sending the secret sequence 00100 using F1 method. But now, let's suppose that our program has access not only to the secret data, but also to the file of data we want to send. After obtaining a packet and altering its length, what we can do is re-write the original data we want to send into the newly formed packet using the space we have: if we have 1912 bytes available to fill, 1912 bytes will be written; if we just have 1432, only 1432 will be written.

After the end of the secret transmission, since we know we are using a steganographic method, we can make our application produce some extra packets that will not be used to carry secret information, but to send the normal information that has yet to be sent.

If for example, we know that we have sent a total of 400 secret 1's, we will know that after the secret transmission is finished we have yet to send 400 more fragments of real information: 100 packets. Since we will be aware of this fact, we can adapt our file or our application to it. For example, we can modify the file we are going to send, filling the end of the file with useless data that will be replaced by the original one after the steganographic method is applied on it. We illustrate this in the next figure:



After preparing the initial packet so it is divided into three or four fragments, we re-write the original data into the obtained packet. At the end, we have to send the unsent original data, even if we have to send just one fragment.

Figure 23: F1 method correct implementation.

So, to sum it up, what our program does is to capture the packet; then depending on if a secret 1 or 0 is being sent it changes the total packet length (by changing the value of variable ret, which contains the captured packet's length); and finally we introduce the random data we generated into the newly obtained packet, the same way we did it in the no steganography case. On the receiver size, since we obtain full packets of different lengths, we determine if a 0 or a 1 was sent by looking at the packet's length (again, stored in variable ret after capturing the packet).

One thing that we must notice is that we are working by altering and observing full packets, instead of fragments: we are doing so because we couldn't manage to capture single fragments with IP-tables and Net Filter: even when using the tools they offer for it, we only managed to capture full packets. Luckily, it's possible to implement all the steganographic methods we want to implement by working with the full packets instead of with the fragments.

5.2.3 F3 Method

Let's go with F3 method now. This method, as we already described in previous sections, utilizes legitimate fragments with steganogram inserted into payload for higher steganographic bandwidth and harder detection. SS is the source of the fragmentation and control the process. During the fragmentation, SS inserts secret data instead of inserting user data into the payload of selected fragments.

To make the steganographic fragments distinguishable from others yet hard to detect the following procedure was introduced: if SS and SR share a secret Steg-Key (SK) then for each fragment chosen for steganographic communication the following hash function (H) is used to calculate what we call the Identifying Sequence (IS):

$$IS = H(SK||Fragment\ offset||Identification)$$

Where *Fragment offset* and *Identification* denote values from these IP fragment header fields, and || bits concatenation function. For every fragment used for hidden communication, the resulting IS will have different value due to the values change in the fragment offset and identification fields. All IS bits or only selected ones are distributed across payload field in predefined manner. Thus, for each fragment SR can calculate appropriate IS and verifies if it contains secret or user data. If the verification is successful then the rest of the payload is considered as hidden data and extracted. Then, SR does not utilize this fragment in reassembly process of original IP packet.

So, when using this method, some extra fragments are generated: if we compare it with the no steganography case, we are sending 4 fragments instead of 3: the same 3 fragments that carry the information and one extra fragment per packet that carries the hidden information.

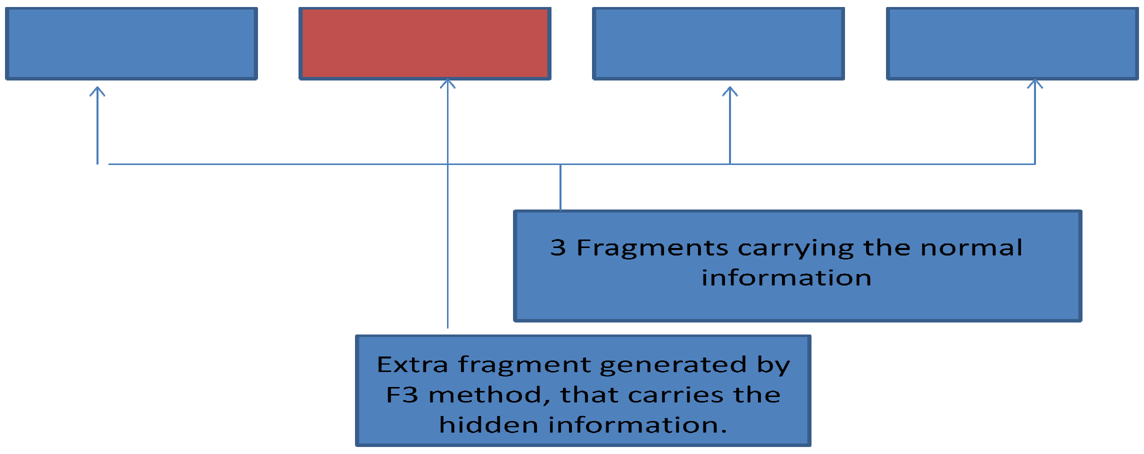

3 Fragments carrying the normal information
Extra fragment generated by F3 method, that carries the hidden information.

Figure 24: F3 method description.

For this method implementation, we are using 1912-length packets again. This time we won't be modifying their length, so we will always be obtaining 4 fragments per packet.

We decided to use the second fragment for sending the secret information. There is no special reason for this: we just didn't want to make the programming code more complicated. The normal data is introduced into the first, third and fourth fragments, so even if we have 4 fragments, we are sending the same quantity of normal information per packet that we were sending on the No Steganography case. Of course, we do all this working with the full packet: since we know exactly how the packet is going to be fragmented and which bytes are going to be in every packet, there is no problem with doing it.

As for the second fragment, we fully use it to send 476 hidden bits (the total fragment's size minus the number of bytes occupied by the IS), so in this case we are sending 476 bits per packet. We implement the method exactly as we described before: first, we introduce into the first bytes of an array the Steg-Key (which is known for both SS and SR), followed by the fragment offset (Known because we know it's the second fragment and the number of bytes carried by each fragment) and the packet ID (extracted from the packet's IP Header).

Of course, all of this is affected by a hash function known by both sides of communication. We decided to use the function hash(), already included in the standard C++ libraries. With it, we obtain a 4 character string, which we put at the beginning of the fragment. After this 4 character string, we introduce the 476 secret bits we are sending. Finally, we recalculate both ICMP and IP header checksums, as in the previous case.

In the receiving side, we just extract the correct fragments in order into a buffer (which will contain the normal data) that can later give it to the application. After checking if the second fragment has a correct identifying sequence, the 8 bits secretly sent are extracted. The rest of the second fragment is discarded, since it's useless.

5.2.4 F1+F3 Methods

In first scenario's final case, we will be using F1 and F3 methods combined. Because F1 method modulates the number of fragments that the packet is divided into and F3 inserts fake fragments, we can use them combined in a way that will make the total steganographic cost decrease and undetectability increase: we use F1 Method, with which we are obtaining 4 or 3 fragments per packet, and then we also use F3 method every time we have obtained 3 fragments using method F1, so we obtain an extra fragment that makes us always have 4 fragments per packet. The steganographic cost decrease we obtain when combining the two methods is what we call *super-position steganography.* You can see how the method works in the figure below.

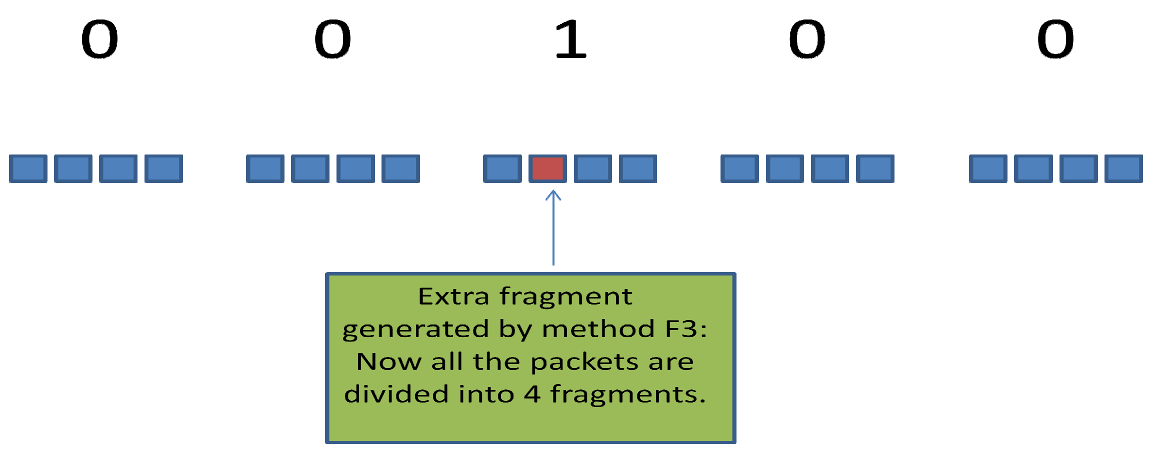

0
0
1
0
0
Extra fragment generated by method F3: Now all the packets are divided into 4 fragments.

Figure 25: F1+F3 methods description.

For the implementation of this method, we just combined in a proper way the programs we had already made for the F1 and F3 cases separately: we generate 1912 byte-length pings, which will be divided into 4 fragments each. We start by using the F1 method: if a 0 was sent, then we just fill the packet with the information we generated and send it back to the network. If a 1 was sent, we apply the F3 method to this packet: we introduce the normal information into the first, third and fourth fragment and use the second one for introducing more hidden data. Of course, after all these alterations, we recalculate both ICMP and IP Header-Checksums.

In the receiving side, it's easy to extract the secret: after receiving the packet, we search for the Identifying sequence (IS) we use with method F3: if we don't find it, we know that a 0 was sent using method F1 and we save all the information contained by the packet. If we find the IS, we know that a 1 was sent using method F1 and that method F3 was also used, so we write the secret one sent using F1 method into the secret information, then we extract the hidden information sent using F3 method in the second fragment, and finally we save the information carried by the first, third and fourth fragment.

5.3 HTTP RELATED CASES

We will now proceed to describe the HTTP scenario's cases. We will start by explaining what are we sending in the no steganography case, and after that we will explain each of the used steganographic methods and the way we implemented them.

5.3.1 No Steganography

In this scenario, we are going to work with HTTP protocol: PCA will be requesting PCB (which works as a web server) for webpage index.html. We will be repeating this HTTP Request a lot of times, and when using the steganographic methods, we will be altering the HTTP headers in order to send the secret information.

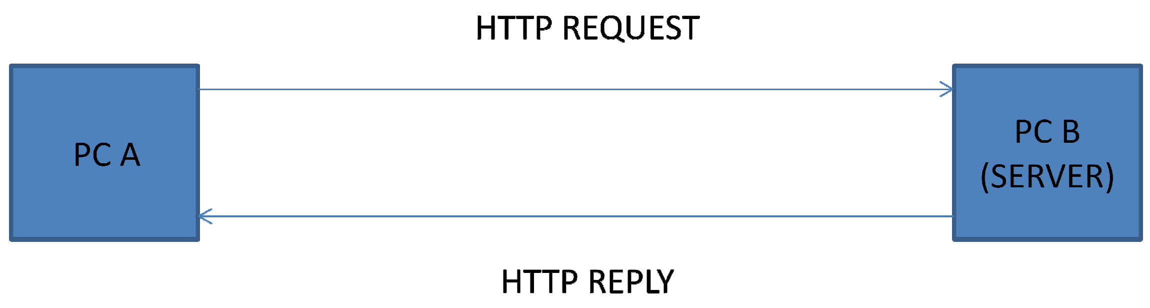


Figure 26: HTTP scenario.

Unlike the first scenario's No steganography case, in which we had to alter the packet in order to introduce the random data generated by ourselves, in this case we don't have to modify the HTTP request, so we just have to run the script we prepared for the second scenario.

