Università di Perugia
Department of Math and Computer Science

Bachelor Thesis in Computer Science

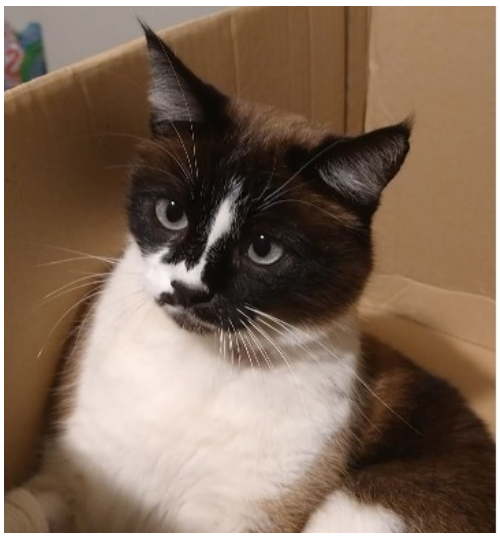
A definition and implementation of a covert channel based on TCP packets

Advisor
Prof. Francesco Santini

Candidate
Andrea Imparato

Academic Year 2021-2022

In loving memory of my beloved cat Yoko. After staying with me for almost ten years, you unfortunately passed away just a few days before this thesis was completed. Since everything I have done so far was made possible by your contribution too, consider this my personal way of saying thank you for all your love. You will be missed.



Chapter 1

Introduction

1.1 From steganography to network steganography

Steganography consists in the science and art of hiding information transfer and storage [23]. Throughout history, many cultures and societies have studied this discipline: it saw use in Ancient Greece, Ancient Rome, the Middle Ages, the Enlightenment Ages and even the two World Wars. Examples of techniques adopted historically include hiding a message on the shaved head of a slave, hiding it in the stomach of hare, carving it on wooden tables covered with wax or encoding it using newspapers, microdots and various other stratagems [4].

Steganography is not to be confused with cryptography: they both share the ultimate goal of protecting information but the former attempts to hide it in order to make it "difficult to notice" while the latter attempts to scramble it in order to make it "difficult to read" [12]. Therefore, cryptography is defeated if someone successfully reads the communication's content while steganography is defeated as soon as someone notices the information exchange itself.

Steganography predates the development of complex machines and exists independently from computer science. The arrival of computers and the internet did however introduce a plethora of new domains where steganography can be applied. Examples of this include hiding information in file systems or in digital media such as JPEG images, MP3 audios and video streams. Generally speaking, a covert transfer of information always features the following elements regardless of its specific domain [23]:

- **Covert Sender**, the entity that sends secret information.
- **Covert Receiver**, the entity that receives secret information.
- **Covert Object**, the data carrier in which the Covert Sender hides secret information. It must be selected so that it does not represent an anomaly but at the same time has enough embedding capacity.
- **Representation**, the specific way in which secret information is embedded in the Covert Object. It must be known to both the Covert Sender and the Covert Receiver.

Among new domains made available by the digital revolution, Network Steganography stands out for its ability to use regular network traffic as a Covert Object to conceal information transfers [23].

Many existing works call this type of communication a **Covert Channel**, referencing a concept first introduced by Lampson in 1973: according to Lampson a covert channel is any channel "not intended for information transfer at all" (as in capable of inadvertently exfiltrating information). The US Department of Defense provided an alternative definition in 1985: a covert channel is "any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy" [21]. This definition emphasizes well both the benefits and the dangers associated with the ever increasing use of network steganography as a tool to circumvent network security: a goal that is ultimately shared both by people with legitimate intentions (such as those who are trying to resist censorship) and individuals with less noble purposes.

In Section 2 we shall describe the context in which covert channels come into play, their most important qualities and some examples of channels proposed by past authors. In Chapter 3 we shall introduce our own covert channel, analyzing the definition of its steganography method, some details of its practical implementation and the results of our field tests. Finally, in Chapter 4 we shall explore some possible countermeasures that can be used to defeat our own channel.

Chapter 2

An overview of the existing context

2.1 Covert channels and their qualities

The prisoners' problem (represented in Figure 2.1 and first formulated by Simmons in 1983 [20]) provides a good abstract example to introduce the context in which covert channels come into play: Alice and Bob are in prison and need to coordinate their efforts in order to escape. They are allowed to exchange written messages but a warden constantly watches their communications to ensure they are not exchanging notes for malicious purposes. In order to hamper their plan the warden could either analyze the messages with the intent of spotting suspicious elements (in which case the warden is a "passive warden") or modify them in ways that prevent the flow of secret information but do not shut down the communication entirely (in which case the warden is an "active warden"). The ultimate goal of Alice and Bob is to communicate successfully without drawing the warden's attention.

Transitioning to the more specific context of network steganography alters some details in this model but not its fundamental premises: in this case Alice and Bob need to communicate over a network instead of a prison, they might as well be the same person (for example someone who installed an application on a certain host that needs to covertly send data to another one) and the warden consists in any hardware or software guarding the network (for example a firewall or an intrusion detection system). Therefore the term "warden" will be henceforth used to reference a wide range of devices and programs, each one potentially capable of adopting different types of countermeasures in order to detect, hamper or prevent secret communications.

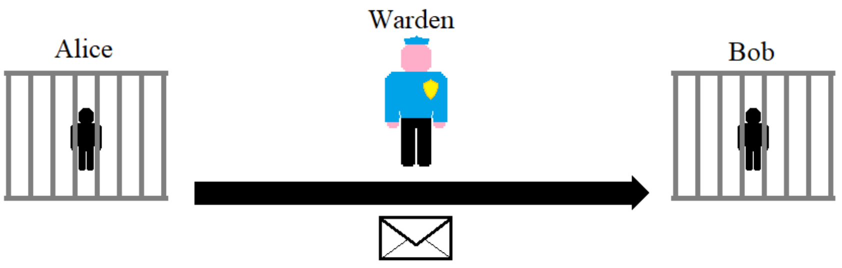


Figure 2.1: Visual representation of the prisoners' problem: Alice is trying to send Bob a note containing a hidden message, but Bob will not receive it unless the message is concealed well enough to elude the warden's inspection.

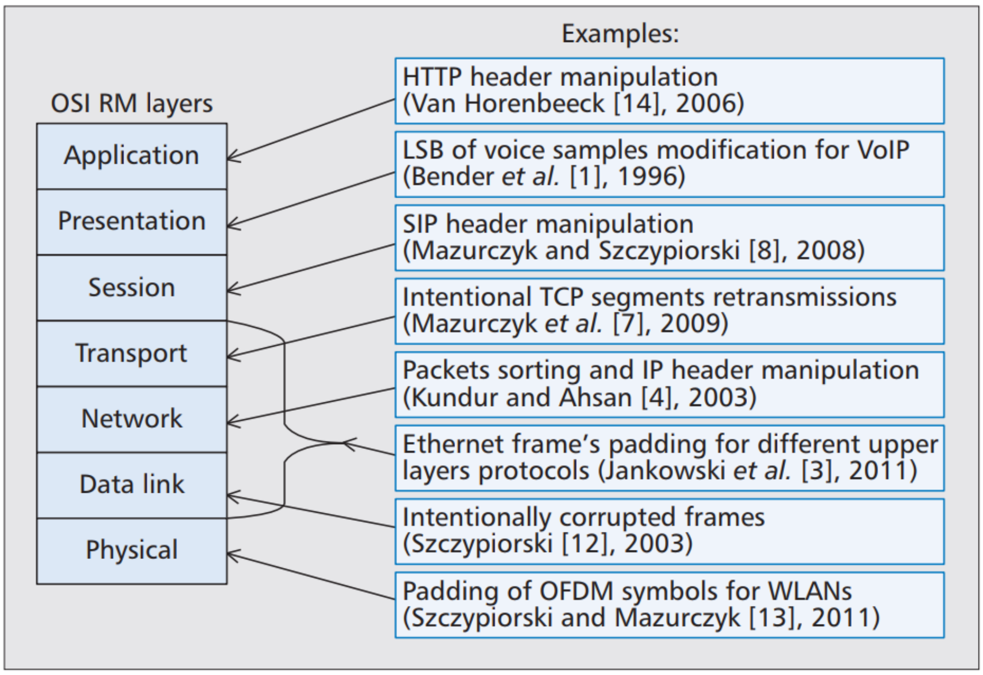

Examples:
OSI RM layers
Application
Presentation
Session
Transport
Network
Data link
Physical
HTTP header manipulation (Van Horenbeeck [14], 2006)
LSB of voice samples modification for VoIP (Bender et al. [1], 1996)
SIP header manipulation (Mazurczyk and Szczypiorski [8], 2008)
Intentional TCP segments retransmissions (Mazurczyk et al. [7], 2009)
Packets sorting and IP header manipulation (Kundur and Ahsan [4], 2003)
Ethernet frame's padding for different upper layers protocols (Jankowski et al. [3], 2011)
Intentionally corrupted frames (Szczypiorski [12], 2003)
Padding of OFDM symbols for WLANs (Szczypiorski and Mazurczyk [13], 2011)

Figure 2.2: Examples of various network steganography methods and their related OSI RM layers [12].

Using regular network traffic to implement a covert channel grants a quite wide margin of freedom in regards to where secret information can be hidden since networks function thanks to a vast set of cooperating protocols. It is common habit to classify these protocols using the OSI reference model shown in Figure 2.2. Covert channels proposed by existing works and their specific implementations do not differ however just in the OSI RM layer at which they operate but also in many other aspects. Unfortunately, a unified approach aimed at classifying the qualities of covert channels is currently missing, meaning each work usually introduces its own framework of reference and ultimately describes these qualities in a slightly different (albeit often similar) fashion. In the attempt to establish which qualities a good covert channel should always have we shall now illustrate the three most recurring concepts found in existing literature, which also happen to be the ones we consider most relevant in light of our own work's scope:

- **Bandwidth** expresses a channel's transmission capacity [12], usually measured either in bits per second (Raw Bit Rate, RBR) or bits per packet (Packet Raw Bit Rate, PRBR) [15]. As we will see in Section 2.2, we shall focus mainly on channels that operate at level 3 and 4 of the OSI reference model, meaning by "packets" we refer to TCP/IP packets specifically.

- **Stealthiness** expresses how successful a channel is at evading the warden's attention. This quality is hard to evaluate objectively because it does not just depend on how sophisticated a given steganography method is: it ultimately also depends on how effective the warden's countermeasures are [12]. In regards to this last point Iglesias et al. argued that while researchers have already conceived many techniques that could be successful at detecting network steganography, several existing security tools have not implemented them just yet [8].

- **Robustness** expresses how much alteration a channel can withstand without compromising the flow of secret information [12]. Much like stealthiness, this quality is also hard to evaluate objectively because of the warden's potential impact. Speaking very generally, the more robust a channel is the less frequently errors and information loss shall occur. For example, in the context of regular network traffic a TCP based connection is more robust than an UDP based data transfer because of the error detection and retransmission capabilities offered by TCP.

While ideally a good covert channel should strive to maximize its performance in regards to all three of these qualities in reality they are interdependent: for example, the increase of bandwidth usually comes at the cost of reducing robustness and stealthiness [12].

On a final note, many works also tend to classify covert channels in two very broad categories (sometimes extended to three) depending on the general type of strategy they adopt to communicate. Specifically [4]:

- **Storage Channels** are channels in which the Covert Sender writes information on specific fields (such as unused fields in a packet header) and the Covert Receiver reads it.
- **Timing Channels** are channels in which the Covert Sender encodes information using the timing of specific events (for example by increasing or decreasing the rate of packets sent over a certain timespan).
- **Hybrid Channels** are channels in which the Covert Sender uses a combination of both.

On average, storage channels tend to have higher embedding capacity compared to timing and hybrid channels but also less robustness and stealthiness. Channels based on timing on the other hand require synchronization on part of the Covert Receiver and are usually also harder to implement [15].

2.2 Related work

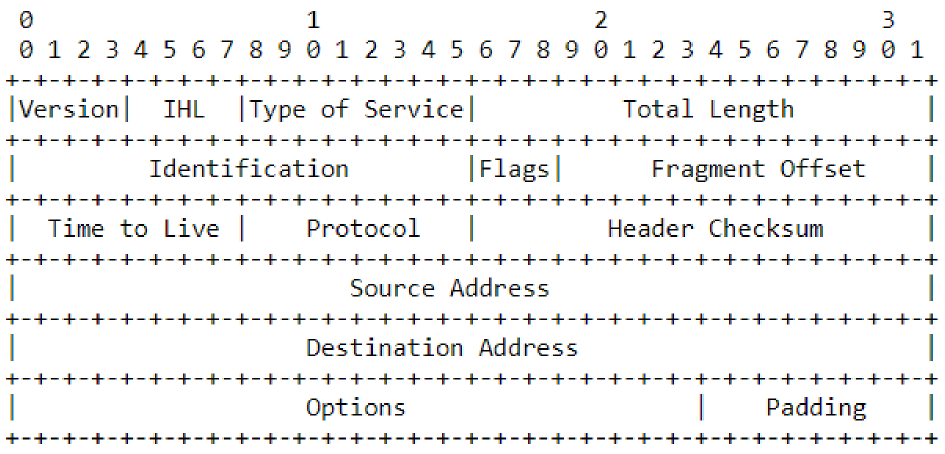
Before introducing our own implementation we want to provide a brief overview of some solutions proposed by past authors. For the purpose of this work we shall limit our focus mainly to channels that operate at the Transport and Network layers of the OSI reference model, usually by manipulating the header fields of TCP/IP packets. Working with the Transport and Network layers offers several advantages compared to the ones above and below. Namely:

- Working with layers below (such as Data link) would limit communication range to the local network since IP routers (and more generally any device that connects networks at a higher layer) never forward any information encoded in them. Moreover, embedding information in the lower layers often requires tampering with the hardware at low level, which might prove very hard to do [16].

- Working with layers above (such as Application) would reduce versatility, in that using a protocol such as FTP to embed a secret message implies both the sending and receiving hosts must run applications that rely on said protocol, which is not necessarily guaranteed. Protocols belonging to the Transport and Network layers do not pose this problem because they are required for internet connections to work [16].

- The widespread adoption of protocols such as TCP and IP also makes a covert channel based on them more "plausible", in that any host using them does not inherently look suspicious to the warden. Likewise, these protocols are also "indispensable", in that the warden cannot simply forbid their use entirely in order to prevent a secret communication [5].

Both IP and TCP work by wrapping user data in a header that contains all information necessary for the protocol to operate [17] [3]. As shown in Figure 2.3 for IP and in Figure 2.4 for TCP these headers include many fields of different length, most of the time fixed.