This script basically asks a lot of times for the index.html page, using the following order:

Wget 10.0.1.2/index.html --no -cache

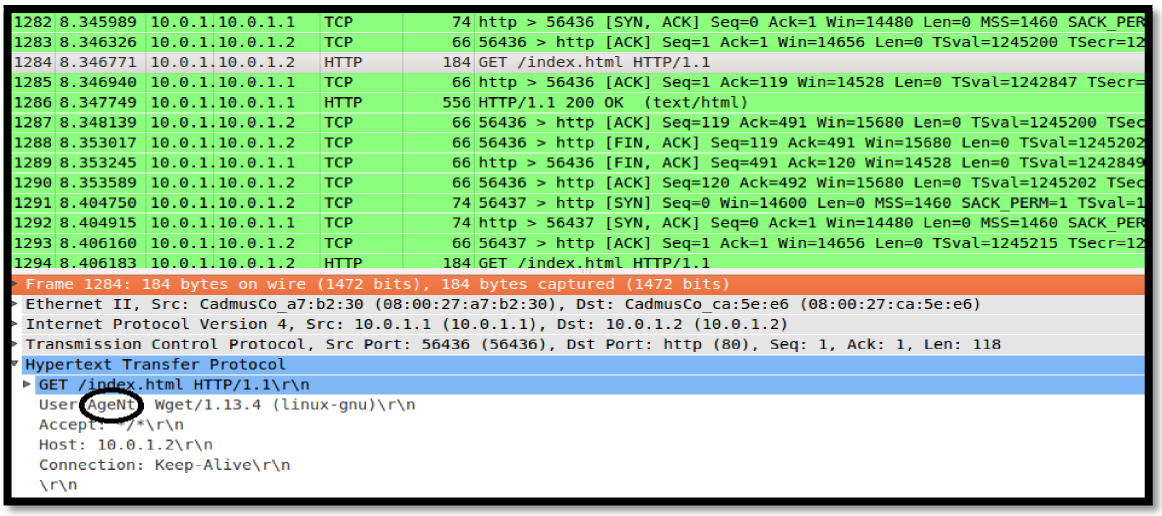
Since they end up being stored on the PC, we delete the obtained index.html page after each request.

It's important to note that, in the script, we send a normal request and then we wait for 20 seconds before sending the rest of the request. This first request is sent because, just like on the previous fragmentation cases, when capturing the packets using IP Tables for their modification in the steganographic cases, there are some synchronization issues that are again solved using a trigger in our programs, so no information is sent on this first HTTP request. The 20 second wait time is so we can start the wireshark capture.

5.3.2 F6 Method

The first steganographic method we will describe is the F6 method. This method basically changes the case of one or more of the header fields, using capital or small letters depending on the information we want to send. We can use for example small letters for representing secret 0's and capital letters for representing secret 1's. The total header length is not altered by the usage of this method.

For its implementation, we decided to only use the User-Agent header, modifying only the case of the word Agent. Since it has 5 letters, we are able to send 5 secret bits per packet. We can see an example on the next image:



```
1282 8.345989 10.0.1.10.0.1.1   TCP      74 http > 56436 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PER
1283 8.346326 10.0.1.10.0.1.2   TCP      66 56436 > http [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1245200 TSecr=12
1284 8.346771 10.0.1.10.0.1.2   HTTP    184 GET /index.html HTTP/1.1
1285 8.346940 10.0.1.10.0.1.1   TCP      66 http > 56436 [ACK] Seq=1 Ack=119 Win=14528 Len=0 TSval=1242847 TSecr=
1286 8.347749 10.0.1.10.0.1.1   HTTP    556 HTTP/1.1 200 OK  (text/html)
1287 8.348139 10.0.1.10.0.1.2   TCP      66 56436 > http [ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=1245200 TSec
1288 8.353017 10.0.1.10.0.1.2   TCP      66 56436 > http [FIN, ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=1245202
1289 8.353245 10.0.1.10.0.1.1   TCP      66 http > 56436 [FIN, ACK] Seq=491 Ack=120 Win=14528 Len=0 TSval=1242849
1290 8.353589 10.0.1.10.0.1.2   TCP      66 56436 > http [ACK] Seq=120 Ack=492 Win=15680 Len=0 TSval=1245202 TSec
1291 8.404750 10.0.1.10.0.1.2   TCP      74 56437 > http [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=1
1292 8.404915 10.0.1.10.0.1.1   TCP      74 http > 56437 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PER
1293 8.406160 10.0.1.10.0.1.2   TCP      66 56437 > http [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1245215 TSecr=12
1294 8.406183 10.0.1.10.0.1.2   HTTP    184 GET /index.html HTTP/1.1
▸ Frame 1284: 184 bytes on wire (1472 bits), 184 bytes captured (1472 bits)
▸ Ethernet II, Src: CadmusCo_a7:b2:30 (08:00:27:a7:b2:30), Dst: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6)
▸ Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.1.2 (10.0.1.2)
▸ Transmission Control Protocol, Src Port: 56436 (56436), Dst Port: http (80), Seq: 1, Ack: 1, Len: 118
▾ Hypertext Transfer Protocol
  ▸ GET /index.html HTTP/1.1\r\n
    User-AgeNt: Wget/1.13.4 (linux-gnu)\r\n
    Accept: */*\r\n
    Host: 10.0.1.2\r\n
    Connection: Keep-Alive\r\n
    \r\n
```

Figure 27: Wireshark capture, F6 method.

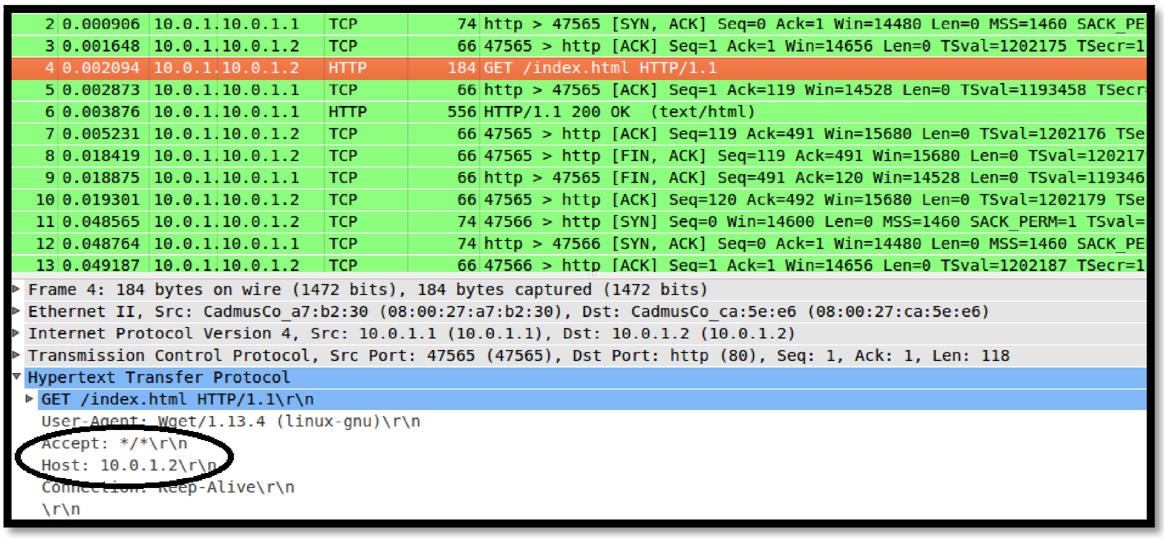
The case modification is pretty easy to do: after capturing the HTTP Request, our program has a pointer to the Agent word in the User-Agent header. Then, depending on the secret information we want to send, it changes the case of the letters by adding or subtracting the fixed quantity that separates small and capital letters in ASCII code: 32. After doing this, we recalculate both IP and TCP header checksums, so they fit, and then we send the packet back to the network.

On the receiving side, we just check the ASCII code for the Agent word in the User-Agent header in order to know if a secret 1 or 0 was sent.

5.3.3 F7 Method

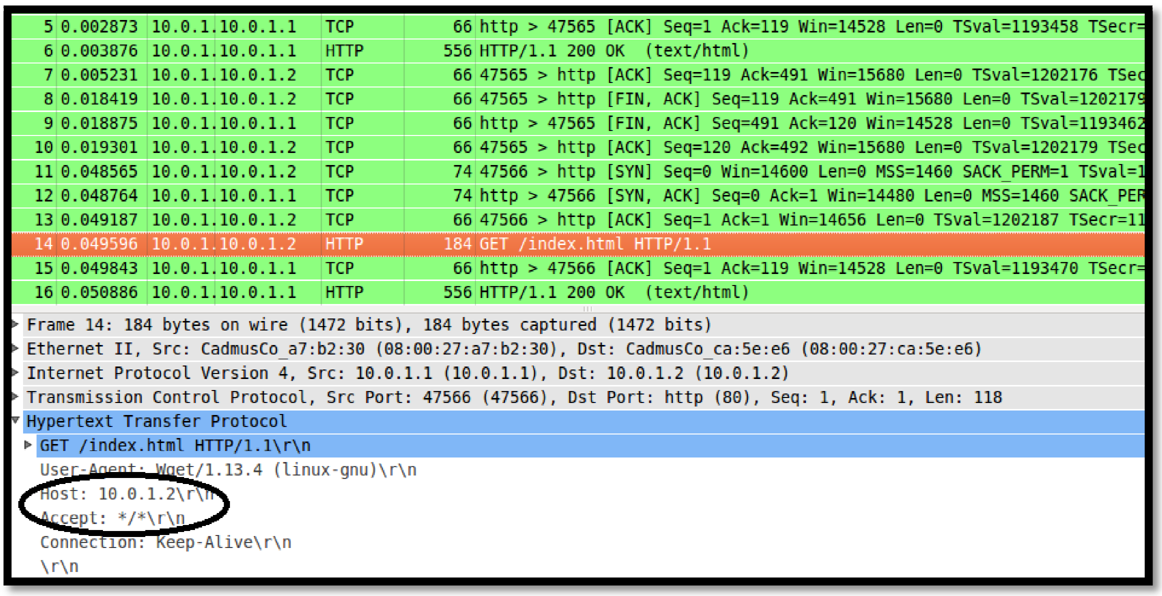
We will now explain F7 method. This method works by changing the order of the header fields, so depending on the secret information we are sending the headers will be following one order or the other. At least 2 headers are required for the implementation of this method. The total header length is not altered by the usage of this method.

For its implementation, we decided to use the Host header and the Accept header. When we are sending a secret 0, they are in their predetermined order (first the Accept header, then the Host one), and when we are sending a secret 1, their order is changed (first the host header, then the Accept one).



```
 2 0.000906 10.0.1.10.0.1.1  TCP       74 http > 47565 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PE
 3 0.001648 10.0.1.10.0.1.2  TCP       66 47565 > http [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1202175 TSecr=1
 4 0.002094 10.0.1.10.0.1.2  HTTP     184 GET /index.html HTTP/1.1
 5 0.002873 10.0.1.10.0.1.1  TCP       66 http > 47565 [ACK] Seq=1 Ack=119 Win=14528 Len=0 TSval=1193458 TSecr
 6 0.003876 10.0.1.10.0.1.1  HTTP     556 HTTP/1.1 200 OK  (text/html)
 7 0.005231 10.0.1.10.0.1.2  TCP       66 47565 > http [ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=1202176 TSe
 8 0.018419 10.0.1.10.0.1.2  TCP       66 47565 > http [FIN, ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=120217
 9 0.018875 10.0.1.10.0.1.1  TCP       66 http > 47565 [FIN, ACK] Seq=491 Ack=120 Win=14528 Len=0 TSval=119346
10 0.019301 10.0.1.10.0.1.2  TCP       66 47565 > http [ACK] Seq=120 Ack=492 Win=15680 Len=0 TSval=1202179 TSe
11 0.048565 10.0.1.10.0.1.2  TCP       74 47566 > http [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=
12 0.048764 10.0.1.10.0.1.1  TCP       74 http > 47566 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PE
13 0.049187 10.0.1.10.0.1.2  TCP       66 47566 > http [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1202187 TSecr=1
▸ Frame 4: 184 bytes on wire (1472 bits), 184 bytes captured (1472 bits)
▸ Ethernet II, Src: CadmusCo_a7:b2:30 (08:00:27:a7:b2:30), Dst: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6)
▸ Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.1.2 (10.0.1.2)
▸ Transmission Control Protocol, Src Port: 47565 (47565), Dst Port: http (80), Seq: 1, Ack: 1, Len: 118
▾ Hypertext Transfer Protocol
  ▸ GET /index.html HTTP/1.1\r\n
    User-Agent: Wget/1.13.4 (linux-gnu)\r\n
    Accept: */*\r\n
    Host: 10.0.1.2\r\n
    Connection: Keep-Alive\r\n
    \r\n
```

Figure 28: Wireshark capture, F7 method (I).



```
 5 0.002873 10.0.1.10.0.1.1  TCP       66 http > 47565 [ACK] Seq=1 Ack=119 Win=14528 Len=0 TSval=1193458 TSecr=
 6 0.003876 10.0.1.10.0.1.1  HTTP     556 HTTP/1.1 200 OK  (text/html)
 7 0.005231 10.0.1.10.0.1.2  TCP       66 47565 > http [ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=1202176 TSec
 8 0.018419 10.0.1.10.0.1.2  TCP       66 47565 > http [FIN, ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=1202179
 9 0.018875 10.0.1.10.0.1.1  TCP       66 http > 47565 [FIN, ACK] Seq=491 Ack=120 Win=14528 Len=0 TSval=1193462
10 0.019301 10.0.1.10.0.1.2  TCP       66 47565 > http [ACK] Seq=120 Ack=492 Win=15680 Len=0 TSval=1202179 TSec
11 0.048565 10.0.1.10.0.1.2  TCP       74 47566 > http [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=1
12 0.048764 10.0.1.10.0.1.1  TCP       74 http > 47566 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PER
13 0.049187 10.0.1.10.0.1.2  TCP       66 47566 > http [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1202187 TSecr=11
14 0.049596 10.0.1.10.0.1.2  HTTP     184 GET /index.html HTTP/1.1
15 0.049843 10.0.1.10.0.1.1  TCP       66 http > 47566 [ACK] Seq=1 Ack=119 Win=14528 Len=0 TSval=1193470 TSecr=
16 0.050886 10.0.1.10.0.1.1  HTTP     556 HTTP/1.1 200 OK  (text/html)
▸ Frame 14: 184 bytes on wire (1472 bits), 184 bytes captured (1472 bits)
▸ Ethernet II, Src: CadmusCo_a7:b2:30 (08:00:27:a7:b2:30), Dst: CadmusCo_ca:5e:e6 (08:00:27:ca:5e:e6)
▸ Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.1.2 (10.0.1.2)
▸ Transmission Control Protocol, Src Port: 47566 (47566), Dst Port: http (80), Seq: 1, Ack: 1, Len: 118
▾ Hypertext Transfer Protocol
  ▸ GET /index.html HTTP/1.1\r\n
    User-Agent: Wget/1.13.4 (linux-gnu)\r\n
    Host: 10.0.1.2\r\n
    Accept: */*\r\n
    Connection: Keep-Alive\r\n
    \r\n
```

Figure 29: Wireshark capture, F7 method (II).

This method is pretty easy to implement too: first we capture the HTTP request packet, and then only when we have to send a secret 1 we just change the header's order using an auxiliary buffer. After this, we recalculate IP and TCP header-checksums and we send the packets back to the network.

One the receiving side, we just look which header is first by looking at the ASCII code of the first letter in the first header: just with that, we are able to determine if a secret 0 or 1 was sent.

5.3.4 F8 Method

The next method we will be explaining is the F8 method. This method changed the total number of header fields in order to send the secret information: depending on the total number of header fields sent per packet, we will know if a secret 0 or 1 was sent. The total header length is altered when using this method.

In our case, what we decided to do is to send a total number of 4 header fields when sending a secret 0 (the ones normally generated when making our HTTP Request: User-Agent, Accept, Host and Connection) and a number of just 3 header fields when sending a secret 1 (eliminating in this case the Connection Header field). So, we either send the packet without altering it or we make it shorter eliminating one header field. Again, we decided to make the packet shorter instead of longer because we found it easier to do with netfilter and IP Tables than making it longer.

In the receiving side, initially we just checked the total packet length in order to see if a secret 0 or 1 had been sent.

Unfortunately, when doing this, something really bad was happening: the HTTP connection was constantly resetting. After some research, we discovered that the problem was in the TCP connection and the ACK method that it uses.

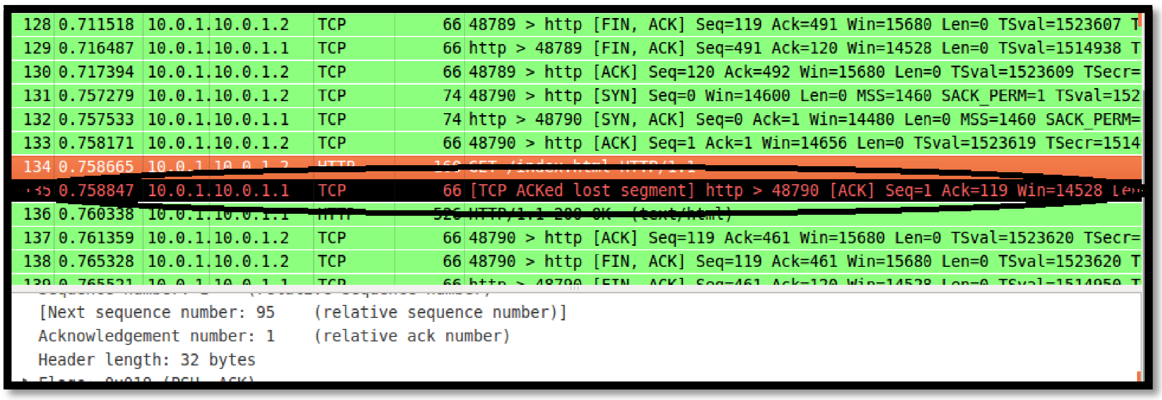
We will try to illustrate the problem: let's imagine our application has created a 200 byte-length packet and it sends it into the network. This packet is captured by our program and modified, so it ends up having only a total amount of 180 bytes for example. It reaches the receiving side of the connection and, since it's a correct packet, the receiving side just extracts it and sends back an ACK packet to the sending side. And here is where the problem resides: the sending side receives an ACK confirmation for 180 bytes, and since it sent a 200 byte-length packet, it assumes that the last bytes were lost and so it retransmits this last 20 bytes.

This retransmission reaches the receiving side again, but since they only carry the last part of an HTTP request without header and anything the receiving side doesn't know what to do with this data, and so the connection crashes.

We tried to solve this problem modifying the ACK numbers, but it didn't solve anything. The main problem is that, no matter how much we modify the packet, the application that originally generated it knows how it was, and so it notices if something is going wrong with the connection and so it ends up resetting it.

After much thinking, we came out with a solution, the one we finally implemented: on the sending side, the program used is the same (we eliminate the connection header field when sending a 1), but on the receiving side, after looking at the packet's length and determining if a secret 0 or 1 had been sent, we reconstruct the eliminated header field in case a secret 1 was sent before sending the packet to the application. By doing this, the receiving side sends back to the sending side an ACK number that matches its expectations, and so the connection doesn't reset and ends up in a proper way.

Of course, this solution is not 100% perfect: even though the connection doesn't end up resetting, we can see some strange packets in wireshark every time we send a 1. These aren't incorrect packets, but packets that wireshark marks as strange, since even though everything works fine there is something that doesn't fit with the ACK numbers. Wireshark makes the interpretation that the connection went fine, and that there was a packet which it didn't capture but reached the destiny correctly following another possible network route or path. You can see these strange packets in the figure below, obtained every time a 1 is being sent:



```
128 0.711518 10.0.1.10.0.1.2  TCP   66 48789 > http [FIN, ACK] Seq=119 Ack=491 Win=15680 Len=0 TSval=1523607 T
129 0.716487 10.0.1.10.0.1.1  TCP   66 http > 48789 [FIN, ACK] Seq=491 Ack=120 Win=14528 Len=0 TSval=1514938 T
130 0.717394 10.0.1.10.0.1.2  TCP   66 48789 > http [ACK] Seq=120 Ack=492 Win=15680 Len=0 TSval=1523609 TSecr=
131 0.757279 10.0.1.10.0.1.2  TCP   74 48790 > http [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=152
132 0.757533 10.0.1.10.0.1.1  TCP   74 http > 48790 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM=
133 0.758171 10.0.1.10.0.1.2  TCP   66 48790 > http [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=1523619 TSecr=1514
134 0.758665 10.0.1.10.0.1.2  HTTP 160 GET /index.html HTTP/1.1
135 0.758847 10.0.1.10.0.1.1  TCP   66 [TCP ACKed lost segment] http > 48790 [ACK] Seq=1 Ack=119 Win=14528 Len
136 0.760338 10.0.1.10.0.1.1  HTTP 526 HTTP/1.1 200 OK  (text/html)
137 0.761359 10.0.1.10.0.1.2  TCP   66 48790 > http [ACK] Seq=119 Ack=461 Win=15680 Len=0 TSval=1523620 TSecr=
138 0.765328 10.0.1.10.0.1.2  TCP   66 48790 > http [FIN, ACK] Seq=119 Ack=461 Win=15680 Len=0 TSval=1523620 T
139 0.765521 10.0.1.10.0.1.1  TCP   66 http > 48790 [FIN, ACK] Seq=461 Ack=120 Win=14528 Len=0 TSval=1514950 T

   Sequence number: 1    (relative sequence number)
   [Next sequence number: 95    (relative sequence number)]
   Acknowledgement number: 1    (relative ack number)
   Header length: 32 bytes
 ▸ Flags: 0x018 (PSH, ACK)
```

Figure 30: Wireshark capture, strange packets obtained when trying F8 method.

After many trials we weren't able to eliminate the appearance of at least some of these strange packets, but at least we managed for the F8 method to work as it should without the connection resetting. And, since the appearance of this kind of packets would be not that strange in bigger networks with multiple paths, we think that it's not that big of a problem.

5.3.5 F6+F7+F8 methods

For the final case, we just used F6, F7 and F8 methods all together, working as we described earlier. For its implementation, we just combined the already made programs for the individual cases.

So, in the sending side, we first apply F8 method, then F6 method and then F7 method to the captured packet. After receiving it, we extract the secret information the exact same way.

6. SIMULATION AND ANALYSIS OF RESULTS

6.1 Simulation

With all the programs done, the next step is to make all the simulations, in order to get the results so we can analyze them and see if we were right with the superposition steganography's and the zero cost steganography's hypothesis.

6.1.1 Scenario 1

For the first scenario (IP-fragmentation), as we already know, we have two virtual machines, PC-A and PC-B. They are connected to the internet and to a common intern network (network 10.0.1.0/24). PC-B is the steganogram sender (SS), and PC-A acts as the steganogram receiver (SR). We prepared a pair of scripts that make PC-B send a total amount of 2400 ICMP echo request packets (altered by PC-B itself so they carry random data as we mentioned) directed to PC-A, via the intern network that connects them. The random data carried is always generated the same way and so it's always the same stream of random information, no matter the steganographic method applied, so we are sending the same normal, not hidden information in every transmission.

PC-B further alters the packets that are going to be sent depending on the steganographic method we are implementing. Then, PC-A (the SR) will be receiving them and, before sending back the echo reply, separating the normal and the secret information. Only the interfaces connected to the intern network will be needed: the internet interfaces aren't used on this scenario.

We will be capturing all the packets on the SR side (PC-A) with Wireshark. The time of the capture will always starts once the first fragment of the first data packet is captured, and we will be sending the same number of packets on each simulation. The network MTU is equal to 500 bytes, and that's the size that all the fragments will be having on this scenario (the total length of the packets and, with it, the total number of fragments they are fragmented into will vary depending on the steganographic method utilized). We will be making 10 simulations per case, so a total of 40 simulations for the first scenario.

The secret information that we are sending is t the first chapter of book "El ingenioso Hidalgo Don Quixote de la Mancha" (we translated each character into their respective ASCII code, and that's what we send).

We illustrate the scenario on the next figure.