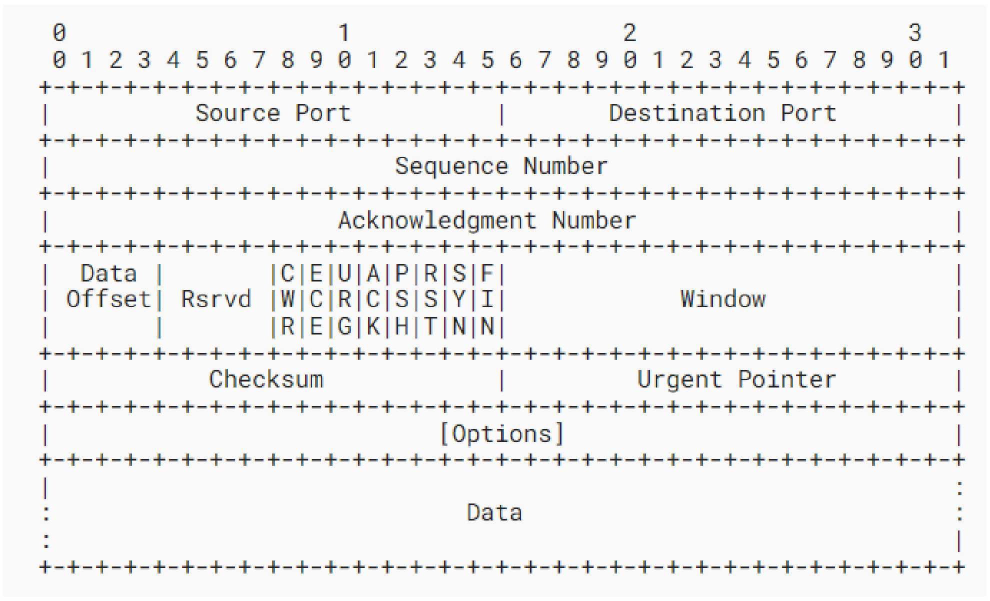


```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |Version|  IHL  |Type of Service|          Total Length         |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |         Identification        |Flags|      Fragment Offset    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Time to Live |    Protocol   |         Header Checksum       |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                       Source Address                          |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Destination Address                        |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                    Options                    |    Padding    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.3: Structure of the IP protocol's header. Each tick mark represents one bit position [17].

Some of these fields cannot be manipulated without entirely disrupting the communication: for example, writing an arbitrary value in the "destination address" field of the IP header would cause the packet to be delivered to the wrong host (and possibly the wrong network altogether).

Other fields however are either entirely redundant or can be modified in ways that do not interfere with the protocols' correct functioning. As such they have historically been the basis for the development of many network steganography techniques, some of which we shall illustrate in the next sections (check also Table 2.1 and Table 2.2 at the end of this chapter for an abridged summary).



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Rsrvd |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           [Options]                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                             Data                              :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.4: Structure of the TCP protocol's header. Each tick mark represents one bit position [3].

We shall also briefly mention some common countermeasures available to the warden: while many of them are tailored to defeat specific steganography methods, some consist in safety measures designed for the general case that should always be taken into account when implementing new covert channels. For example, any given field's redundancy can also play to the warden's advantage, in that should it decide to clear the field or reset it to a specific value there would be no negative consequences on regular network traffic whatsoever. Because of this, many network security devices and softwares actually opt to do so preemptively as a form of precaution. Another very frequent protection measure consists in the warden inspecting all packets in order to either "reject" or "drop" invalid ones. In both cases the packet is not forwarded and shall not arrive at its destination address: the “reject” policy will however also notify the original sender with an error message, while the “drop” policy will maintain complete silence. Most of the time the “drop” policy is considered the safest and as such is often the default one adopted by firewalls [10].

2.2.1 Steganography in the IP header’s Identification field

Among existing works, several proposed embedding a secret message in the Identification field of the IP header. This 16 bits field contains a value initialized during packet creation that is unique for each packet and allows correct identification of fragments belonging to a same packet whenever fragmentation occurs. Since the field can theoretically be set to any value, Rowland suggested a Covert Sender could use it to hide a single ASCII character by embedding it in the 8 most significant bits, assigning a random value to the rest in order to create less suspicious value distributions. The Covert Reader would then divide the field’s content by 256 and interpret the final result as an ASCII value, ultimately extracting a hidden character from each arriving packet for a total PRBR of 8 bits [18].

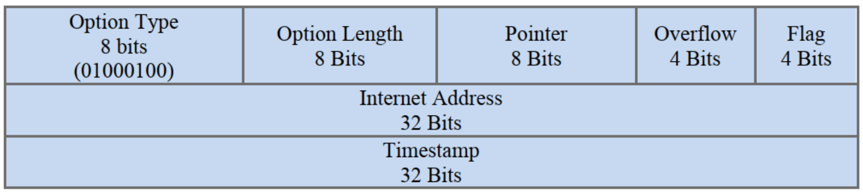
Murdoch and Lewis however argued that values normally assigned to the Identification field are actually not as random as they may seem. The only constraint contemplated by operating systems upon assigning new values consists in keeping them unique over the timespan in which a packet could realistically remain in the network (as well as unpredictable for security reasons). In doing so, different operating systems do not generally follow the same strategy but do often exhibit predictable patterns. These patterns could theoretically be used by the warden to draw a comparison with any traffic considered suspicious, ultimately detecting the field alteration produced by this steganography method [16].

In light of this, Tommasi et al. suggested a different strategy in which the Covert Sender does not directly manipulate the Identification field of a single packet but rather sends a continous stream of packets to exploit the progressive increase of the field’s values naturally operated by the OS. The Covert Sender has to wait for this progression to reach a number that correctly represents secret information it wants to communicate and then mark the corresponding packet by setting its MF bit to 1: MF stands for “More Fragments” and is a bit normally used to signal a given packet has been fragmented and its other parts will soon follow, thus letting the receiver know they have to be reassembled together. In this case however no additional fragments will follow and the Covert Receiver has to interpret the absence of fragments despite MF being set to 1 as indicating the packet's Identification field actually contains secret information. This information consists in a single character belonging to a previously arranged alphabet and in order to extract it the Covert Receiver has to use the modulo operator with the alphabet's size: in case of a letter belonging to the English alphabet for example the Identification field's value modulo 26 yields the result (0 for A, 1 for B, 2 for C, etc.). Since the MF bit is regularly used whenever fragmentation occurs the warden is not expected to modify its value. The abnormally frequent use of the MF bit as well as the absence of following fragments could however alert it and cause it to block suspicious packets. The authors thus introduced an additional safety measure consisting in the Covert Receiver having to interpret any missing number in the Identification field's value progression as belonging to a packet that originally carried secret information but was later dropped by the warden. Since despite the packet drop the receiver is still able to deduce its embedded information the warden's attempt to suppress the covert communication ultimately fails. Unfortunately, this countermeasure also introduces a lot of noise in the channel, in that any packet of the original sequence that ends up being dropped for causes unrelated to the warden also creates a gap in the sequence that is misinterpreted by the Covert Receiver as secret information: an example could be a packet that was dropped simply because its TTL had reached a value of zero. The bandwidth of this solution is ultimately difficult to estimate because it depends on how often the Identification field's value progression produces packets fit for carrying secret information [21].

Kartik Umesh Sharma proposed instead to encode covert information in the payload of the IP packet while using the Identification field as a pointer to the secret data. The bandwidth of this solution is up to 4 bytes per packet. In order to read the embedded information the Covert Receiver has to interpret the 16 bits of the Identification field as a sequence of four nibbles A, B, C and D each consisting of 4 bits. If both B and C are prime numbers the packet contains a secret message consisting of four ASCII characters, otherwise it is just a regular packet. The four bytes of the secret message must be extracted from the packet's payload and their exact starting locations correspond to the values of A, B, C and D respectively [19].

2.2.2 Steganography in the IP header's Options field

Other works suggested the possibility of embedding a secret message in the Options field of the IP header. As the name itself suggests this field is normally used to include additional information useful in certain situations but generally not required for the protocol to operate. The main advantage associated with using this field to implement a covert channel is high bandwidth potential, as its length is not fixed and can occupy up to 320 bits. The main disadvantage however is that since IP Options are generally used very rarely the risk of being spotted by the warden increases dramatically [16]. Moreover, since altering the field's content does not affect the protocol's correct functioning an active warden could simply opt to clear the field altogether in order to prevent its possible misuse.



Option Type 8 bits (01000100)	Option Length 8 Bits	Pointer 8 Bits	Overflow 4 Bits	Flag 4 Bits
Internet Address 32 Bits				
Timestamp 32 Bits				

Figure 2.5: Structure of the IP header's Timestamp option [1].

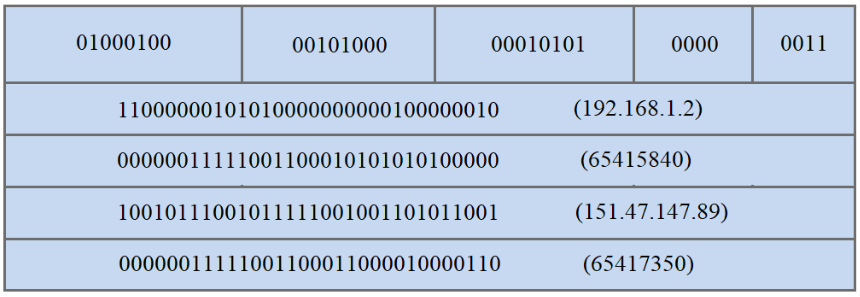
Bedi and Dua implemented a covert channel using the IP Timestamp Option [1]. This option instructs nodes along the path to append individual timestamps to the transiting packet. As shown in Figure 2.5, this option always begins with a header consisting of the following fields:

- Option type (8 bits), to declare which specific option is being used (in this case Timestamp).
- Option length (8 bits), to specify how much space the option occupies as a whole.
- Pointer (8 bits), to keep track of the next available location for a new entry. If its value ever surpasses the value of Option Length it means no space is left.

- Overflow (4 bits), to keep track of how many nodes were unable to add an entry due to lack of available space.

- Flag (4 bits), to specify whether nodes need to add just their timestamp, both their timestamp and their IP address or just their timestamp but only if their IP address matches one of the predefined values provided by the packet sender. These three policies correspond to values 0, 1 and 3 respectively.

In normal circumstances the basic premise behind how the IP Timestamp option works is that whenever a node adds a new entry it must also advance the Pointer field to reflect how much available space is left (see Figure 2.6 for an example). If any node fails to add a new entry because no space is left it must instead increase by one the Overflow counter. This field is normally initialized with a value of 0 and is exactly where the authors propose to embed the secret message, resulting in a bandwidth of 4 bits per packet. The authors conducted their experiment on a local network, thus ensuring no node ever writes any timestamp. In the internet however there is the risk of a real overflow overwriting embedded data. Moreover, according to Murdoch and Lewis any packet that uses the IP Timestamp option can travel at most 20 hops regardless, potentially limiting the covert channel's range [16]. Finally, the warden could easily detect this channel by noticing the Overflow field has a value greater than 0 despite the Pointer and Option length fields indicating there is still space available.



01000100	00101000	00010101	0000	0011
11000000101010000000000100000010	(192.168.1.2)			
00000011111001100010101010100000	(65415840)			
10010111001011111001001101011001	(151.47.147.89)			
00000011111001100011000010000110	(65417350)			

Figure 2.6: Example of IP Timestamp option with Flag field set to 3 and two entries added (each consisting of IP address followed by timestamp value).

Trabelsi and Jawhar opted instead to implement a covert channel using the Traceroute option [22]. This option instructs nodes along the path to append their IP address to the transiting packet. The secondary header adopted by this option is shown in Figure 2.7 and is relatively similar to the one used by the Timestamp option, in that the first three fields are used in the exact same way. The most notable difference is the absence of the Overflow and Flag fields due to the fact this option contemplates just one single policy: each node has to either add its IP address or do nothing at all, depending on whether there is still space or not.

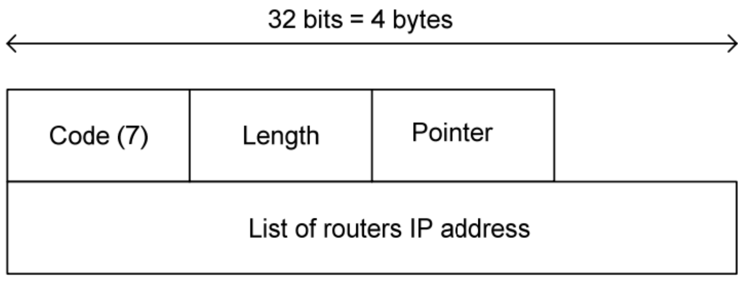
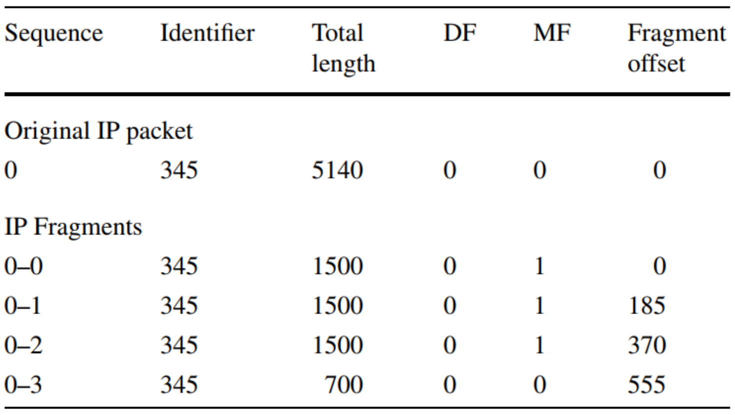


Figure 2.7: Structure of the IP header's Traceroute option [22].

The authors argue that if the Covert Sender manipulates the Pointer field by slightly increasing its value the resulting offset will cause any node along the path to write only from that location onwards, effectively creating an area of reserved space that can be used to embed a secret message. If the Pointer field is instead initialized with a value greater than the Option length field, no node will be able to add any entry at all and the reserved area will effectively occupy all available space, resulting in a total bandwidth of 36 bytes per packet. Moreover, since the 8 bits left free by the absence of the Overflow and Flag fields are not enough for an IP address to fit they simply become redundant and can be used to expand the covert channel's bandwidth even more. Since the authors implemented a two-way communication they chose however to reserve these bits for traffic control functionalities very similar to the ones used in TCP in order to improve robustness. Compared to using the Timestamp option this solution is more robust and offers more bandwidth but is still prone to draw the warden's attention quite easily. Specifically, the warden could realize either that some IP addresses make no sense (for example if they match any value reserved for private networks) or that their number does not reflect the number of hops realistically traveled by the packet (expecially if the warden is very close to the Covert Sender).

2.2.3 Steganography in the fragmentation process

Fragmentation is a process that occurs whenever an IP packet encounters a segment along the path that lacks a Maxmimum Transmission Unit (MTU) big enough to carry the packet as a whole: as a consequence, the packet is split in smaller parts (called Fragments) that are individually sent along the segment and later reassembled by the receiver. Figure 2.8 shows an example of how an IP packet is fragmented. Every fragment of the sequence except the last one signals the presence of other incoming fragments by keeping its MF (More Fragments) bit set to 1. Since fragments are not guaranteed to arrive in the correct order, the Fragment offset field is used to determine which part of the original packet is carried by each fragment. All fragments belonging to a given packet share its Identification field value so they can be easily distinguished from fragments belonging to other packets.



Sequence	Identifier	Total length	DF	MF	Fragment offset
Original IP packet					
0	345	5140	0	0	0
IP Fragments					
0–0	345	1500	0	1	0
0–1	345	1500	0	1	185
0–2	345	1500	0	1	370
0–3	345	700	0	0	555

Figure 2.8: Example of IP Fragmentation [14].

Mazurczyk and Szczypiorski proposed various steganography methods based on forging fragment sequences that at first glance look like the mere product of a natural fragmentation process but actually carry hidden data [14]:

- The first method consists simply in conveying information through the total number of fragments created: an even number represents a 0 and odd number represents a 1, for a total bandwidth of 1 bit per packet.
- The second method consists in embedding information in the Fragment offset of each fragment: once again even values represent 0 and an odd values represent 1. This method's bandwidth ultimately depends on how many fragments are created but since as shown in Figure 2.8 fragments other than the last one usually have the same size the authors also suggested a variant where only the final fragment's offset is modulated: in this case the bandwidth drops to 1 bit per packet but the covert channel becomes harder to spot.
- The third method consists in injecting in an authentic sequence of fragments some forged ones containing secret information in their payload instead of parts of the original packet. In order to allow the Covert Receiver to distinguish these forged fragments their payload must include a label that matches a specific hash value. This hash value depends on the packet's Identification field, the fragment's offset and a secret key known only to the Covert Sender and Receiver. The total bandwidth of this method depends on the total number of fragments as well as their payload size.
- Other methods rely on timing rather than storage as a transmission strategy. Examples include altering the fragments' transmission rate to represent 0 (less frequent) or 1 (more frequent) or conveying information through their permutations.
- One final method hides information in fragmentation irregularities. Since as shown in Figure 2.8 Fragment offset values belonging to a same sequence usually follow a regular increase, any gap in their value progression implicitly tells the Covert Receiver a fragment is missing. The Covert Sender can ultimately choose to alter a sequence by deleting a fragment to represent a 1 or to leave it intact to represent a 0, resulting in a total bandwidth of 1 bit per packet.

In order to defeat these steganography methods a passive warden can resort to statistical analysis aimed at detecting anomalies in the fragmentation process. An even better solution consists in deploying an active warden that reassembles the fragments and randomly splits them again in order to jam any covert information. According to Klein, reassembling all fragments belonging to a given packet before forwarding it might also be the default policy already followed by certain *CGNATs (Carrier Grade NATs)*, potentially limiting the viability of these methods [10].

2.2.4 Steganography in other fields of the IP header

IP packets can also be labeled to prevent their fragmentation altogether, specifically by setting their DF (Don't Fragment) bit to 1. This aspect was exploited by Kundur and Ahsan to develop yet another covert channel [11]. In this case the embedded information consists exactly in the value chosen by the Covert Sender for the DF bit. Naturally this also implies the Covert Sender has to create packets small enough to fit the MTU of all segments along the path, lest the impossibility of fragmenting any packet that has DF set to 1 making it impossible to deliver. Such a channel is overall very easy to implement and grants a total bandwidth of 1 bit per packet. Murdoch and Lewis argued however that a warden could spot it with relative ease since the normal state of the DF bit can be predicted from the packet's context [16].

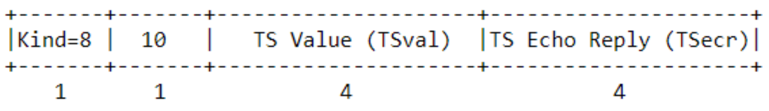
Another steganography method analogous to embedding information in the DF bit is to instead embed it in the Delay bit as suggested by Hintz [7]. The Delay bit belongs more generally to the ToS (Type of Service) field, which was originally intended to carry additional information related to quality of service parameters but is nowadays almost never used with its original semantic. While in light of this the Covert Sender could theoretically use all 8 bits of the field to embed secret information, doing so would in practice cause it to alert the warden since this field is set to zero by default in almost all operating systems [16]. The author himself thus recommends using only the fourth bit (Delay) to slightly improve stealthiness, effectively limiting the bandwidth to 1 bit per packet.

Zander et al. explored the possibility of using the IP header's TTL field to carry secret information [24]. This field is of fundamental importance in regular network traffic because it prevents network congestion caused by packets trapped in infinite routing loops. The original sender of any packet must thus always initialize it with a certain value and every node that receives and retransmits the packet decreases it by one: if the value ever drops to zero the packet is simply deleted without retransmission, ultimately ensuring it will only travel for a limited number of hops. Naturally this means the field must be always initialized with a value great enough to ensure the packet is not accidentally dropped just because it was routed on a path consisting of many hops. Implementing a covert channel using the TTL field is not an easy task because its value is purposefully designed to be altered upon each hop, ultimately resulting in a very high risk of information loss. Since the field can be initialized with any value between 0 and 255 and the number of hops between two hosts of the Internet is generally assumed to be smaller than 32, the best way to avoid this issue would be to encode secret information using two distinct values with a difference between them much greater than 32: this way the Covert Receiver would interpret a high value as representing a 1 and a low value as representing a 0 without being significantly affected by alterations that naturally occur along the path, for a total bandwidth of 1 bit per packet. Doing so however could draw the warden's attention because most operating systems stick to one fixed and well known value when initializing the TTL field. The authors thus analyzed existing network traffic in order to discover what type of changes to TTL values could effectively blend in well enough to avoid detection. Their analysis ultimately suggested a good covert channel should use at most two different TTL values immediatly adjacent to each other and avoid changing values more often than once every 2-3 packet pairs at most. An effective strategy to defeat such a channel would be to deploy an active warden that simply normalizes the TTL value of all transiting packets [15].

2.2.5 Steganography in the TCP header

Similarly to the IP header, the TCP header also includes several fields that can be used to embed secret information. A good example is the Urgent pointer field, which was designed to be used together with the URG flag to signal the presence of urgent data in the payload [3]. Since nowadays this field is almost never used, Hintz suggested the possibility of using it to develop a covert channel [7]. Such a channel would be easy to implement and grant a total bandwidth of 16 bits per packet, but Hintz himself recognized it would also be easy to detect (since any value other than zero is inherently suspicious) and easy to prevent (since an active warden could simply clear the field's value with no consequences).

Giffin et al. developed instead a solution based on using the Timestamp option of the TCP header, not to be confused with the homonym option of the IP header [5]. The Options field of the TCP header is generally similar to the one already seen in the IP header, in that it once again contains information that can be occasionally useful but is generally not necessary for the protocol to operate. In TCP the Timestamp option is used to measure the time gap between a packet's departure from a sending host and its arrival at the receiving one [9]. In order to do so the sending host appends its local clock's timestamp in the 32 bit TS value field shown in Figure 2.9. The receiving host then copies this information in the acknowledgement packet used to confirm the original one's arrival and adds its own timestamp in the TS echo reply field before sending it.



```
+-------+-------+---------------------+---------------------+
|Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
+-------+-------+---------------------+---------------------+
    1       1              4                     4
```

Figure 2.9: Structure of the TCP timestamp option. The length of each field is expressed in bytes [9].

The authors suggested modulating the least significant bit of the sender's timestamp to convey secret information, for a total bandwidth of 1 bit per packet. Since timestamp values must be strictly monotonic [16] this modulation must be always done through an increment: the authors thus recommended also delaying the packet's departure just enough for the new timestamp value to look authentic. While an active warden could theoretically clear the option's fields altogether, the authors argued that doing so in every active TCP connection could prove quite expensive. Moreover, including the Timestamp option in many TCP packets is actually the norm in some Linux versions and other Unix-like operating systems [16], meaning its use would inherently look suspicious only if the sender is running an operating system that follows a different policy. Nevertheless, the timestamp alteration produced by this steganography method can still be detected by either measuring the ratio of total to different timestamps or applying a complex randomness test to the least significant bits, depending on connection speed [7].

Several works proposed implementing a covert channel using TCP's ISN field. ISN stands for Initial Sequence Number and is a vital field used during TCP connections to both keep track of how many bytes the sending host has progressively transmitted and distinguish different TCP connections that use the same socket [16]. The field's initial value is decided by the sending host and can theoretically be any number ranging from 0 to $2^{32} - 1$. The main advantage of any steganography method that modulates this field consists in the warden not being able to simply prohibit its use or arbitrarily alter its value without affecting regular network traffic as well. An active warden's best shot at neutralizing such a covert channel would be to proxy all outgoing TCP connections so that ISN values are always overwritten but Hintz argued doing so would be difficult, possibly hinting at the resulting overhead negatively impacting network performance [7].

Rowland suggested the easiest way to hide secret information in the ISN field would be to use embedding techniques similar to the one already discussed for the IP header's Identification field, consisting in reserving some bits (usually the most significant ones) to encode an ASCII character and randomizing the rest to create more realistic value distributions. The resulting bandwidth would be 8 bits per TCP connection [18].

Rowland also suggested a very interesting evolution of this strategy where secret information is still encoded in the ISN field but is transported through a TCP handshake rather than a direct packet transmission. A TCP handshake occurs whenever two hosts want to establish a new TCP connection and is what allows them to synchronize each other on the counterpart's initial sequence number [3]. The handshake begins with a calling host transmitting a packet containing the ISN field set to a chosen value and the SYN flag set to 1. In order to continue the handshake the recipient replies with a packet containing its own initial sequence number, both the SYN and ACK flags set to 1 and the Acknowledgement number field set to match the caller's ISN value increased by one. Finally, the caller completes the handshake by replying with a third packet containing the Acknowledgement number field set to match the recipient's ISN value increased by one and the ACK flag set to 1. Rowland's ultimate idea consists in forging a packet so that:

- The ISN field in the TCP header contains secret information.
- The SYN flag in the TCP header is set to 1.
- The IP header's Source address field is altered to actually reflect the Covert Receiver's IP address.
- The IP header's Destination address field is set to match the address of any legitimate server known to be reachable from the Covert Sender's network.

This strategy relies on the legitimate server interpreting the incoming packet as an attempt to establish a new TCP connection on the Covert Receiver's part. It will thus answer with a packet directed to the Covert Receiver actually containing the Covert Sender's ISN increased by one in the Acknowledgement number field. The Covert Receiver can thus read this packet's Acknowledgement number value, decrease it by one and ultimately extract the secret information originally embedded by the Covert Sender [18]. The most relevant aspect of this strategy is that having a legitimate server "bounce" secret information lets the Covert Sender bypass a typical limitation consisting in the warden allowing outgoing connections only towards certain specific destination addresses. The warden could however still be alerted by the presence of a packet whose source address does not match any of the network's hosts: in this sense it is nowadays not uncommon for firewalls to apply *SAV (Source Address Validation)* as a default policy [10].

Ganivev et al. created their own variant of Rowland's original strategy using both the TCP header's ISN field and the IP header's Identification field at the same time. Their method consists in having secret information undergo an encryption process before being embedded in the ISN field while using the Identification field to carry the decryption key necessary for the Covert Receiver to read it [4].

Upon considering the alleged randomness of ISN values Murdoch and Lewis made an argument similar to the one already presented for the IP header's Identification field, in that ISNs need to respect certain constraints (one of which consists in being unpredictable for security reasons) but are ultimately not determined at random. Since strategies adopted by several operating systems to compute new ISNs are well known the Covert Sender cannot simply assign an arbitrary value to the field without potentially alerting the warden. The authors thus developed an alternative embedding method called "Lathra" that takes into account the ISN generation policies contemplated in operating systems such as Linux and OpenBSD in order to produce field values that look as realistic as possible while still carrying hidden information [16].

Table 2.1: Summary of IP based steganography methods

Method	**Bandwidth**	**Advantages**	**Disadvantages**
IP identification modulation (1997) [18]	8 bits per packet	• Easy to implement • Warden cannot clear the field	• Warden can detect the channel through statistical analysis
DF bit modulation (2003) [11]	1 bit per packet	• Easy to implement	• Warden can notice the field's alteration by looking at context
Delay bit modulation (2003) [7]	1 bit per packet	• Easy to implement	• Warden can clear the field • Warden can detect the channel through statistical analysis
TTL modulation (2007) [24]	1 bit per packet	• Based on a vital field • Can be hard to spot (depending on implementation)	• Warden can reset the field to a default value • Packet traveling alters the field (risk of information loss)
IP traceroute option modulation (2010) [22]	Up to 296 bits per packet	• Offers very high bandwidth • Part of the bandwidth can be used to improve robustness	• Warden can clear the field • Warden can notice the field's inconsistency

Table 2.1: Summary of IP based steganography methods

Method	**Bandwidth**	**Advantages**	**Disadvantages**
Fragmentation based methods (2012) [14]	Changes with each specific method (often 1 bit per packet)	No specific advantage	• Fragmentation is rarely used • Reassembling the fragments jams the information (can be done by the warden or by CGNATs)
Using IP identification as pointer (2016) [19]	Up to 32 bits per packet	• Easy to implement • Does not alter header fields	• Alters the payload
IP timestamp option modulation (2020) [1]	4 bits per packet	No specific advantage	• Limited range • Warden can clear the field • Warden can notice the field's inconsistency
COTIIP (2022) [21]	Difficult to estimate (affected by alphabeth size and message structure)	• Can resist packet filtering	• Receiver misinterprets normal packet drops as relevant information • Sender is forced to transmit continously

Table 2.2: Summary of TCP based steganography methods

Method	**Bandwidth**	**Advantages**	**Disadvantages**
ISN modulation (1997) [18]	8 bits per TCP connection	• Easy to implement • Warden cannot clear the field	• Warden can detect the channel through statistical analysis • ISN field is overwritten if the TCP connection is proxied
TCP handshake bounce (1997) [18]	8 bits per TCP connection (message embedded in ISN field)	• Easy to implement • Warden cannot clear the field • Works in networks that restrict destination IP addresses	• Can be neutralized with SAV (Source Address Validation) • Warden can spot ISN alterations through statistical analysis • ISN field is overwritten if the TCP connection is proxied
TCP timestamp option modulation (2002) [5]	1 bit per packet	• Based on a field that is used quite often	• Warden can clear the field • Warden can detect the channel through statistical analysis
URG pointer modulation (2003) [7]	16 bits per packet	• Easy to implement • Good bandwidth	• Use of this field is suspicious (almost never used in normal circumstances) • Warden can clear the field
OS sensitive ISN modulation (2005) [16]	Varies on each OS	• Warden cannot clear the field • Modulation is very stealth	• Implementation is complex and OS dependent • ISN field is overwritten if the TCP connection is proxied

Table 2.2: Summary of TCP based steganography methods

Method	Bandwidth	Advantages	Disadvantages
Encryption in ISN with decription key in IP identification (2021) [4]	8 bits per TCP connection	• Warden cannot clear the fields • Modulated values have a better distribution compared to simple ISN modulation	• Warden can possibly spot ISN anomalies through statistical analysis • IP identification values can look suspicious • ISN field is overwritten if the TCP connection is proxied

2.2.6 A word on timing channels

At the end of section 2.1 we mentioned how certain covert channels convey secret information by altering the timing of specific events rather than by hiding it within outgoing packets. The channel based on TCP's Timestamp option [5] illustrated in the last section can be considered to an extent a channel of this kind: while information is technically still embedded in a header field, the channel also modulates its transmission timings in order to make the forged timestamp values look authentic. We also mentioned how certain fragmentation related steganography methods are based on timing, for example altering the fragments' transmission rate to represent 0 (less frequent) or 1 (more frequent), or using fragment permutations to represent information [14].

Many more valid examples are available in existing literature. One of the most simple consists in the channel proposed by Cabuk et al. [2], where information is conveyed through either the transmission (1) or the absence of transmission (0) of a single IP packet within a specific time frame. A somewhat similar but more articulate strategy was suggested by Luo et al. [13] and consists in using well distanced TCP packet bursts, each consisting of a different number of packets aimed at reflecting a specific secret symbol. These packets always share the same size: as long as the first and last packet arrive correctly the Receiver can thus always figure their total number just by looking at the gap between the first and last sequence numbers (ultimately ensuring secret information is received even if some packets in the middle are dropped).

Overall, timing based channels represent a valid alternative to storage based ones but usually also entail additional drawbacks such as synchronization issues, noisiness, major complexity and reduced bandwidth [15]. Because of this, upon implementing our own covert channel we decided to prioritize a storage based steganography method. In the next chapter we shall illustrate the specific steps followed during its creation process.

Chapter 3

Our implementation

In the last chapters we introduced the concept of network steganography and we also listed several existing methods proposed in the past by different authors that allow its implementation in various forms. In this chapter we will instead focus on our very own approach to the matter of creating a covert channel, starting from the fundamental aspect of choosing the header field most fit to carry a secret message. Before presenting the channel itself, we must however first state a few vital premises in order to define as clearly as possible the context in which we imagine to operate, since a given covert channel does generally not work in any given situation.

3.1 Establishing a context

The basic scenario we are picturing revolves around a host belonging to a network that is property of a small to medium sized private company. In this situation, we play the role of the attacker and as such are interested in stealing data from this specific host. We can assume we already succeeded in preemptively violating it and gaining full control of it, but we now need a way to exfiltrate any useful information we might have found. We ultimately plan to have the compromised host send information to a receiving end that is located on another network through TCP/IP packets, using network steganography to conceal the information transfer. The key takeaways that are most relevant for our work are the following:

- Some form of software or hardware based protection is guarding the network (i.e. if we are playing the role of the attacker a warden is playing against us as the defender).

- No human will generally ever check network traffic unless alerted by the warden.
- Only certain services are allowed on the company's network, meaning the compromised host cannot use any given destination port. We can however assume internet navigation is authorized, meaning the use of port 80 and port 443 is allowed.
- There are no preemptive restrictions in regards to which IP addresses the compromised host is allowed to contact. IP blacklisting (in both directions) is only applied following the warden's alerts.
- The receiver's IP address is completely unknown and no host on the network would normally ever try to make contact with it. Traffic associated with our information exfiltration activity will effectively be the only traffic related to it (in both directions).
- The operating system running on the compromised host is Windows (either version 10 or 11).
- For simplicity, assume the information we plan to exfiltrate consists in 7-bit characters belonging to the original ASCII set.

3.2 Defining a steganography method

Once the context's basic premises are established, defining an appropriate steganography method comes next. In this section we shall thus illustrate all the logical steps we followed in order to achieve this goal.

3.2.1 Choosing the most appropriate field

The first and most important aspect of defining a new storage based steganography method consists in choosing the field in which the secret message will be embedded. The vast list of methods we introduced in the last chapter is a good starting point, but after carefully reflecting on the risks associated with each proposed technique we unfortunately concluded only a few of them can be helpful in our case. The problem associated with most of them is that traffic normalizers and active wardens are the type of countermeasure we fear the most. This is because the type of action they need to operate is generally very easy to implement and has a very low computational cost, usually consisting in nothing more than just resetting or overwriting the targeted field's value. Moreover, this action often does not need to be triggered by specific conditions and can simply be applied as a default policy. For example, Handley et al. suggested a traffic normalizer can simply delete all IP options from packets because their removal hampers at most diagnostic tools and generally does not affect higher layers' semantics at all [6]. We ultimately expect the vast majority of wardens to either clear or reset any secondary header field that is not vital for the basic functioning of the TCP and IP protocols and thus consider too risky hiding the secret message in any field that can be simply overwritten with no negative consequences. This rules out the possibility that we base our method on any of the following:

- TCP timestamp option modulation [5].
- DF bit modulation [11].
- Delay bit modulation [7].
- URG pointer modulation [7].
- TTL modulation [24].
- IP traceroute option modulation [22].
- Fragmentation based methods [14] (while they do not necessarily rely on hiding the message in a field, traffic normalization is still effective at neutralizing them).
- IP timestamp option modulation [1].
- COTIIP [21] (since it relies on the MF bit).

Among remaining methods, three involve altering the IP identification field and as such cannot be used in our case. The reason for this is how Windows (which we have established being the operating system used by our compromised host) handles the IP identification field. An in-depth explanation on this topic was provided by Klein [10], who resorted to reverse engineering to study the field's implementation in Windows 10 (though judging from our experience, we believe the same implementation is still used in Windows 11 as well). In order to initialize the IP identification field, Windows keeps a dedicated counter for each *(Source address, Destination address)* tuple related to outgoing traffic. Each counter ultimately resides in an object called "Path" and each Path is stored in a hash table called "PathSet". The first time a new packet is being sent from a given source address to a given destination address, a new Path object is created and its identification counter is initialized with a random value. Then, for each new packet sent from this exact source address to this exact destination address, the counter is simply incremented by one every time. Path objects are ultimately removed from the PathSet only in extremely rare occasions: when PathSet size exceeds some (pretty high) thresholds, when PathSet growth rate exceeds the limit of 10000 new Path objects per second or when the system is rebooted. Therefore, after choosing any given value for the first packet's Identification field we would then have to always increment said value by one, lest producing forged packets that display a substantial difference compared to legitimate ones. Ultimately, this means we cannot modulate the field as we please and thus cannot base our method on any of the following either:

- IP identification modulation [18].
- Using IP identification as a pointer to embedded data [19].
- Encryption in ISN with decryption key in IP identification [4].

In the last chapter we also established that exploiting the TCP handshake by spoofing the source address (as originally suggested by Rowland [18]) is not a viable alternative anymore due to the majority of firewalls adopting *SAV (Source Address Validation)* as their default policy [10]. This ultimately means only two methods reamain [16] [18], both of which are based on the same concept: **modulating the ISN field**. This is more than just a mere coincidence: after having already ruled out the possibility of embedding the secret message in any secondary header field, a quick look at the remaining, more vital, fields can easily allow us to conclude that the ISN field is indeed the only one that allows alteration with a sufficient degree of flexibility in our situation (ultimately implying we too should base our method on ISN modulation). In fact:

- **Source address (IP)** cannot be altered, as we just established that source address spoofing is quite risky. Moreover, the field's original value would be lost as soon as the packet leaves the private network due to NAT.
- **Destination address (IP)** cannot be altered, lest causing the communication to be delivered to the wrong recipient.
- **Identification(IP)** cannot be altered without risking detection because identification values in Windows follow a well-known and predictable pattern of constant increase.
- **Protocol (IP)** cannot be altered because its value is fixed (since packets we are working with are always TCP packets).
- **TTL (IP)** and **Window (TCP)** cannot be altered because these fields have a fixed value that is determined by the operating system's settings.
- **Source port (TCP)** can theoretically be altered but its original value would be lost as soon as the packet leaves the private network due to NAT.
- **Destination port (TCP)** offers very limited flexibility in terms of how much we can alter it due to the limited range of services allowed on the private network.
- **Acknowledgement (TCP)** cannot be altered because its value is directly dependent on the last packet's sequence number value (or is alternatively set to 0 if the packet is the first one of a new connection).
- **Initial header length (IP)**, **Total length (IP)**, **Data offset (TCP)**, **Checksum (IP)** and **Checksum (TCP)** cannot be altered because their value is directly determined by all other fields' size (in the case of the first three) or value (in the case of the last two).

3.2.2 Assessing the risks associated with ISN modulation

Unfortunately, choosing TCP's initial sequence number as the embedding field does not come without potential negative consequences. The first and most important issue consists in the fact that any middle-box that operates TCP proxying on all connections would automatically rewrite the field and thus neutralize the channel. Murdoch and Lewis argued that such an operation would be computationally expensive and is thus not very likely [16], but since both hardware and computational power have made quite significant progress in the last twenty years we cannot be sure such a claim would still be valid in 2023. If TCP proxying occurs, there is ultimately no countermeasure we can adopt to maintain the channel operational. We can, however, adapt the channel to detect it (and more generally detect any type of alteration to ISN that occurs after the packet's departure) so that transmission can be interrupted accordingly (thus avoiding unnecessary risks).

Another reason for concern is that it might be possible to predict how a given operating system normally assigns new sequence number values, thus allowing the warden to notice the difference between legitimate values and forged ones. This issue was first raised by Murdoch and Lewis [16], but compared to their original context our situation offers two elements that can play to our advantage and ultimately make the risk more acceptable. The first one is that improving the process of sequence number generation with the intent of making it as hard as possible to understand or replicate is in every operating system's best interest, meaning we expect sequence number generation to have become harder to predict compared to twenty years ago. The second one is that our compromised host uses Windows: being it a proprietary software, little to no information on how initial sequence numbers are generated in version 10 or 11 is generally available (implying we do not expect the warden to possess such knowledge either).

Bandwidth is also a tricky aspect to consider: on the one hand the sequence number field is 32 bits long (thus being one of the largest in both headers), but on the other hand its value can be altered unconditionally only if the packet is the first one of a new connection (as its value should otherwise increase in order to reflect how many bytes the current packet is transferring). As a result, the first packet of any given connection is the only one that can carry secret information and any one that follows would effectively be just a waste of bandwidth. In light of this, we concluded that in order to maximize bandwidth the type of behaviour our channel should try to simulate consists in multiple connection attempts that get continously rejected by the receiver (thus allowing the sender to continously refresh its ISN values).

Finally, great care must be taken when handling the embedding process itself, lest producing ISN value distributions that would look quite odd in case of statistical analysis. For example, if the message consisted of English written text and we simply embedded its characters in the sequence number field with no additional processing of sort, the resulting sequence number values would display a distribution that does not look uniform at all. This happens because English language only uses a small fraction of the 128 symbols offered by original ASCII and among those certain ones are used a lot more often than others (in particular the ones representing vowels, since they are the letters used most frequently).

3.2.3 Illustrating our own steganography method

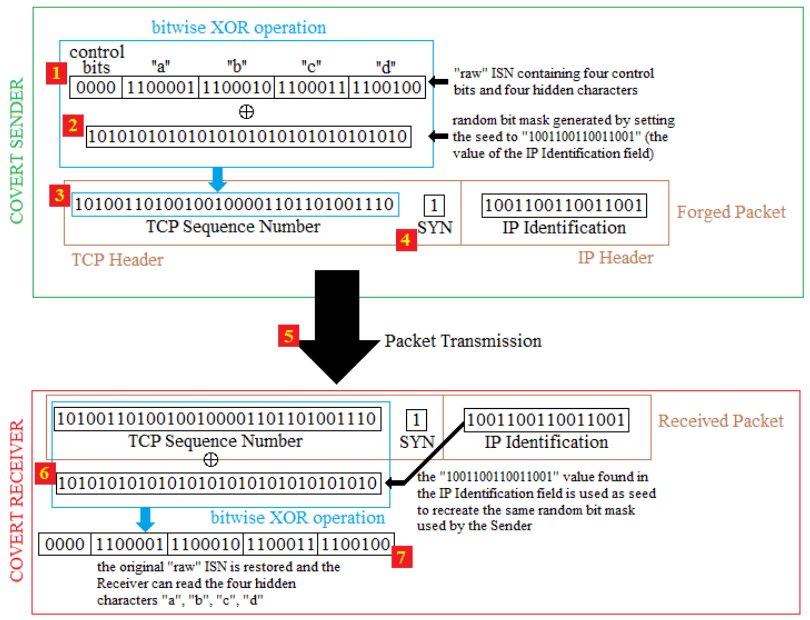
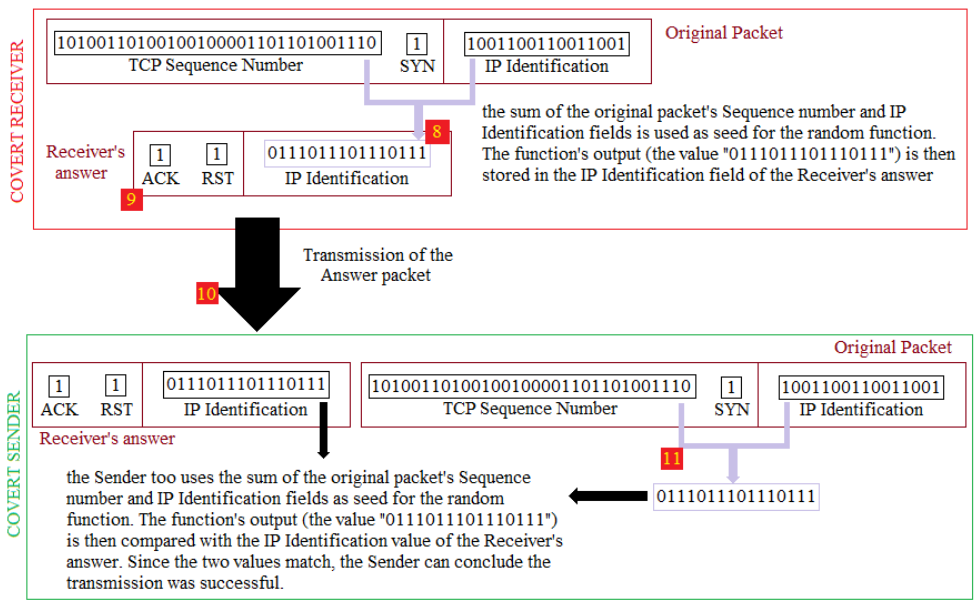


Figure 3.1: Visual representation of how the Covert Sender embeds and transmits the secret message in our steganography method. The enumeration in red matches the steps illustrated in list 3.2.3.



COVERT RECEIVER
10100110100100100001101101001110
TCP Sequence Number
1
SYN
1001100110011001
IP Identification
Original Packet
the sum of the original packet's Sequence number and IP Identification fields is used as seed for the random function. The function's output (the value "0111011101110111") is then stored in the IP Identification field of the Receiver's answer
8
Receiver's answer
9
1
ACK
1
RST
0111011101110111
IP Identification
10
Transmission of the Answer packet
COVERT SENDER
Original Packet
1
ACK
1
RST
0111011101110111
IP Identification
Receiver's answer
10100110100100100001101101001110
TCP Sequence Number
1
SYN
1001100110011001
IP Identification
11
0111011101110111
the Sender too uses the sum of the original packet's Sequence number and IP Identification fields as seed for the random function. The function's output (the value "0111011101110111") is then compared with the IP Identification value of the Receiver's answer. Since the two values match, the Sender can conclude the transmission was successful.

Figure 3.2: Visual representation of how the Covert Receiver creates and transmits an answer in our steganography method. The enumeration in red matches the steps illustrated in list 3.2.3.

The steps followed by our steganography method ultimately take into account all the risks we mentioned in the last section and are illustrated in Figure 3.1 and Figure 3.2. In order:

1. The Covert Sender creates a forged TCP packet and embeds the secret message in the Sequence number feld of the TCP header. Up to four ASCII characters are embedded using the 28 least signifcant bits of the feld (the four most signifcant bits are instead used to implement error codes and control codes aimed at improving robustness, as we will show in section 3.4.1).