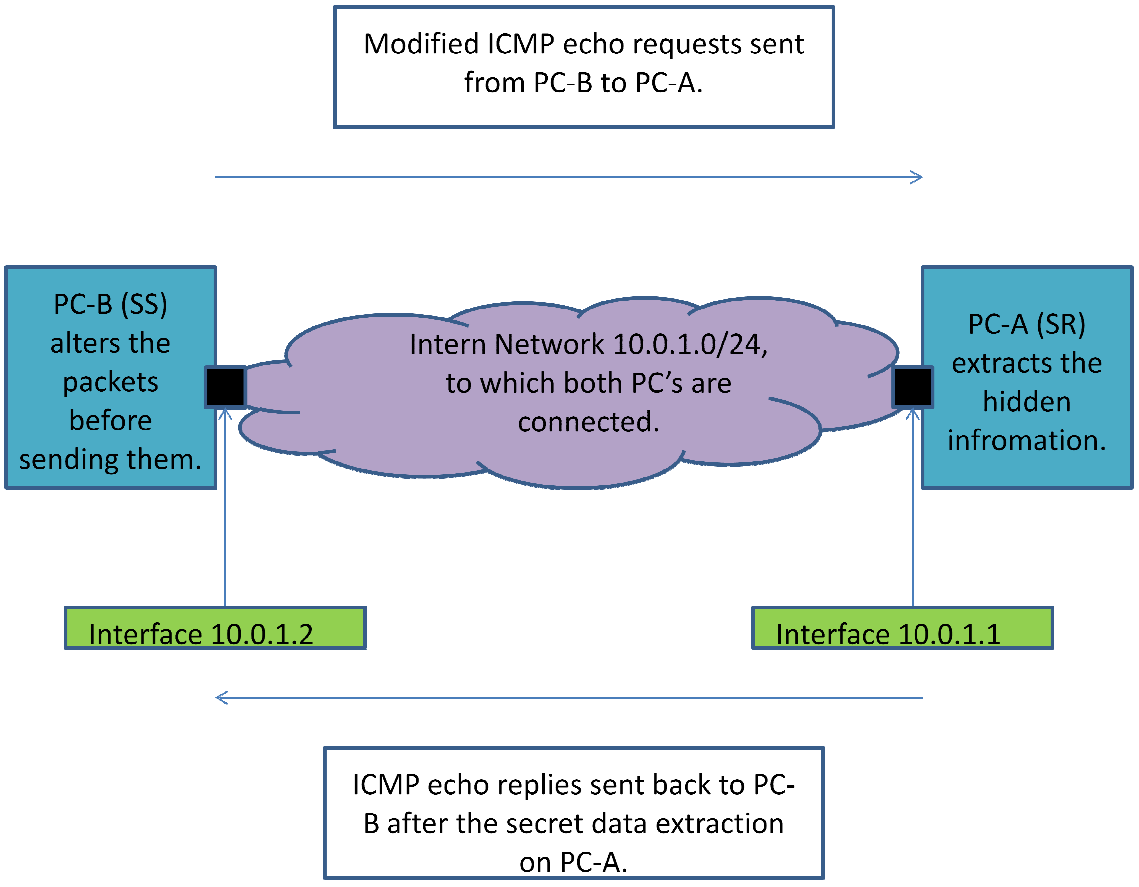

Modified ICMP echo requests sent from PC-B to PC-A.
PC-B (SS) alters the packets before sending them.
Intern Network 10.0.1.0/24, to which both PC's are connected.
PC-A (SR) extracts the hidden infromation.
Interface 10.0.1.2
Interface 10.0.1.1
ICMP echo replies sent back to PC-B after the secret data extraction on PC-A.

Figure 31: ICMP Scenario depiction.

6.1.2 Scenario 2

The second scenario is pretty similar to the first one: again, we have the same 2 virtual machines (PC-A and PC-B), connected directly to the internet and to an intern network. This time, PC-B will be acting as a WEB Server, and PC-A will be constantly making the same HTTP Request for the webpage "index.html", a total number of 900 times, using the command:

Wget 10.0.1.2/index.html -- no - cache

This command allows us to request for a webpage without using Firefox or any other web browser, and it makes sure that we don't have problems with the browser's cache (since we are going to ask for the webpage multiple times, and we don't want for the cache to affect our results).

PC-A acts as the steganogram sender (SS) this time: it changes the HTTP Headers of the HTTP request packets (depending on the applied method) before sending them into the network. Once they arrive to the server on PC-B (the steganogram receiver, SR), the secret information is extracted. Then, PC-B sends the asked webpage "index.html" back to PC-A. Once again, the internet interfaces aren't used on the scenario.

The capture will be realized on PC-B this time (SR side), using Wireshark. The time of the capture will always be starting once the first HTTP connection packet for the first request is captured. We will be making the same number of requests each time. As with the previous scenario, we will be making 10 simulations per case (so a total amount of 50 simulations for this scenario).

Again, the secret information that we are sending is the first chapter of book "El ingenioso Hidalgo Don Quixote de la Mancha".

We illustrate the second scenario in the figure below.

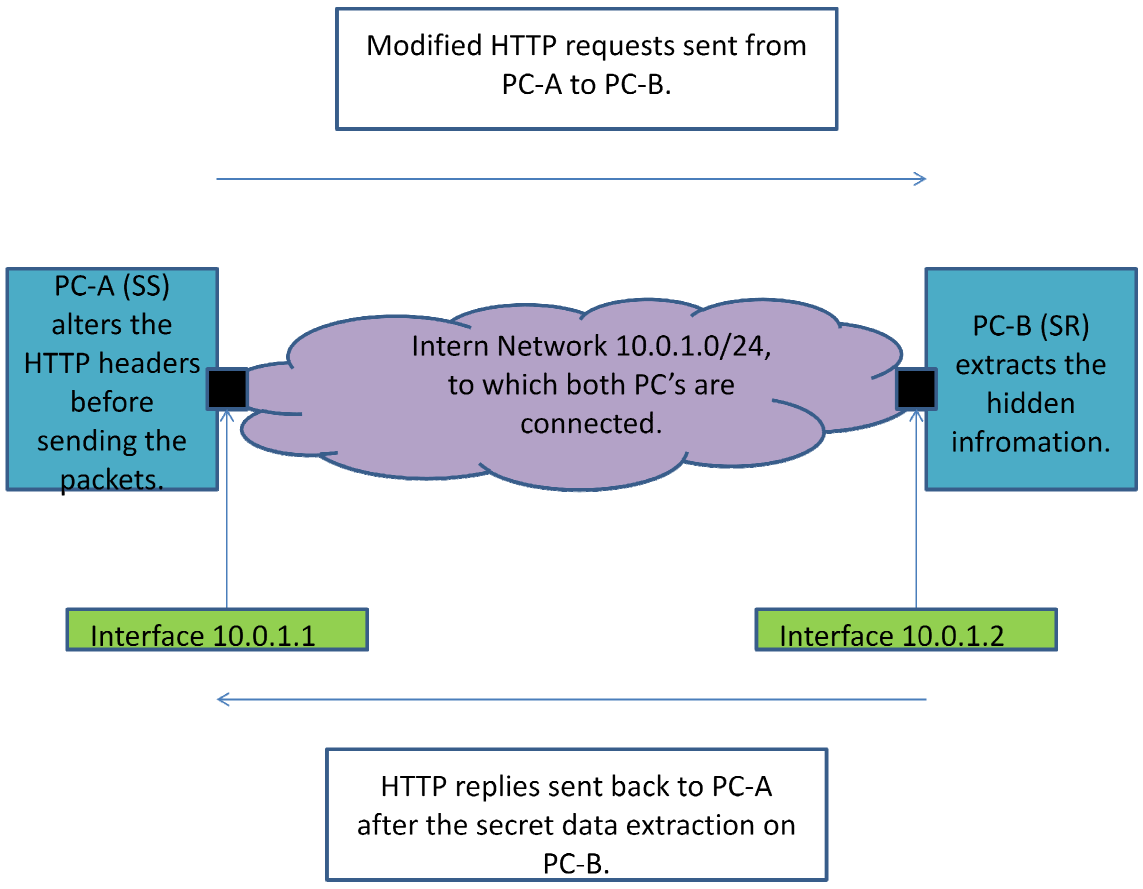

Modified HTTP requests sent from PC-A to PC-B.
PC-A (SS) alters the HTTP headers before sending the packets.
Intern Network 10.0.1.0/24, to which both PC's are connected.
PC-B (SR) extracts the hidden infromation.
Interface 10.0.1.1
Interface 10.0.1.2
HTTP replies sent back to PC-A after the secret data extraction on PC-B.

Figure 32: HTTP Scenario depiction.

6.2 Analysis of the results

We will now present a summary of the obtained results, including some charts and graphics. We will also discuss them in order to see if we are right or not with our hypothesis. All the 90 obtained captures are included in the CD annex to this paper.

6.2.1 Scenario 1: IP Fragmentation

For this scenario, we decided to use the total connection time and the distribution of the number of fragments obtained per packet in order to measure the steganographic cost and its variations.

These are the time results obtained for the first scenario:

	No Steganography ICMP Total Time (s)	F1 Method Total Time(s)	F3 Method Total Time (s)	F1+F3 Methods Total time (s)
	76,273770	73,102758	79,503706	77,850667
	73,996007	73,203190	81,625881	77,906488
	73,560484	74,343157	81,470780	77,858460
	74,496355	73,456459	81,180423	78,403213
	74,557246	72,631783	80,791409	77,948830
	74,393596	73,930039	80,160926	79,086417
	75,185952	74,227908	80,502305	77,786762
	75,948565	73,579992	81,038079	78,239853
	74,885760	74,717068	80,750701	78,138004
	74,471930	74,294615	80,699552	78,322218
MEAN	74,7769665	73,7486969	80,7723762	78,1540912
STD. DEVIATION	0,790502148	0,6267261	0,592942475	0,372156939

Figure 33: IP Fragmentation, table of times.

Each obtained time makes reference to the total connection time, from the moment when the first ICMP echo request packet is sent, until the moment when the last ICMP echo reply is fully received. As we said, we made 10 different captures per method, and so we used them to obtain a mean time and a standard deviation per case. Assuming these times follow a normal distribution, we drew the resulting distributions for each of the methods, using for it the calculated means and standard deviations:

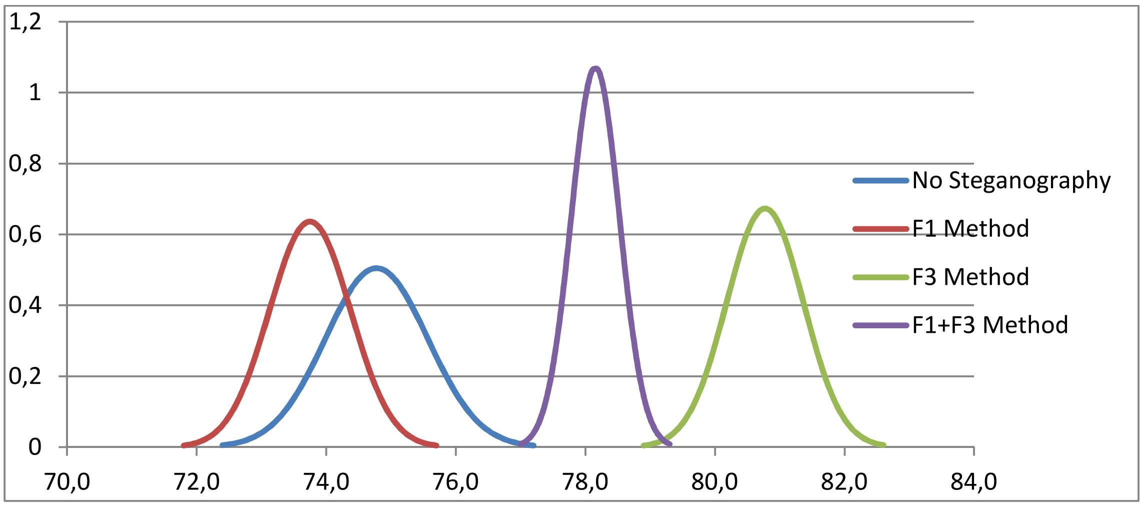


Figure 34: Time Distribution for each of the Methods, Scenario 1

We also took measures about other aspects related with the fragments and packets distributions for each case, as you can see in the next charts:

FRAGMENTS SIZE
Including Ethernet header, 514 bytes every fragment for all the conexions.

Figure 35: Fragment size for each case.

TOTAL NUMBER OF PACKETS.
2400 in every case

Figure 36: Total number of packets for each case.

TOTAL NUMBER OF FRAGMENTS SENT.			
No Steganography	F1 Method	F3 Method	F1+F3 Methods
7200 fragments.	8498 fragments.	9600 fragments.	9600 fragments.

Figure 37: Total number of fragments sent for each case.

NUMBER OF FRAGMENTS EACH PACKET IS FRAGMENTED INTO			
No Steganography	F1 Method	F3 method	F1+F3 Methods
3 fragments per IP packet.	1498 packets into 4 fragments	4 fragments per IP packet.	4 fragments per IP packet.
	902 packets into 3 fragments.		

Figure 38: Number of fragments each packet is fragmented into for each case.

Of all these measures, we mainly used the ones obtained for the total number of fragments sent for each case.

We will start by analyzing the time results. We can see by looking at Figure 34 that the mean connection time is almost the same for the No Steganography and the F1 Method cases, being this duration bigger for F3 method and for F1 and F3 methods combined.

If we think about the obtained values, we can tell that something seems strange: the logical thinking is that the total connection time for the No Steganography case should be smaller than the connection time for the F1 method, but F1 Method's connection mean time is smaller than in the No Steganography case. Nevertheless, when you think about it, this obtained result makes sense: if you check the programming code for the No Steganography case and the F1 method case (Annex 1), you will see that they are pretty similar. In both cases the whole packet is captured, the random data is introduced and both IP and ICMP header checksums are recalculated. The only difference is that, on F1's case, we sometimes make the whole packet smaller before doing all this, which suppose just an extra instruction. Because of the difference between both methods' programming codes being that small, similar connection times should be expected in our case, and so a result like the one we obtained (with a difference in their mean times that is smaller than a second) is totally plausible.

We should notice that, if we hadn't used modified ICMP packets for the No Steganography case, the obtained times would have differed more, with the No Steganography case's time being probably a little bit smaller than the F1 method's time.

We also see that the time obtained for the F3 method is considerably bigger than the times obtained for the No Steganography and F1 methods: the connection last for around 5 or 6 seconds more. This is logical, since F3 Method is the most complex of the used methods, in which we have to calculate in both sides of the connection a hash function over an IS (Identifying Sequence) that changes in every iteration.

So, we have that the No Steganography and F1 Method cases have similar connection times, with F3 method case having a connection time that last for about 5 seconds longer. In the F1+F3 methods case, when both methods are applied simultaneously, the intuition is that the connection will last even longer. However, the resulting connection time is only about 3 seconds longer. Therefore the duration of the connection in case of the joint methods is shorter as in case of F3 applied alone. This implies that if methods F1 and F3 are combined the resulting steganographic cost is lower as compared to the steganographic cost of the single method (the one that introduces higher steganographic cost). This is the effect we call super-position steganography that we wanted to prove with the experiment explained on this paper.

Let us go now with the analysis of the other obtained results.

The first thing that we analyzed is the size of each of the fragments. We prepared the sent packets for this experiment so each and every of the fragments would be of the same size: 500 bytes per fragment (not including the 14 Ethernet header bytes). As expected, in all the captures, the size of all the fragments is 500 bytes, which is the interface fixed MTU for this scenario.

Let us talk now about the total number of fragments sent on each of the connections (Figure 37). When we compare the distribution of the number of fragments per packet the situation is similar to what we obtained when analyzing the times. The F1 method introduces irregularities in the number of fragments per packet, while F3 increases the overall number of fragments per packet. Since the third-party observer does not possess the knowledge of how many fragments the packets will be divided into in advance, the F3 technique can be considered less detectable. However it must be noted that in the joint-method's case the resulting number of fragments per packet is the same as for the case when F3 is applied alone, as we can see in Figure 37. It is the same number of fragments since irregularities introduced by F1 are "smoothed" by the second method making the overall steganographic cost for the F1+F3 methods' case the same as for the F3 method's case. Therefore the overall steganographic cost is not elevated.

We can reach a similar conclusion when analyzing the number of fragments each packet is fragmented into (Figure 38).

6.2.2 Scenario 2: HTTP Connection

Let us go with scenario 2. Here, what we want to see is that, when combined with F8 method, F6 and F7 methods are 0-cost steganographic methods. In this scenario, in order to analyze the steganographic cost we will be measuring the total connection time and the distribution of the HTTP headers size.

You can see the different connection times obtained for each of the steganographic cases in the next figure:

	NoStegHTTP total time (s).	Method F6 total time (s).	Method F7 total time (s).	Method F8 total time (s).	Method F6+F7+F8 total time (s).
	66,826009	73,806067	73,983599	73,252588	72,692545
	66,443482	74,163563	74,020009	72,441612	73,052037
	67,587337	73,899322	73,358913	72,464690	73,849342
	67,587136	72,903730	72,750580	73,820218	73,261461
	67,945304	74,138459	73,042230	73,542648	73,589569
	68,362785	72,515376	73,981025	73,583339	74,834103
	67,575615	74,729640	74,352792	72,738383	74,290865
	68,053174	75,085206	73,456017	72,953182	73,547481
	65,903195	73,661615	73,978436	74,403576	73,548531
	67,159170	73,642392	73,294256	72,582144	73,006379
MEAN	67,3443207	73,854537	73,6217857	73,178238	73,5672313
STD DEVIATION	0,725514862	0,723907853	0,487735009	0,621871387	0,60398567

Figure 39: HTTP Scenario, table of times.

Each obtained time makes reference to the total connection time: from the moment the first SYN TCP packet is sent starting the first connection, until the moment the last ACK is sent closing the last one. Just like we did with the first scenario, we obtained for each case a mean time and a standard deviation. We drew all the time distributions assuming they follow a normal distribution: you can see them below.

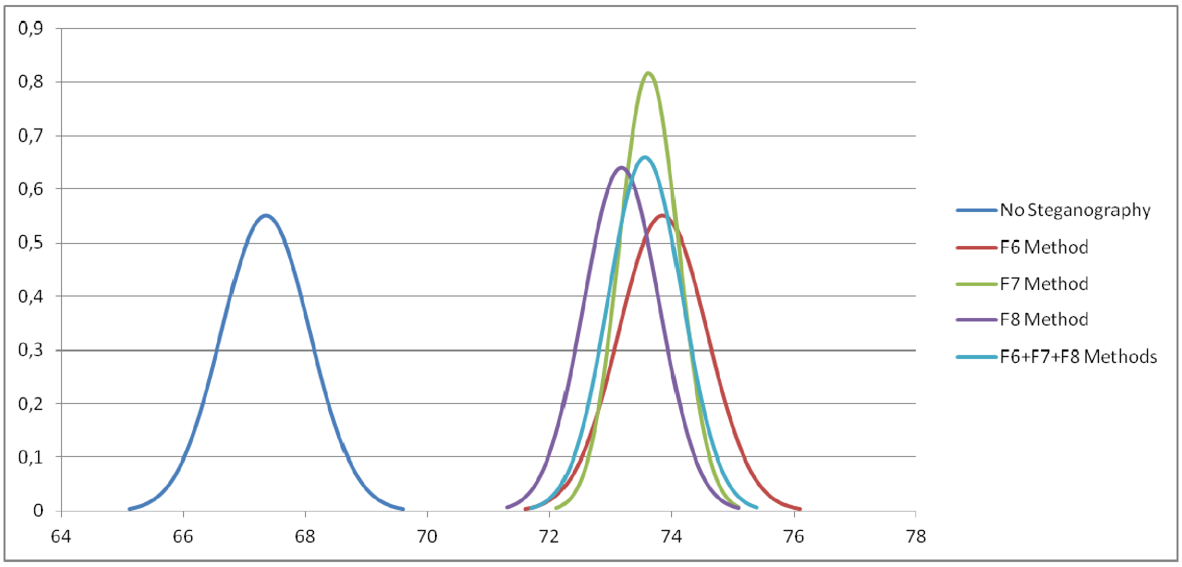


Figure 40: Time distribution for each of the methods, scenario 2.

We also analyzed how the protocol performance was affected by each of the methods, in order to see if we can proof our hypothesis. For it, as we told, we analyzed the HTTP Header length for each case. In the next chart, you can see the obtained results.

HTTP HEADER LENGTH				
No Steg	F6 method	F7 method	F8 method	F6+F7+F8 methods
178 bytes, 900 packets.	178 bytes, 900 packets.	178 bytes, 900 packets.	178 bytes, 497 packets.	178 bytes, 511 packets.
			154 bytes, 403 packets.	154 bytes, 389 packets.

Figure 41: HTTP header lengths for each of the cases.

We will start talking about the obtained times. By inspecting the overall connection time we can observe that after applying each of the steganographic methods alone the resulting connection time increases by about 6 seconds when comparing it with the No Steganography case. The same result is achieved for the F6+F7+F8 combined methods case. Therefore simultaneous utilization of all three methods does not influence the total connection time. This is the *zero cost steganography* phenomena we wanted to prove with this experiment, and we can understand why are we calling it like that with the obtained results: adding additional methods to the existing one does not influence the resulting steganographic cost.

If we analyze the obtained header lengths, we can see that when using F8 method alone some irregularities are introduced in HTTP headers sizes. For the F6+F7+F8 combined methods' case, where two more methods are added, the irregularities are still present but they are similar as in case of F8 method applied alone. Thus we can conclude that, in the F6+F7+F8 combined methods' case, the resulting total steganographic cost is not higher than in case of method F8 applied alone.

7. CONCLUSIONS

Our obtained experimental results show that it is feasible to combine multiple steganographic methods to the same carrier in a way that the overall steganographic cost caused by these methods is lower as in case of a separate combination of these methods (super-position steganography). Results additionally show that multiple steganographic methods can be combined with another method without causing any additional cost, which is a special case of super-position steganography called zero cost steganography.

This is an important result: knowing about the existence of these two phenomena will allow steganalayzers to think about possible ways of countering future steganographic techniques that will try to exploit one of these two phenomena where more than one steganographic method is combined without making the total steganographic cost increase.

We must also remark that steganography has also some legal usages, so a lot of legal organizations (like governments for example) will be able to use superposition steganography or zero cost steganography, in order to increase the steganographic bandwidth they are obtaining with their actual methods without increasing the steganographic cost.

The results obtained in this thesis have served as a basis of a scientific paper which is currently under consideration for publication in Security and Communication Networks Journal [11].

ANNEX 1: PROGRAMMING CODE

We will be including all the implemented programming code in this section. We will start by showing the parts of the code that are common to all of the cases, in both transmission and reception: needed libraries and common variables, the code related with the GTK graphic interface, the different functions we implemented and the main program. After that, we will focus on one of the functions, function which is different depending on the used method, and on if we are on the sending or on the receiving side. We will finally present the full compilation line that we have to use in order to compile our programs.

- Needed libraries and common variables

```
#include <stdlib.h>
#include <netinet/in.h>
#include <linux/netfilter.h>
#include <libnetfilter_queue/libnetfilter_queue.h>
#include <gtk/gtk.h>

#include <iostream>

using namespace std;

//Variables
//Variables must be static and global
//so they can be seen in every threat

#define YES_IT_IS    (1)
#define NO_IT_IS_NOT (0)
typedef unsigned short u16;
char buf[4096];
unsigned char dane[4096];
int rv;
int i, tmp;

struct nfq_handle *h;
struct nfq_q_handle *qh;
struct nfnl_handle *nh;
int fd;


//global GTK variables
GtkWidget *window;
GtkWidget *label_iptables_control;
GtkWidget *text;
GtkWidget *vbox;
GtkTextBuffer *buffer;
GtkTextIter iter;

GtkTextMark *insert_mark;
GtkWidget *table;
GtkWidget *button_iptables_INPUT_add;
GtkWidget *button_iptables_INPUT_delete;
GtkWidget *button_iptables_OUTPUT_add;
GtkWidget *button_iptables_OUTPUT_delete;
GtkWidget *button_iptables_FORWARD_add;
GtkWidget *button_iptables_FORWARD_delete;
GtkWidget *port;
GtkWidget *port_label;
```

- Function unsignedShortToInt, which converses an unsigned short char variable into an int variable

```
//Short to Int
static unsigned int unsignedShortToInt(unsigned char b[])
{
    int i = 0;
    i |= b[0] & 0xFF;
    i <<= 8;
    i |= b[1] & 0xFF;
    return i;
}
```

- Function csum, which calculates the total sum for the IP/ICMP/TCP checksum

```
//Part of checksum
unsigned short csum (unsigned short *buf, int nwords)
{
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--)
    {
        sum += *buf++;
    }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);

    return ((unsigned short) ~sum);
}
```