2. In order to improve their value distribution, a randomized bit mask is also added to each forged sequence number. This mask is obtained by calling a given random function (known to both the Sender and the Receiver) after its seed has been set to reflect the packet's IP identification value. Since each packet has a unique identification value, the randomized bit mask generated is always different.

3. Said mask is then applied to the "raw" sequence number through a bitwise XOR operation.

4. The Sender then sets to 1 the packet's SYN flag in order to pretend the Sender is attempting a new three-way handshake with the Receiver.

5. The Sender then proceeds to deliver the packet.

6. The Covert Receiver intercepts the incoming packet and extracts the value of its Identification field first. It then uses this value to set the seed of the same random function used by the Sender in order to recreate the same randomized bit mask.

7. The Receiver then extracts the incoming packet's Sequence number and operates a bitwise XOR with the bit mask just recreated. The Receiver's bitwise XOR ultimately undoes the Sender's, thus restoring the original "raw" sequence number and allowing the Receiver to extract the hidden message from it.

8. The Receiver then sets the random function's seed to a value that corresponds to the sum of the incoming packet's IP identification and Sequence number values. The output generated by calling the random function will thus be a randomized "signature" that is directly determined by the two fields' value.

9. Finally, the Receiver assembles an answer consisting in yet another forged TCP packet. The RST and ACK flags of this packet are set to 1 in order to pretend the Receiver is simply trying to reject the Sender's attempt to establish a three-way handshake. In reality though, the Receiver is actually confirming the original packet's reception and also passing its "signature" to the Sender by placing it in the answer's Identification field.

10. The Receiver then proceeds to deliver its answer packet.

11. Upon receiving the answer packet, the Sender sets the random function's seed to the sum of the original packet's IP identification and sequence number values. It then compares the random function's output with the answer's Identification field (i.e. the Receiver's "signature"): if the two values differ, it implies the original packet's header fields were somehow altered after the packet's departure

(likely due to proxying), ultimately meaning the transmission failed and must stop. Otherwise, the Sender can continue its transmission and prepare the next forged packet.

3.3 Forging packets and connections

In the last section we illustrated the steps followed during the definition of our new steganography method. In this section we will instead illustrate the most important steps of the implementation process itself, in particular those aimed at creating forged packets that need to look as realistic as possible. In order to keep implementation as simple as we could, we opted to write both the Covert Sender and the Covert Receiver in Python, more specifically using Scapy. Scapy is a packet manipulation libary capable of forging or decoding packets belonging to a wide range of protocols, sending them on the wire, capturing them (while also applying filters if necessary) and much more[1]. In our specific case, we used said library to handle both the creation, the transmission and the reception of forged TCP packets.

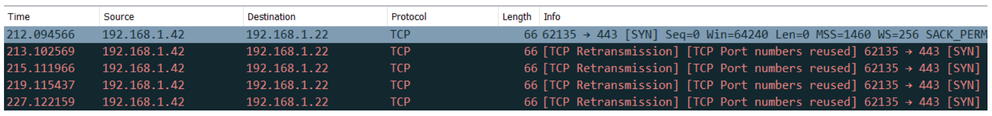
3.3.1 Imitating a failed three-way handshake in Windows

The main principle we followed while writing our code was to replicate as faithfully as possible the sequence of events that normally occurs in Windows 11 when an attempt to start a new three-way handshake fails. The reason behind this is that since we made no assumptions just yet in regards to how the warden might try to stop us, our best shot at improving stealthiness consists at this time in just making our transmissions as akin to regular ones as possible. Should we encounter problems due to specific countermeasures on part of the warden, we shall then reflect on how to further adapt our program accordingly. This ultimately implies the Sender program must also try to gather as much information as possible from the local settings before starting its transmission.

The basic behaviour we ultimately aim to replicate is displayed in Figure 3.3 and Figure 3.4, respectively representing a failed attempt to create a three-way handshake in Windows due to the destination IP sending no reply (Figure 3.3) or replying with

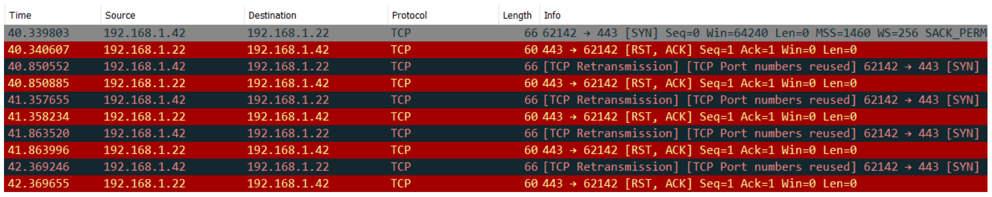
[1]https://github.com/secdev/scapy

RST (Figure 3.4).



Time	Source	Destination	Protocol	Length	Info
212.094566	192.168.1.42	192.168.1.22	TCP	66	62135 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
213.102569	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
215.111966	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
219.115437	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
227.122159	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]

Figure 3.3: Wireshark snapshot showing a failed attempt to create a three-way handshake in Windows (the destination host sent no reply).



Time	Source	Destination	Protocol	Length	Info
40.339803	192.168.1.42	192.168.1.22	TCP	66	62142 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
40.340607	192.168.1.22	192.168.1.42	TCP	60	443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
40.850552	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
40.850885	192.168.1.22	192.168.1.42	TCP	60	443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
41.357655	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
41.358234	192.168.1.22	192.168.1.42	TCP	60	443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
41.863520	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
41.863996	192.168.1.22	192.168.1.42	TCP	60	443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
42.369246	192.168.1.42	192.168.1.22	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
42.369655	192.168.1.22	192.168.1.42	TCP	60	443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figure 3.4: Wireshark snapshot showing a failed attempt to create a three-way handshake in Windows (the destination host replied with RST).

As we can see, in both cases the TCP/IP stack of Windows always tries multiple times before stopping and reporting an error: upon retransmission, a given packet's IP identification field is always increased by one but its TCP source port and initial sequence number remain the same. The total number of retransmissions is the same in both cases, but in the "no answer" scenario the initial waiting time is always doubled after each failure (as Windows assumes the answer might have simply taken too long to arrive due to bad network quality). Our transmission shall follow the same pattern, in that the same forged packet shall be retransmitted a specific number of times and the Covert Sender shall also spend a specific amount of time waiting for the Receiver's answer after each transmission (doubled in case the answer fails to arrive). In Windows 11 the default number of retransmissions is four and the default initial waiting time is one second, but both settings can theoretically be changed. Therefore, the Sender shall always preemptively scout their value by calling the Windows **"Get-NetTCPSetting -Setting Internet"** command and adapt its behaviour accordingly. The Receiver on the other hand must always take into account the Sender always retransmits every packet multiple times: since unlike the Sender it cannot know how many times exactly, it shall ultimately simply discard any packet that has the same sequence number of the one that arrived immediatly before.

3.3.2 Initializing the forged packet's fields

Before beginning transmission, the Sender must also make sure every field of the forged packet is initialized properly. Some of them are quite straightforward:

- **Destination address (IP)** must obviously be the Covert Receiver's.
- **Initial sequence number (TCP)** is determined during the embedding process.
- **Acknowledgement number (TCP)** must be 0 since the Sender's forged packet is the first one of a new connection attempt.
- **Flags (TCP)** must be 2 (corresponding to SYN being set to 1 and everything else to 0).
- **Destination port (TCP)** must reflect a service allowed on the private network (in our tests we used port 443).
- **Version (IP)** must be 4 (since we are using ipv4).
- **Protocol (IP)** must be 6 (since we are using TCP).
- **Urgent pointer (TCP)** must be 0 (since the packet's payload contains no data).
- **Fragmentation offset (IP)** must be 0 (since the packet is very small and thus not subject to fragmentation).
- **Initial header length (IP)**, **Total length (IP)**, **Data offset (TCP)**, **Checksum (IP)** and **Checksum (TCP)** are calculated automatically after all other fields are initialized.

The IP source address field is a bit more tricky as it should always reflect an interface (among the compromised host's active ones) that is both connected and capable of reaching the internet. In order to make sure a chosen interface can indeed reach the internet, the Sender shall check scapy's routing table through the **"conf.route"** variable. In order to make sure it is also actively connected, the Sender shall check its connection status using the **"netsh int ipv4 show interfaces"** Windows command.

Finally, the Sender shall also use the **“ipconfig”** and **“arp”** Windows commands to discover the MAC addresses of both the chosen interface and its default gateway. Manual configuration of transmission parameters at the link level (layer 2 of the OSI reference model) should have not been necessary and goes a bit off topic in regards to our original goal of only working with layer 3 and layer 4. Unfortunately, the scapy library functions that were designed to make this process automatic are affected by bugs (at least when called in Windows 10 and Windows 11), leaving us no choice but to handle the layer 2 transmission ourselves.

The remaining fields of the TCP and IP headers should be initialized with values that are as similar as possible to the ones seen in authentic packets. The easiest way to make sure chosen values are correct is to clone them from a sample packet generated by the real TCP/IP stack: this can be achieved by launching a subprocess that attempts to open a socket through any of the most commonly used APIs (thus causing a system call that summons the TCP/IP stack) while simultaneously using scapy’s *sniff()* function in the main process to intercept the packet created to fullfill the subprocess’ request. In order to remain as silent as possible, the destination of the subprocess’ connection attempt should always be the loopback address: this way the generated packet never leaves the compromised host and cannot be seen on the network (though it remains possible to sniff it from the compromised host itself). The fields that can be successfully copied using this strategy are TTL (IP), Flags (IP), Options (IP), Source port (TCP) and all TCP Options except “maximum segment size” (which is interface specific and in this case has a much greater value than usual). Among them, the most crucial is by far the TCP source port as it would otherwise be very hard to set. This is because the Covert Sender would need to sniff the last source port value used by a regular SYN packet and increase it by one: depending on how many authentic TCP connections the real TCP/IP stack is creating at that time, there could be either the risk of collision (in that the TCP/IP stack also uses the same value) or a very long wait time before a value is eventually scouted. Attempting a connection on the loopback interface instead allows the Sender to obtain immediatly the source port value it needs and also prevents the possibility of another connection using the same value (since the TCP/IP stack itself effectively “reserved” it for the loopback connection).

3.3.3 Initializing interface dependent fields

A few additional fields are however interface dependent and cannot be copied from the connection attempt that takes place on the loopback interface. Their values can either be set to their expected value in Windows (the most silent but also less accurate choice) or can be scouted using the same strategy as before, this time however using the Covert Receiver's address as destination for the subprocess' connection (thus generating accurate values but also creating a connection attempt that is visible on the private network). In the case of the second option, in order to allow the Receiver to distinguish connection attempts that carry secret information from connection attempts used for the mere purpose of scouting these values, the attempts shall occur on different destination ports (for example 443 for connection attempts that carry secret information and 80 for connection attempts used for scouting). Fields that can be scouted this way include:

- **Terms of Service (IP)**, which we chose to include in this category due to its various purposes and multiple redefinitions over the years making it hard for us to predict in which way the field could be used. Nevertheless, we must also point out we never saw the field ever being set to anything but zero (meaning we do not expect any negative consequence in case such a value is used).

- **Maximum segment size**, which is a specific option of TCP that is directly determined by the interface's maximum transmission unit. Since most interfaces have a MTU of 1500, this option is usually set to 1460: 1500 minus the 40 bytes of the TCP and IP headers combined (options excluded).