- GTK related functions.

```
typedef struct
{
    GtkWidget *label;
    int what;
} yes_or_no_args;

G_LOCK_DEFINE_STATIC (yes_or_no);
static volatile int yes_or_no = YES_IT_IS;

void destroy(GtkWidget *widget, gpointer data)
{
    gtk_main_quit();
}
```

```
//main thread for net
void *network_thread(void *args)
{

    //main loop for packets
    while ((rv = recv(fd, buf, sizeof(buf), 0)) && rv >= 0) {

   //packet handler
      nfq_handle_packet(h, buf, rv);

    }
}


//action for button IPtables OUTPUT Add
void akcja_iptables_OUTPUT_add(GtkWidget *widget, gpointer
window)
{
    system("iptables -A OUTPUT -p icmp -j QUEUE");
    gtk_text_buffer_insert(buffer, &iter, "[ON] Enabled IPtables
OUTPUT\n", -1);
    gtk_widget_set_sensitive(button_iptables_INPUT_add,false);
    gtk_widget_set_sensitive(button_iptables_OUTPUT_add,false);
    gtk_widget_set_sensitive(button_iptables_FORWARD_add,false);

gtk_widget_set_sensitive(button_iptables_INPUT_delete,false);

gtk_widget_set_sensitive(button_iptables_FORWARD_delete,false);



}

//action for button IPtables OUTPUT Delete
void akcja_iptables_OUTPUT_delete(GtkWidget *widget, gpointer
window)
{
    system("iptables -D OUTPUT -p icmp -j QUEUE");
    gtk_text_buffer_insert(buffer, &iter, "[OFF] Disabled
IPtables OUTPUT\n", -1);
    gtk_widget_set_sensitive(button_iptables_INPUT_add,true);
    gtk_widget_set_sensitive(button_iptables_OUTPUT_add,true);
    gtk_widget_set_sensitive(button_iptables_FORWARD_add,true);
    gtk_widget_set_sensitive(button_iptables_INPUT_delete,true);

gtk_widget_set_sensitive(button_iptables_OUTPUT_delete,true);

gtk_widget_set_sensitive(button_iptables_FORWARD_delete,true);

}



//action for button IPtables INPUT Add
void akcja_iptables_INPUT_add(GtkWidget *widget, gpointer
window)
{
    system("iptables -A INPUT -p icmp -j QUEUE");
    gtk_text_buffer_insert(buffer, &iter, "[ON] Enabled IPtables
INPUT\n", -1);
    gtk_widget_set_sensitive(button_iptables_INPUT_add,false);
    gtk_widget_set_sensitive(button_iptables_OUTPUT_add,false);
    gtk_widget_set_sensitive(button_iptables_FORWARD_add,false);

gtk_widget_set_sensitive(button_iptables_OUTPUT_delete,false);

gtk_widget_set_sensitive(button_iptables_FORWARD_delete,false);
}

//action for button IPtables INPUT Delete
void akcja_iptables_INPUT_delete(GtkWidget *widget, gpointer
window)
{
    system("iptables -D INPUT -p icmp -j QUEUE");
    gtk_text_buffer_insert(buffer, &iter, "[OFF] Disabled
IPtables INPUT\n", -1);
    gtk_widget_set_sensitive(button_iptables_INPUT_add,true);
    gtk_widget_set_sensitive(button_iptables_OUTPUT_add,true);
    gtk_widget_set_sensitive(button_iptables_FORWARD_add,true);
    gtk_widget_set_sensitive(button_iptables_INPUT_delete,true);

gtk_widget_set_sensitive(button_iptables_OUTPUT_delete,true);

gtk_widget_set_sensitive(button_iptables_FORWARD_delete,true);

}



//action for button IPtables FORWARD Add
void akcja_iptables_FORWARD_add(GtkWidget *widget, gpointer
window)
{
    system("iptables -A FORWARD -p icmp -j QUEUE");
    gtk_text_buffer_insert(buffer, &iter, "[ON] Enabled IPtables
FORWARD\n", -1);
    gtk_widget_set_sensitive(button_iptables_INPUT_add,false);
    gtk_widget_set_sensitive(button_iptables_OUTPUT_add,false);
    gtk_widget_set_sensitive(button_iptables_FORWARD_add,false);

gtk_widget_set_sensitive(button_iptables_INPUT_delete,false);

gtk_widget_set_sensitive(button_iptables_OUTPUT_delete,false);
}

//action for button IPtables FORWARD Delete
void akcja_iptables_FORWARD_delete(GtkWidget *widget, gpointer
window)
{
    system("iptables -D FORWARD -p icmp -j QUEUE");
    gtk_text_buffer_insert(buffer, &iter, "[OFF] Disabled
IPtables FORWARD\n", -1);
    gtk_widget_set_sensitive(button_iptables_INPUT_add,true);
    gtk_widget_set_sensitive(button_iptables_OUTPUT_add,true);
    gtk_widget_set_sensitive(button_iptables_FORWARD_add,true);
    gtk_widget_set_sensitive(button_iptables_INPUT_delete,true);

gtk_widget_set_sensitive(button_iptables_OUTPUT_delete,true);

gtk_widget_set_sensitive(button_iptables_FORWARD_delete,true);

}
```

- <u>Main function, which calls the graphic interface for modifying the packets</u>

```
//main program function
int main(int argc, char *argv[])
{

system("clear");

    //adding netfilter module for kernel
    int result =  system("modprobe iptable_filter");

    GError *error = NULL;
    yes_or_no_args yes_args, no_args;

    /* init threads */
    g_thread_init(NULL);
    gdk_threads_init();

    /* init gtk */
    gtk_init(&argc, &argv);

    /* create a window */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW(window), "Packet
modifier");
    gtk_window_set_position(GTK_WINDOW(window),
GTK_WIN_POS_CENTER);
    gtk_widget_set_size_request (window, 450, 450);
    gtk_window_set_resizable(GTK_WINDOW(window), TRUE);

    text = gtk_text_view_new();
    gtk_text_view_set_editable(GTK_TEXT_VIEW (text), FALSE);
    port = gtk_entry_new();
    gtk_entry_set_text(GTK_ENTRY(port),"1234");
   port_label = gtk_label_new("DST UDP PORT:");

    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));

    gtk_text_buffer_get_iter_at_offset(buffer, &iter, 0);


    GtkWidget *scrolled_window;
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW
(scrolled_window),GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);

    gtk_scrolled_window_add_with_viewport (GTK_SCROLLED_WINDOW
(scrolled_window), text);
```

```
    g_signal_connect(window, "destroy",G_CALLBACK(destroy),
NULL);

    gtk_container_set_border_width(GTK_CONTAINER (window), 10);

    table = gtk_table_new(4, 3, TRUE);
    gtk_table_set_row_spacings(GTK_TABLE(table), 2);
    gtk_table_set_col_spacings(GTK_TABLE(table), 2);


gtk_table_attach_defaults(GTK_TABLE(table),
label_iptables_control, 1, 2, 0, 1 );
    button_iptables_OUTPUT_add =
gtk_button_new_with_label("OUTPUT [ON]");
    button_iptables_OUTPUT_delete =
gtk_button_new_with_label("OUTPUT [OFF]");

    button_iptables_INPUT_add = gtk_button_new_with_label("INPUT
[ON]");
    button_iptables_INPUT_delete =
gtk_button_new_with_label("INPUT [OFF]");

    button_iptables_FORWARD_add =
gtk_button_new_with_label("FORWARD [ON]");
    button_iptables_FORWARD_delete =
gtk_button_new_with_label("FORWARD [OFF]");


    gtk_table_attach_defaults(GTK_TABLE(table),
button_iptables_OUTPUT_add , 1, 2, 1, 2 );
    gtk_table_attach_defaults(GTK_TABLE(table),
button_iptables_OUTPUT_delete , 2, 3, 1, 2 );

    gtk_table_attach_defaults(GTK_TABLE(table),
button_iptables_INPUT_add , 1, 2, 2, 3 );
    gtk_table_attach_defaults(GTK_TABLE(table),
button_iptables_INPUT_delete , 2, 3, 2, 3 );

    gtk_table_attach_defaults(GTK_TABLE(table),
button_iptables_FORWARD_add , 1, 2, 3, 4 );
    gtk_table_attach_defaults(GTK_TABLE(table),
button_iptables_FORWARD_delete , 2, 3, 3, 4 );

    gtk_table_attach_defaults(GTK_TABLE(table), port_label , 0,
1, 1, 2 );
    gtk_table_attach_defaults(GTK_TABLE(table), port , 0, 1, 2,
3 );

    vbox = gtk_vbox_new(FALSE, 1);
    gtk_container_add(GTK_CONTAINER(window), vbox);



    gtk_box_pack_start(GTK_BOX(vbox), scrolled_window, TRUE,
TRUE, 0);
    gtk_box_pack_start(GTK_BOX(vbox), table, FALSE, FALSE, 0);

    gtk_widget_show_all(window);
    //acctions for buttons
    g_signal_connect(G_OBJECT(button_iptables_OUTPUT_add), 
"clicked", G_CALLBACK(akcja_iptables_OUTPUT_add), (gpointer) 
window);
    g_signal_connect(G_OBJECT(button_iptables_OUTPUT_delete), 
"clicked", G_CALLBACK(akcja_iptables_OUTPUT_delete), (gpointer) 
window);
    g_signal_connect(G_OBJECT(button_iptables_INPUT_add), 
"clicked", G_CALLBACK(akcja_iptables_INPUT_add), (gpointer) 
window);
    g_signal_connect(G_OBJECT(button_iptables_INPUT_delete), 
"clicked", G_CALLBACK(akcja_iptables_INPUT_delete), (gpointer) 
window);
    g_signal_connect(G_OBJECT(button_iptables_FORWARD_add), 
"clicked", G_CALLBACK(akcja_iptables_FORWARD_add), (gpointer) 
window);
    g_signal_connect(G_OBJECT(button_iptables_FORWARD_delete), 
"clicked", G_CALLBACK(akcja_iptables_FORWARD_delete), (gpointer) 
window);





    //printf("opening library handle\n");
    h = nfq_open();
    if (!h) {
        fprintf(stderr, "error during nfq_open()\n");
        exit(1);
    }

    //printf("unbinding existing nf_queue handler for AF_INET 
(if any)\n");
    if (nfq_unbind_pf(h, AF_INET) < 0) {
        fprintf(stderr, "error during nfq_unbind_pf()\n");
        exit(1);
    }

    //printf("binding nfnetlink_queue as nf_queue handler for 
AF_INET\n");
    if (nfq_bind_pf(h, AF_INET) < 0) {
        fprintf(stderr, "error during nfq_bind_pf()\n");
        exit(1);
    }

    //printf("binding this socket to queue '0'\n");
    qh = nfq_create_queue(h,  0, &cb, NULL);
    if (!qh) {
        fprintf(stderr, "error during nfq_create_queue()\n");
        exit(1);
    }

    //printf("setting copy_packet mode\n");
    if (nfq_set_mode(qh, NFQNL_COPY_PACKET, 0xffff) < 0) {
        fprintf(stderr, "can't set packet_copy mode\n");
        exit(1);
    }

    nh = nfq_nfnlh(h);
    fd = nfnl_fd(nh);


    //therad for net
    if (!g_thread_create(network_thread, &yes_args, FALSE,
&error))
    {
        g_printerr ("Failed to create NET thread: %s\n", error-
>message);
        return 1;
    }


    //main GTK loop
    gdk_threads_enter();
    gtk_main();

    gdk_threads_leave();
//}

    return 0;
}
```

- <u>ICMP No Steganography case, main function for modifying the packets,sender side</u>

```
int trigger=2;


//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg
*nfmsg,
              struct nfq_data *nfa, void *data)
{

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

      }

if (trigger>=0)
      {
      srand(8);
      }

for (int i=28; i<ret; i++)
      {
      dane[i]=rand()%16;
      }
trigger=trigger-1;

            //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

            //ICMP checksum
            u16 icmp_message[(ret-20)/2];

            dane[22] = 0x00;
            dane[23] = 0x00;

            for(int i=0; i<(ret-20)/2; i++)
            {
                icmp_message[i] =
unsignedShortToInt(dane+20+2*i);
            }

            u16 icmp_message_sum = csum(icmp_message,(ret-
20)/2);
            u16 icmp_message_sum_begin = icmp_message_sum;
            icmp_message_sum_begin >>= 8;
            dane[22] = icmp_message_sum_begin;
            dane[23] = icmp_message_sum;

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -
1);

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>ICMP No Steganography case, main function for modifying the packets, receiver side</u>

```
//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)
```

```
        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];
    }

if (trigger==0)
{
dane[1]=dane[1];
std::cout<< " ";
}

else if (trigger!=0)
{
trigger=trigger-1;
}

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>F1 Method case, main function for modifying the packets, sender side</u>

```
//Introduce a message made with 1’s and 0’s.

int message[]={}

int position=0;

int length=sizeof(message)/sizeof(message[0]);

int trigger=2;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
           struct nfq_data *nfa, void *data)
   {

       char *data2;
       int ret = nfq_get_payload(nfa, &data2);
       if (ret >= 0)

           for(int i=0; i<ret; i++)
           {
               dane[i] = (unsigned char)data2[i];

       }

   if (trigger>=0)
       {
       srand(8);
       }


   if (message[position]==1)
       {
       ret=1460;
       }

   for (int i=28; i<ret; i++)
```

```
    {
    dane[i]=rand()%16;
    }

            //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

            //ICMP checksum
            u16 icmp_message[(ret-20)/2];

            dane[22] = 0x00;
            dane[23] = 0x00;

            for(int i=0; i<(ret-20)/2; i++)
            {
                icmp_message[i] =
unsignedShortToInt(dane+20+2*i);
            }

            u16 icmp_message_sum = csum(icmp_message,(ret-
20)/2);
            u16 icmp_message_sum_begin = icmp_message_sum;
            icmp_message_sum_begin >>= 8;
            dane[22] = icmp_message_sum_begin;
            dane[23] = icmp_message_sum;


if (trigger<=0)
{
position=position+1;
}

trigger=trigger-1;

if(position==length)
    {
    position=0;
    //system("iptables -D OUTPUT -p icmp -j QUEUE");
    }


gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>F1 method case, main function for modifying the packets, receiver side</u>

```
int message[4096];
int position=0;
int trigger=2;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

if (trigger==2)
    {
    std::cout<< "\nThe secret message received is, starting from
the third digit: ";
    }

if (ret==1940)
    {
    message[position]=0;
    std::cout<<message[position];
    }

else if (ret==1460)
    {
    message[position]=1;
    std::cout<<message[position];
    }

if (trigger<=0)
{
position=position+1;
}

trigger=trigger-1;

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>F3 method case, main function for modifying the packets, sender side</u>

```
//Introduce a message made with 1's and 0's.

int message[]={}

int length=sizeof(message)/sizeof(message[0]);
int position=0;
```

```
unsigned char sk[2]={0x56,0x65};
char IS[5]={0x00,0x00,0x00,0x00,0x00};
unsigned long int change=0;
char ISaux[10];
int trigger=2;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

if (trigger>=0)
    {
         srand(8);
    }

//PACKET modification
for (int i=28; i<500; i++)
    {
    dane[i]=rand()%16;
    }

for (int i=980; i<1940; i++)
    {
    dane[i]=rand()%16;
    }

for (int i=0; i<5; i++)
    {
        if (i<2)
        {
        IS[i]=sk[i];
        }
    else if (i==2)
        {
        IS[i]=0x3c;
        }
    else if (2<i<5)
        {
        IS[i]=dane[i+1];
        }
    }

std::string chain (IS);
std::hash<std::string> final;

size_t hashed=final(IS);

change=hashed;

sprintf(ISaux,"%ld",change);

for (int i=0;i<4;i++)
```

```
    {
    dane[i+500]=ISaux[i];
    }


for (int i=504; i<980; i++)
    {
    dane[i]=message[i-504+position];
    }

if (trigger<=0)
{
position=position+476;
}

trigger=trigger-1;

if(position>=length-476)
    {
    position=0;
    //system("iptables -D OUTPUT -p icmp -j QUEUE");
    }

            //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

            //ICMP checksum
            u16 icmp_message[960];

            dane[22] = 0x00;
            dane[23] = 0x00;

            for(int i=0; i<960; i++)
            {
                icmp_message[i] = unsignedShortToInt(dane+20+2*i);
            }

            u16 icmp_message_sum = csum(icmp_message,960);
            u16 icmp_message_sum_begin = icmp_message_sum;
            icmp_message_sum_begin >>= 8;
            dane[22] = icmp_message_sum_begin;
            dane[23] = icmp_message_sum;

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);



    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>F3 Method case, main function for modifying the packets, receiver side</u>

```
int message[4096];
int length=324573;
int position=0;
unsigned char sk[2]={0x56,0x65};
char IS[5]={0x00,0x00,0x00,0x00,0x00};
unsigned long int change=0;
char ISaux[10];
unsigned char IScomp[10];
int trigger=2;



//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }


//PACKET modification
if (trigger==2)
    {
    std::cout<< "\nThe secret message received is, starting from
the third digit: ";
    }

for (int i=0; i<500; i++)
    {
    dane2[i]=dane[i];
    }

for (int i=980; i<1940; i++)
    {
    dane2[i-480]=dane[i];
    }


for (int i=0;i<5;i++)
    {
    if (i<2)
        {
        IS[i]=sk[i];
        }
    else if (i==2)
        {
        IS[i]=0x3c;
        }
    else if (2<i<5)
```

```
            {
            IS[i]=dane[i+1];
            }
        }

    std::string chain (IS);
    std::hash<std::string> final;

    size_t hashed=final(IS);

    change=hashed;

    sprintf(ISaux,"%ld",change);

    for (int i=0; i<4;i++)
        {
        IScomp[i]=ISaux[i];
        }

    if ((dane[500]==(IScomp[0])) && (dane[501]==(IScomp[1])) &&
    (dane[502]==(IScomp[2])) && (dane[503]==(IScomp[3])))

        {
        //length=dane[505];
        for (int i=0; i<476; i++)
            {
            message[i+position]=dane[i+504];
            std::cout<<message[i+position];
            }
        if (trigger<=0)
            {
            position=position+476;
            }
        }

    if (position==length-1)
        {
            system("iptables -D INPUT -p icmp -j QUEUE");
        std::cout<< ".\n";
        }

    trigger=trigger-1;

    gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);


        return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
    }
```

When combining F1 and F3 Methods, we use the previously shown code combined.

- <u>F6 Method case, main function for modifying the packets, sender side</u>

```
    //Introduce a message made with 1’s and 0’s.

    int message[]={};
    int length=sizeof(message)/sizeof(message[0]);
    int puntero=143;
    int punteromessage=0;
    int trigger=1;
```

```
//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

//Checking if this is desired port p will be port number from
window field. packet will be sent without modification
int p = atoi((char*) gtk_entry_get_text(GTK_ENTRY(port)));
//porrt is 16 bit number so two unsigned char for 8 older bits and
and 8 younger bits must be compared to desired port
if ((dane[22]==(p>>8))&&(dane[23]==(p&255))&& (ret>60))
{
if (trigger==0)
{
for (int i=0; i<5; i++)
    {

    if (message[punteromessage+i]==0)
        {
        if (dane[puntero]<=0x5a)
            {
            dane[puntero]=dane[puntero]+0x20;
            }
        }

    else if (message[punteromessage+i]==1)
        {
        if (dane[puntero]>0x5a)
            {
            dane[puntero]=dane[puntero]-0x20;
            }
        }

    puntero=puntero+1;
    }

puntero=143;
punteromessage=punteromessage+5;
length=length-5;
if (length==0)
    {
    length=sizeof(message)/sizeof(message[0]);
    punteromessage=0;
    //system("iptables -D OUTPUT -p TCP -j QUEUE");
    }

            //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

        //TCP checksum
            u16 tcp_header[((ret-20)/2)+6];

            dane[36] = 0x00;
            dane[37] = 0x00;

        danetcp[0] = 0x00;
        danetcp[1] = dane[9];

            for(int i=0; i<(ret-20)/2; i++)
            {
                tcp_header[i] = unsignedShortToInt(dane+20+2*i);
            }

            for(int i=0; i<4; i++)
            {
                tcp_header[i+(ret-20)/2] =
unsignedShortToInt(dane+12+2*i);
            }

        tcp_header[((ret-20)/2)+4]= unsignedShortToInt(danetcp);

        tcp_header[((ret-20)/2)+5]= ret-20;

            u16 tcp_header_sum = csum(tcp_header,((ret-20)/2)+6);
            u16 tcp_header_sum_begin = tcp_header_sum;
            tcp_header_sum_begin >>= 8;
            dane[36] = tcp_header_sum_begin;
            dane[37] = tcp_header_sum;
}

else if (trigger!=0)
{
trigger=trigger-1;
}

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);
}

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- F6 method case, main function for modifying the packets, receiver side

```
int message[4096];
int puntero=143;
int punteromessage=0;
int trigger=1;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

//Checking if this is desired port p will be port number from
window field. packet will be sent without modification
int p = atoi((char*) gtk_entry_get_text(GTK_ENTRY(port)));
//porrt is 16 bit number so two unsigned char for 8 older bits and
and 8 younger bits must be compared to desired port
if ((dane[22]==(p>>8))&&(dane[23]==(p&255))&& (ret>60))
{

if (trigger==0)
{
for (int i=0; i<5; i++)
    {
    if (dane[puntero+i]>0x5a)
        {
        message[punteromessage]=0;
        std::cout<<message[punteromessage];
        punteromessage=punteromessage+1;
        }

    else if (dane[puntero+i]<=0x5a)
        {
        message[punteromessage]=1;
        std::cout<<message[punteromessage];
        punteromessage=punteromessage+1;
        }
    }

}

else if (trigger!=0)
{
std::cout<< "\nThe secret message received is: ";
trigger=trigger-1;
}

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);
}

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>F7 Method case, main function for modifying the packets, Sender side</u>

```
//Introduce a message made with 1’s and 0’s.


int message[]={};
int length=sizeof(message)/sizeof(message[0]);
int puntero=175;
int punteromessage=0;
int trigger=1;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

//Checking if this is desired port p will be port number from
window field. packet will be sent without modification
int p = atoi((char*) gtk_entry_get_text(GTK_ENTRY(port)));
//porrt is 16 bit number so two unsigned char for 8 older bits and
and 8 younger bits must be compared to desired port
if ((dane[22]==(p>>8))&&(dane[23]==(p&255))&& (ret>60))
{

if (trigger==0)
{

if (message[punteromessage]==1)
    {
    for (int i=0; i<13;i++)
        {
        daneaux[i]=dane[puntero+i];
        }

    for (int i=0; i<16;i++)
        {
        dane[puntero+i]=dane[puntero+i+13];
        }

    for (int i=16;i<29;i++)
        {
```

```
        dane[puntero+i]=daneaux[i-16];
        }
    }

punteromessage=punteromessage+1;
length=length-1;
if (length==0)
    {
    length=sizeof(message)/sizeof(message[0]);
    punteromessage=0;
    //system("iptables -D OUTPUT -p TCP -j QUEUE");
    }

           //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

        //TCP checksum
            u16 tcp_header[((ret-20)/2)+6];

            dane[36] = 0x00;
            dane[37] = 0x00;

        danetcp[0] = 0x00;
        danetcp[1] = dane[9];

            for(int i=0; i<(ret-20)/2; i++)
            {
                tcp_header[i] = unsignedShortToInt(dane+20+2*i);
            }

            for(int i=0; i<4; i++)
            {
                tcp_header[i+(ret-20)/2] =
unsignedShortToInt(dane+12+2*i);
            }

        tcp_header[((ret-20)/2)+4]= unsignedShortToInt(danetcp);

        tcp_header[((ret-20)/2)+5]= ret-20;

            u16 tcp_header_sum = csum(tcp_header,((ret-20)/2)+6);
            u16 tcp_header_sum_begin = tcp_header_sum;
            tcp_header_sum_begin >>= 8;
            dane[36] = tcp_header_sum_begin;
            dane[37] = tcp_header_sum;
}

else if (trigger!=0)
```

```
{
trigger=trigger-1;
}

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);
}


    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- F7 method case, main function for modifying the packets, receiver side

```
int message[4096];
int puntero=175;
int punteromessage=0;
int trigger=1;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

//Checking if this is desired port p will be port number from
window field. packet will be sent without modification
int p = atoi((char*) gtk_entry_get_text(GTK_ENTRY(port)));
//porrt is 16 bit number so two unsigned char for 8 older bits and
and 8 younger bits must be compared to desired port
if ((dane[22]==(p>>8))&&(dane[23]==(p&255))&& (ret>60))
{

if (trigger==0)
{

if (dane[puntero]==0x41)
    {
    message[punteromessage]=0;
    std::cout<<message[punteromessage];
    punteromessage=punteromessage+1;
    }

else if (dane[puntero]==0x48)
    {
    message[punteromessage]=1;
    std::cout<<message[punteromessage];
    punteromessage=punteromessage+1;
    }
}

else if (trigger!=0)
{
std::cout<< "\nThe secret message received is: ";
trigger=trigger-1;
}



gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);
}


    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- <u>F8 method case, main function for modifying the packets, sender side</u>