- **Window (TCP)**, a field always set to its maximum value as long as the TCP window scale option is used[2] (which is basically always the case in 2023). The problematic aspect of this field is that according to Windows documentation said maximum value is supposed to be 65535, but in our experience the value 64240 is also used quite often (possibly even more frequently than 65535). The reason for this is likely that 65535 is not a multiple of 1460 (the maximum segment size) while 64240 is. Cloning this field's value from an authentic packet

[2]https://learn.microsoft.com/en-us/troubleshoot/windows-server/networking/description-tcp-features

ensures the Sender's forged packets follow the same behaviour adopted by regular packets. Otherwise, we do not believe using 65535 as default value would cause any trouble (since it would simply imply following Windows' documentation).

- **Identification (IP)**, in that we can take note of the last regular packet's identification value and then simply increase it by one with every new forged packet we send. Alternatively, the first value can be simply initialized randomly.

In order to ensure the Identification field behaves as consistently as possible, after finishing all operations the Sender should also save its last used value in a local configuration file (or sqlite database) so that any future launch of the program can resume the sequence from where it stopped (assuming of course this new launch uses the same source address and destination address for its transmission). In case of system reboot however, all Identification values are always reinitialized, meaning database information should also be reset. Since the last boot time can be obtained through the **"Wmic os get lastbootuptime"** Windows command, the database shall store this information as well. Before accessing the database the Sender shall ultimately always call this command once again in order to compare its output to the value stored in the database: if they differ, it ultimately implies a system reboot occurred at some point and the database must thus be cleared.

3.4 A word on robustness

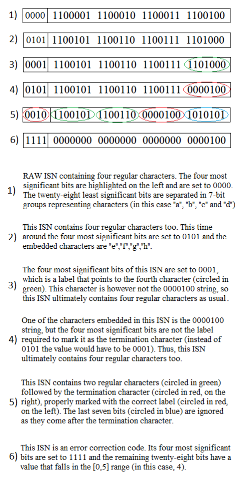
In normal circumstances, the TCP protocol offers good robustness thanks to its error detection, flow control, congestion control and retransmission capabilities. In our case however, even though we are technically using TCP ourselves, we do not have access to these useful features because our packets do not use their payload to carry information (let alone our program never completes any three-way handshake either). As a result, transport becomes unreliable (much like if we were using UDP) and the responsibility of handling possible errors falls on the Sender. In this section we shall thus illustrate the adaptations we introduced in our code in order to handle the most common issues.

3.4.1 Termination character and error correction codes

In section 3.3.1 we mentioned how the Sender program transmits each packet multiple times in order to replicate the normal sequence of events that normally occurs in Windows when a three-way handshake fails. Unfortunately, this does not improve robustness at all because the Receiver can only obtain secret information from the original packet and never from any of its retransmissions: this happens because while the sequence number remains indeed the same, the identification value does not, ultimately preventing the Receiver from recreating the bit mask it needs to correctly extract the secret characters.

In light of this, the Sender can consider a packet's transmission successful only if the Receiver replies with RST to the very first packet: what happens to any of the following retransmissions is completely irrelevant. The Receiver on the other hand must only accept information from the first packet and discard information contained in any subsequent packet that has the same sequence number (though it shall still send an answer to them). This also implies the Receiver must keep track of the last sequence number received from each different Sender, and that each different Sender shall also notify the Receiver when there is no more secret text to send (thus granting the Receiver permission to delete the associated sequence number record).

Among symbols offered by original ASCII we decided to commit the 0000100 bit string (corresponding to the EOT character) to this purpose, meaning the Sender shall always append it as last character to its secret text before beginning operations. However, we also decided this "termination" character shall be marked with an additional label so that the Sender remains generally free to send a generic 0000100 bit string without inadvertently ending the transmission. As we already anticipated when we described the embedding process, we shall use the ISN field's four most significant bits for this purpose: if one of the four secret characters embedded in the twenty-eight least significant bits is the string 0000100 used with the intent of ending transmission, the corresponding bit among the four most significant ones shall be set to 1 and the remaining shall be set to 0 (i.e. 0001 if the termination character is the last one, 1000 if it is the first one, etc.).



1) RAW ISN containing four regular characters. The four most significant bits are highlighted on the left and are set to 0000. The twenty-eight least significant bits are separated in 7-bit groups representing characters (in this case "a", "b", "c" and "d")

2) This ISN contains four regular characters too. This time around the four most significant bits are set to 0101 and the embedded characters are "e","f","g","h".

3) The four most significant bits of this ISN are set to 0001, which is a label that points to the fourth character (circled in green). This character is however not the 0000100 string, so this ISN ultimately contains four regular characters as usual .

4) One of the characters embedded in this ISN is the 0000100 string, but the four most significant bits are not the label required to mark it as the termination character (instead of 0101 the value would have to be 0001). Thus, this ISN ultimately contains four regular characters too.

5) This ISN contains two regular characters (circled in green) followed by the termination character (circled in red, on the right), properly marked with the correct label (circled in red, on the left). The last seven bits (circled in blue) are ignored as they come after the termination character.

6) This ISN is an error correction code. Its four most significant bits are set to 1111 and the remaining twenty-eight bits have a value that falls in the [0,5] range (in this case, 4).

Figure 3.5: Visual representation of some "raw" ISNs aimed at illustrating the possible uses of the four most significant bits.

In any other case the message is instead a normal message containing four regular characters: this also includes the scenario in which one of the characters is the 0000100 string but no 4-bit label is used to mark it and the scenarios in which one of the 4-bit labels is used but the marked character is not the 0000100 string. There is however one important exception: when the four most significant bits are all set to 1 and the twenty-eight least significant bits have a value that falls in the [0, 5] range (for a total of six possibilities), the message contains no secret information and is instead an error correction code used by the Sender to fix a transmission error. Figure 3.5 ultimately provides some examples of different "raw" ISNs (i.e. before the randomized bitmask is applied to them) aimed at illustrating the different possible uses of the four most significant bits.

3.4.2 Common hazards

The error correction codes we introduced in the last section play a vital role when certain specific problems are encountered. Speaking more generally, several unexpected events can potentially occur during the Sender's transmission: in this section we shall illustrate the most common ones, their possible causes and what the Sender can do in response to each.

One relatively common scenario consists in one of the Sender's retransmissions receiving no answer. This happens because either the packet itself or the Receiver's RST answer is dropped. As we already mentioned, retransmission packets are irrelevant and the Receiver ignores them by default. The Sender can thus simply ignore the problem and continue its operations.

Another scenario consists in the Sender's packet never receiving any answer (both the first time it is transmitted and after each retransmission). Since for the average network we do not expect packet loss to be higher than 1% at worst, such a situation likely implies the Receiver is either not operational or unreachable. The Sender shall thus stop all operations.

Another scenario consists in the Receiver replying to the Sender's packet with an answer packet that has the wrong signature in the Identification field. As we already explained when we described the embedding process, this indicates either the original packet's sequence number or its identification value were altered while the packet was traveling (likely due to proxying), ultimately making it impossible for the Receiver to extract information from it. The Sender shall thus abort all operations.

Another scenario consists in the Sender's packet not receiving an answer the first time it is sent, but receiving one (or more) during its following retransmissions. Since the Receiver managed to answer at least once, it ultimately implies it is both operational and reachable. Either the original packet itself or its RST answer were however dropped: these two possible causes lead to two different "variants" of this scenario.

The first variant occurs when the Sender's packet is dropped (Figure 3.6). In this case, the Receiver effectively fails to receive the Sender's information and wrongly identifies one of its following retransmissions as a relevant packet (since from its point of view it was the first one to have a new sequence number). Because of this, the Receiver ultimately extracts wrong information from it and stores it as if it was part of the secret text sent by the Sender: most of the time, this wrong information consists in four gibberish characters, but sometimes it could include the special "termination" character or in worst case scenario even be an error correction code.

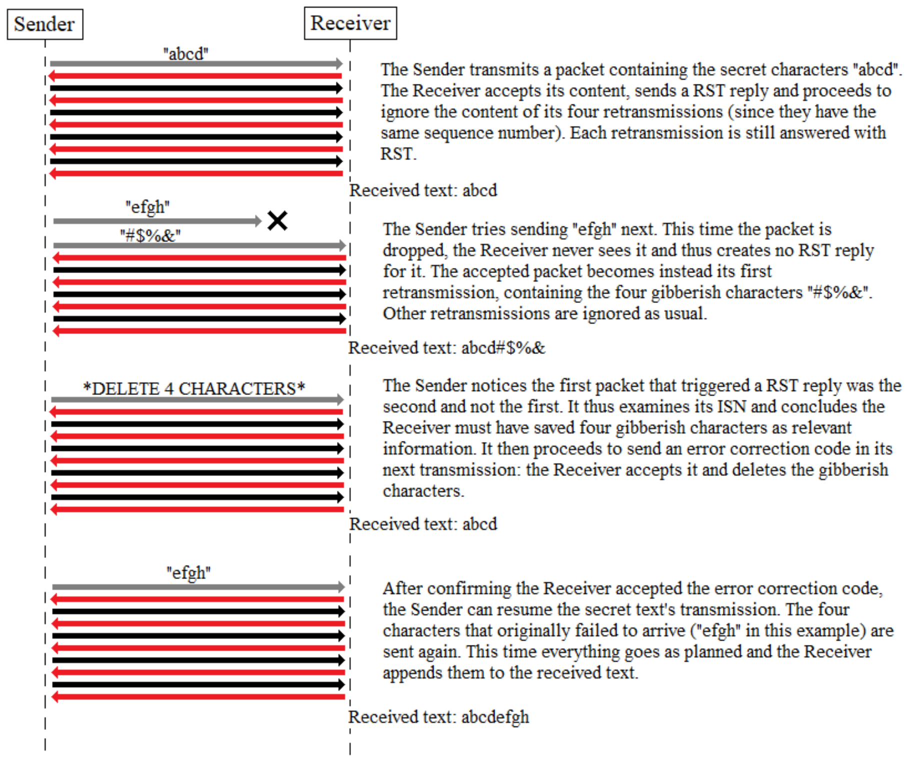

Sender
Receiver
"abcd"
The Sender transmits a packet containing the secret characters "abcd". The Receiver accepts its content, sends a RST reply and proceeds to ignore the content of its four retransmissions (since they have the same sequence number). Each retransmission is still answered with RST.
Received text: abcd
"efgh"
"#$%&"
The Sender tries sending "efgh" next. This time the packet is dropped, the Receiver never sees it and thus creates no RST reply for it. The accepted packet becomes instead its first retransmission, containing the four gibberish characters "#$%&". Other retransmissions are ignored as usual.
Received text: abcd#$%&
DELETE 4 CHARACTERS
The Sender notices the first packet that triggered a RST reply was the second and not the first. It thus examines its ISN and concludes the Receiver must have saved four gibberish characters as relevant information. It then proceeds to send an error correction code in its next transmission: the Receiver accepts it and deletes the gibberish characters.
Received text: abcd
"efgh"
After confirming the Receiver accepted the error correction code, the Sender can resume the secret text's transmission. The four characters that originally failed to arrive ("efgh" in this example) are sent again. This time everything goes as planned and the Receiver appends them to the received text.
Received text: abcdefgh

Figure 3.6: Visual representation of the packet drop scenario's first "variant". Grey arrows represent the first packet delivered by the Sender in each connection attempt (i.e. the one from which the Receiver extracts information), black arrows represent its retransmissions and red arrows represent the Receiver's RST replies.

Fortunately, the Receiver's first answer allows the Sender to detect this error because it always matches the packet accepted as relevant. If such a packet is not the original one, the Sender can calculate its associated bit mask from its Identification value, remove it from the pacekt's sequence number field (as usual, through a bitwise XOR operation) and obtain itself the wrong information mistakenly accepted by the Receiver. Depending on this information's nature, the Sender can then embed an appropriate error correction code in its next connection attempt. The six possible codes respectively instruct the Receiver to:

- **Delete the last "termination" character and the three characters immediatly before**, which is used if the wrong information consists of three regular characters followed by the termination character (happens with a probability of roughly 0.0005%).

- **Delete the last "termination" character and the two characters immediatly before**, which is used if the wrong information consists of two regular characters followed by the termination character (happens with a probability of roughly 0.0005%).

- **Delete the last "termination" character and the single character immediatly before**, which is used if the wrong information consists of a single regular character followed by the termination character (happens with a probability of roughly 0.0005%).