```
//Introduce a message made with 1’s and 0’s.

int message[]={}
int length=sizeof(message)/sizeof(message[0]);
int puntero=175;
int punteromessage=0;
int trigger=1;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

//Checking if this is desired port p will be port number from
window field. packet will be sent without modification
int p = atoi((char*) gtk_entry_get_text(GTK_ENTRY(port)));
//porrt is 16 bit number so two unsigned char for 8 older bits and
and 8 younger bits must be compared to desired port
if ((dane[22]==(p>>8))&&(dane[23]==(p&255))&& (ret>60))
{

if (trigger==0)
{

if (message[punteromessage]==1)
    {

        dane[ret-26]=0x0d;
        dane[ret-25]=0x0a;
    ret=ret-24;
    dane[3]=ret;
    }

punteromessage=punteromessage+1;
length=length-1;
if (length==0)
    {
    length=sizeof(message)/sizeof(message[0]);
    punteromessage=0;
     //system("iptables -D OUTPUT -p TCP -j QUEUE");
    }

            //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

        //TCP checksum
            u16 tcp_header[((ret-20)/2)+6];

            dane[36] = 0x00;
            dane[37] = 0x00;

        danetcp[0] = 0x00;
        danetcp[1] = dane[9];

            for(int i=0; i<(ret-20)/2; i++)
            {
                tcp_header[i] = unsignedShortToInt(dane+20+2*i);
            }

            for(int i=0; i<4; i++)
            {
                tcp_header[i+(ret-20)/2] =
unsignedShortToInt(dane+12+2*i);
            }

        tcp_header[((ret-20)/2)+4]= unsignedShortToInt(danetcp);

        tcp_header[((ret-20)/2)+5]= ret-20;

            u16 tcp_header_sum = csum(tcp_header,((ret-20)/2)+6);
            u16 tcp_header_sum_begin = tcp_header_sum;
            tcp_header_sum_begin >>= 8;
            dane[36] = tcp_header_sum_begin;
            dane[37] = tcp_header_sum;

}
```

```
else if (trigger!=0)
{
trigger=trigger-1;
}

gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);
}


    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

- F8 method case, main function for modifying the packets, receiver side

```
int message[4096];
int puntero=175;
int punteromessage=0;
int trigger=1;

//MAIN FUNCTION FOR modyfying packets
static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
              struct nfq_data *nfa, void *data)
{


    u_int32_t id = print_pkt(nfa);
    //printf("entering callback\n");

    char *data2;
    int ret = nfq_get_payload(nfa, &data2);
    if (ret >= 0)

        for(int i=0; i<ret; i++)
        {
            dane[i] = (unsigned char)data2[i];

    }

//Checking if this is desired port p will be port number from
window field. packet will be sent without modification
int p = atoi((char*) gtk_entry_get_text(GTK_ENTRY(port)));
//porrt is 16 bit number so two unsigned char for 8 older bits and
and 8 younger bits must be compared to desired port
if ((dane[22]==(p>>8))&&(dane[23]==(p&255))&& (ret>60))
{

if (trigger==0)
{
if (ret==230)
    {
    message[punteromessage]=0;
    std::cout<<message[punteromessage];
    punteromessage=punteromessage+1;
    }

else if (ret<230)
    {
    message[punteromessage]=1;
    std::cout<<message[punteromessage];
    punteromessage=punteromessage+1;
    for (int i=0;i<24;i++)
```

```
        {
        dane[214+i]=header[i];
        }
    ret=ret+24;
    dane[3]=ret;
    }

            //IP checksum
            u16 ip_header[10];

            dane[10] = 0x00;
            dane[11] = 0x00;

            for(int i=0; i<10; i++)
            {
                ip_header[i] = unsignedShortToInt(dane+2*i);
            }

            u16 ip_header_sum = csum(ip_header,10);
            u16 ip_header_sum_begin = ip_header_sum;
            ip_header_sum_begin >>= 8;
            dane[10] = ip_header_sum_begin;
            dane[11] = ip_header_sum;

        //TCP checksum
            u16 tcp_header[((ret-20)/2)+6];

            dane[36] = 0x00;
            dane[37] = 0x00;

        danetcp[0] = 0x00;
        danetcp[1] = dane[9];

            for(int i=0; i<(ret-20)/2; i++)
            {
                tcp_header[i] = unsignedShortToInt(dane+20+2*i);
            }

            for(int i=0; i<4; i++)
            {
                tcp_header[i+(ret-20)/2] =
unsignedShortToInt(dane+12+2*i);
            }

        tcp_header[((ret-20)/2)+4]= unsignedShortToInt(danetcp);

        tcp_header[((ret-20)/2)+5]= ret-20;

            u16 tcp_header_sum = csum(tcp_header,((ret-20)/2)+6);
            u16 tcp_header_sum_begin = tcp_header_sum;
            tcp_header_sum_begin >>= 8;
            dane[36] = tcp_header_sum_begin;
            dane[37] = tcp_header_sum;

}

else if (trigger!=0)
{
std::cout<< "\nThe secret message received is: ";
trigger=trigger-1;
}
```

```
gtk_text_buffer_insert(buffer, &iter, "Packet captured\n", -1);
}

    return nfq_set_verdict(qh, id, NF_ACCEPT, ret, dane);
}
```

When combining F6, F7 and F8 Methods, we use the previously shown code combined.

- Main Compilation command

  g++ -std=c++0x –o pogram_name program_name.cc –lnfnetlink –lnetfilter_queue –lgthread-2.0 –w ‘pkg-config gtk+-2.0 --cflags --libs’

ANNEX 2: BIBLIOGRAPHY AND REFERENCES

[1] M. J. Gross, “Exclusive: Operation Shady RAT - unprecedented cyber-espionage campaign and intellectual-property bonanza,” *Vanity Fair*, August 2011.

[2] E. Zielinska, W. Mazurczyk, K. Szczypiorski – *Development trends in steganography*, Warsaw University of Technology, Poland.

[3] D. Alperovitch, *Revealed: operation Shady RAT*. McAfee, 2011, http://www.mcafee.com/us/resources/-white-papers/wp-operation-shady-rat.pdf.

[4] K. Srivastava, “Congress wants answers on world‟s largest security breach,” August 2011, http://-www.mobiledia.com/news/102480.html.

[5] H. Lau, “The truth behind the Shady RAT,” *McAffe report*, August 2011, http://www.symantec.com/-connect/blogs/truth-behind-shady-rat.

[6] N. Falliere, L. Murchu, and E. Chien, “W32.Stuxnet dossier,” *White paper, Symantec Corp., Security Response*, 2011.

[7] T. Chen, “Stuxnet, the real start of cyber warfare?[editor‟s note],” *IEEE Network*, vol. 24, no. 6, pp. 2–3, 2010.

[8] D. Goodin, “Duqu spawned by ‟well-funded team of competent coders‟ - World‟s first known modular rootkit does steganography, too,” *The Register*, November 2011.

[9] Symantec, “W32.Duqu - the precursor to the next Stuxnet (version 1.4),” November 2011, http://-www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/-w32_duqu_the_precursor_to_the_next_stuxnet_research.pdf.

[10] P. Wayner, *Disappearing Cryptography - Information Hiding: Steganography & Watermarking*. Morgan Kaufmann, 2009.

[11] W. Mazurczyk, S. Wendzel, I. Azagra, K. Szczypiorski – *On Importance of Steganographic Cost for Network Steganography*, Warsaw University of Technology, Poland; Fraunhofer FKIE,Germany; Universidad Pública de Navarra, Spain.

[12] J. Kelley, *Terror groups hide behind Web encryption*, ser. USA Today, may 2001, http://-www.usatoday.com/tech/news/2001-02-05-binladen.htm.

[13] D. Sieberg, “Bin Laden exploits technology to suit his needs,” *CNN*, September 2001, http://-edition.cnn.com/2001/US/09/20/inv.terrorist.search/.

[14] T. Kellen, “Hiding in plain view: Could steganography be a terrorist tool?” *SANS Institute InfoSec Reading Room*, 2001, http://www.sans.org/reading_room/whitepapers/stenganography/hiding-plain-view-steganography-terrorist-tool_551.

[15] R. Bryant, Ed., *Investigating Digital Crime*. John Wiley & Sons, 2008, ch. 1, pp. 1–24.

[16] S. Adee, “Spy vs. spy,” *IEEE Spectrum magazine*, August 2008, http://spectrum.ieee.org/computing/-software/spy-vs-spy/1.

[17] N. Shachtman, *FBI: Spies Hid Secret Messages on Public Websites*, ser. Wired, jun 2010, http://-www.wired.com/dangerroom/2010/06/alleged-spies-hid-secret-messages-on-public-websites/.

[18] Jennifer Gumban, “Sunsets and cats can be hazardous to your online bank account”, TrendLabs, March 2014, http://blog.trendmicro.com/trendlabs-security-intelligence/sunsets-and-cats-can-be-hazardous-to-your-online-bank-account/

[19] A. De Sélincourt, *Herodotus: The histories*. Penguin Books, 1954.

[20] S. Singh, *The code book: the secret history of codes and codebreaking*. Fourth Estate, 2000.

[21] D. Smith, “Number Games and Number Rhymes: The Great Number Game of Dice,” *The Teachers College Record*, vol. 13, no. 5, pp. 39–53, 1912.

[22] B. Rudin and R. Tanner, *Making paper: a look into the history of an ancient craft*. Rudins, 1990.

[23] I. Cox, *Digital watermarking and steganography*. Morgan Kaufmann, 2008.

[24] W. White, *The microdot: History and application*. Phillips Publications, 1992.

[25] H. Markey and G. Antheil, “Secret communication system,” Aug. 11 1942, US Patent 2,292,387.

[26] D. Kahn, “The history of steganography,” in *Information Hiding*. Springer, 1996, pp. 1–5.

[27] G. Simmons, “The prisoners‟ problem and the subliminal channel,” in *Advances in Cryptology: Proceedings of Crypto*, vol. 83, 1984, pp. 51–67.

[28] W. Bender, D. Gruhl, N. Morimoto, and A. Lu, “Techniques for data hiding,” *IBM systems journal*, vol. 35, no. 3&4, pp. 313–336, 1996.

[29] N. Johnson and S. Jajodia, “Steganalysis of images created using current steganography software,” in *Information Hiding*. Springer, 1998, pp. 273–289.

[30] J. Fridrich, *Steganography in digital media - Principles, Algorithms, and Applications*. Cambridge University Press, 2010.

[31] Y. Wang and E. Izquierdo, “High-capacity data hiding in MPEG-2 compressed video,” in *Proceeding of the 9th International Workshop on Systems, Signals and Image Processing, Manchester, UK*, 2002, pp. 212–218.

[32] R. Anderson, “Stretching the limits of steganography,” in *Information Hiding*. Springer, 1996, pp. 39–48.

[33] A. Castiglione, A. De Santis, U. Fiore, and F. Palmieri, “An asynchronous covert channel using SPAM,” *Computers & Mathematics with Applications*, 2011.

[34] R. El-Khalil and A. Keromytis, “Hydan: Hiding information in program binaries,” *Information and Communications Security*, pp. 287–291, 2004.

[35] R. Anderson, R. Needham, and A. Shamir, “The steganographic file system,” in *Information Hiding*. Springer, 1998, pp. 73–82.

[36] W. Mazurczyk, K. Szczypiorski - *Evaluation of Steganographic Methods for Oversized IP Packets In: Telecommunication Systems: Modelling, Analysis, Design and Management*, Volume 49, Issue 2 (2012), pp. 207-217.