- **Delete the last "termination" character**, which is used if the wrong information is just the termination character (happens with a probability of roughly 0.0005%).

- **Acknowledge that a "fatal error" has occurred**, which is used if the wrong information is an error code that caused the Receiver to delete part of its stored secret text when it was not supposed to (happens with a probability of roughly 0.0000000014%). This is the only error code after which the Sender cannot resume its operations and must instead abort transmission.

- **Delete the last 4 characters**, which is used if the wrong information consists of four gibberish regular characters (happens in the remaining cases, i.e. roughly 99.997% of the time).

After the error correction code arrives successfully the Sender can ultimately resume its operations, but the four characters that originally failed to arrive must be embedded again in the next connection attempt (the "fatal error" case being the only exception in which transmission stops altogether).

The second variant of this scenario occurs when the Sender's packet arrives correctly but the Receiver's RST answer is dropped (Figure 3.7). In this case the Receiver effectively accepts the correct packet but the Sender mistakenly convinces itself a retransmission was accepted in its place. In such a situation, the Sender should theoretically do nothing at all. Unfortunately, the Sender has ultimately no way to know whether the lost packet was the original one or its answer (i.e. distinguish the scenario's two variants). We thus concluded the safest choice consists in having it always assume the dropped packet is the original, in order to always prompt it to send an error correction code. As we have seen, this error correction code consists 99.997% of the time in an order that causes the Receiver to delete the last four characters received, which are then embedded again in the Sender's next connection attempt. If the lost packet is actually the Receiver's answer (i.e. the Sender's assumption turns out to be wrong), the Sender's error correction code simply causes the Receiver to delete four correct characters that are then sent again anyway: this ultimately slows down the communication but does not damage it.

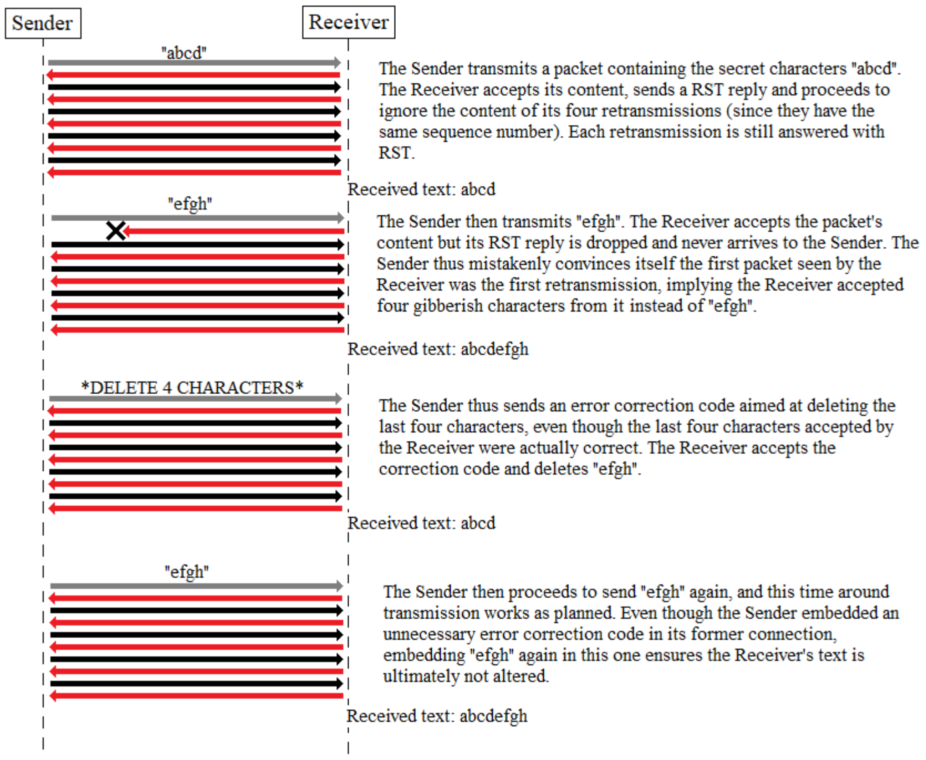

Sender
Receiver
"abcd"
The Sender transmits a packet containing the secret characters "abcd". The Receiver accepts its content, sends a RST reply and proceeds to ignore the content of its four retransmissions (since they have the same sequence number). Each retransmission is still answered with RST.
Received text: abcd
"efgh"
The Sender then transmits "efgh". The Receiver accepts the packet's content but its RST reply is dropped and never arrives to the Sender. The Sender thus mistakenly convinces itself the first packet seen by the Receiver was the first retransmission, implying the Receiver accepted four gibberish characters from it instead of "efgh".
Received text: abcdefgh
DELETE 4 CHARACTERS
The Sender thus sends an error correction code aimed at deleting the last four characters, even though the last four characters accepted by the Receiver were actually correct. The Receiver accepts the correction code and deletes "efgh".
Received text: abcd
"efgh"
The Sender then proceeds to send "efgh" again, and this time around transmission works as planned. Even though the Sender embedded an unnecessary error correction code in its former connection, embedding "efgh" again in this one ensures the Receiver's text is ultimately not altered.
Received text: abcdefgh

Figure 3.7: Visual representation of the packet drop scenario's second "variant". Grey arrows represent the first packet delivered by the Sender in each connection attempt (i.e. the one from which the Receiver extracts information), black arrows represent its retransmissions and red arrows represent the Receiver's RST replies.

Unfortunately, there are still two extremely rare situations in which the communication ends up irreparably compromised. The first one happens when the packet drop occurs while the Sender is trying to transmit an error correction code, regardless of which variant in particular. In such a situation, the Sender cannot resort to a single action capable of single handedly fixing both possible variants of the packet drop scenario and is ultimately forced to take a blind guess in regards to which of the two variants might have occurred. Assuming the packet drop rate is the same in both directions, this would result in a 50% chance of the Sender choosing the wrong action, which we ultimately consider too high: as a result, we concluded the Sender shall simply abort operations if such a situation occurs. Ultimately, this implies two transmission errors in a row are never tolerated in our program. Since error correction codes are only used when a packet drop scenario already occurred once, the probability of such a situation is ultimately equal to the probability of the packet drop scenario occurring twice in a row. This probability is obviously influenced by the network's packet drop rate: assuming no more than 1% of all packets are dropped and taking into account how any one among two packets (the Sender's original packet and the Receiver's RST) lead to this scenario if dropped, the final probability can be estimated to be 0.00004% for each connection attempt.

The second one happens when the Receiver's answer is dropped and the Sender mistakenly believes the Receiver accepted a packet containing a termination character or an error code. This causes the Sender to transmit an error correction code different from the "Delete the last 4 characters" one, resulting in an alteration of the Receiver's accepted text that is not automatically corrected with the Sender's next transmission. Our program has currently no way to deal with such a situation, which fortunately remains very unlikely: as we have seen, the probability of the Sender using an error correction code different from "Delete the last 4 characters" is roughly 0.0020000014% and the probability of a packet being dropped in the first place is not expected to be higher than 1% at worst. The ultimate probability of such a situation is thus roughly 0.000020000014% for each connection attempt.

3.4.3 Network architectures and operability

On a final note, we must also point out that our program is designed to function in a relatively specific network architecture and does not necessarily still work if such architecture is changed too radically. Specifically, we expect the Receiver to either possess its own public IP address or alternatively to be on a private network where https packets are forwarded towards it (specifically by opening port 443). The Sender on the other hand is expected to either be on a private network equipped with a default gateway that allows it to reach the outer Internet or to possess its own public IP address. In the former case, the private network can have at most one transmitting Sender: if two or more Senders were to transmit simultaneously, the Receiver would have problems distinguishing each one's packets as their source address is overwritten due to NAT. We do believe the Receiver could theoretically be adapted to include some form of multiplexing based on the Identification field, but we chose not to prioritize such a feature at this time. Finally, neither the Sender nor the Receiver are expected to function if they are launched on a virtual machine that gains internet access through NAT mode (though they do if Bridge mode is used instead).

3.5 Field tests

Our program was ultimately tested across six different environments, all of which featured very small networks (i.e. at most three hosts on each). Both the Sender and the Receiver were always connected to their respective networks using private IP addresses. More specifically:

- In the first environment both the Sender and the Receiver were located on the same private network and gained internet access through UTP cables. This environment was mainly used to conduct tests that allowed us to correct bugs while the code was still being written.
- In the second environment the Receiver was connected to a private network using a UTP cable while the Sender was given internet access exclusively through a Wi-Fi hotspot created with a nearby mobile device (meaning even though they were physically in the same room, the two hosts were ultimately on separated networks).
- In the third environment the Receiver and the Sender were connected to two different private networks located in different cities altogether. Speficially, the Receiver was located in Perugia (Italy) and the Sender was located in Milan (Italy).

- In the fourth environment the Receiver and the Sender were once again connected to two different private networks located in different cities. This time around the Receiver was still located in Perugia (Italy) but the Sender was located in Rome (Italy).
- In the fifth environment the Receiver and the Sender were connected to two different private networks, both located in Perugia (Italy). The Sender was launched from a virtual machine that obtained internet access in Bridge mode.
- The sixth environment was entirely virtual: both the Sender and the Receiver were virtual machines, respectively connected to two different internal virtual networks. A third virtual machine was then given access to both virtual networks and configured to allow packet transit so that it could act as gateway in both directions. All virtual machines and networks were hosted on the same computer.

These environments were at first used to carry out some preliminary tests aimed at ensuring the channel worked as intended. The length of the secret messages sent in these tests varied from a minimum of 4 to a maximum of 712 characters. In several cases a few packets were lost, but the error correction codes worked as intended and preserved the transmissions' integrity: with the exception of some initial failures that were caused by various bugs that needed correction, our tests were ultimately successful and the Receiver obtained the secret message as intended. We estimate the average packet drop rate to have been slightly less than 1%, confirming one of the initial assumptions we made when assessing robustness.

3.5.1 Assessing stealthiness

In order to evaluate the stealthiness of our transmissions we first used Wireshark in three different instances to capture traffic on the Sender's network while our program was running. We ultimately produced three .pcap files with the intent of later feeding them as input to programs designed to scan network traffic in search of anomalies. All three .pcap files were recorded in the second environment and thus included only traffic related to a single host (i.e. our compromised host), with very sporadic activity aside from the single secret communication featured in each. The secret communication was also slightly different in each instance. In order:

- **PCAP N.1** featured the transmission of 712 characters. Each one of the Sender's connection attempts included one original packet (containing four secret characters) and four retransmissions of it. After each connection attempt, the Sender was instructed to wait a 1 second delay.
- **PCAP N.2** featured the transmission of 356 characters. Each one of the Sender's connection attempts included one original packet (containing four secret characters) and four retransmissions of it. After each connection attempt, the Sender was instructed to wait a 2 seconds delay.
- **PCAP N.3** featured the transmission of 36 characters. Each one of the Sender's connection attempts included one original packet (containing four secret characters) and four retransmissions of it. After each connection attempt, the Sender was instructed to wait a 300 seconds delay.

We then proceeded to install two different network security programs. The first one was Suricata (version 6.0.1), an open-source intrusion detection system (IDS) developed by Open Information Security Foundation (OISF) that uses rule-based and signature-based detection to monitor network traffic for signs of suspicious activity [3]. Suricata's rules are written in a special language called Suricata Rule Language (SRL), and many online communities actively engage in the production of new rules in order to keep Suricata's performance up to date: in light of this, one of Suricata's major advantages consits in how easily it can be enhanced and customized with the addition of new rules. We thus used the **"suricata-update"** command to load roughly 47000 rules from the following sources (all of which are free):

- etnetera/aggressive
- malsilo/win-malware
- sslbl/ja3-fingerprints
- sslbl/ssl-fp-blacklist
- oisf/trafficid
- et/open

[3]https://suricata.io/

- stamus/lateral
- tgreen/hunting

The second security software we installed was Zeek (version 5.1.1). Zeek is an open-source network analysis tool capable of monitoring traffic and producing logs that can be useful both to detect anomalies and to conduct further analysis with additional threat detection modules (specifically designed to be compatible with zeek logs)[4]. Similarly to Suricata, Zeek too can be customized and extended to better fit a given user's needs. In this sense, Zeek provides its own scripting language, allowing users to write their own custom scripts and even upload them as packages to make them available to other users. We thus extended our Zeek installation by adding the following packages:

- conn-burst
- ja3
- HASSH
- BZAR
- dovehawk
- spicy-analyzers
- pingback
- icmp-exfil-detection
- http_csp
- http-stalling-detector
- zeek-httpattacks

We then used both programs to scan our .pcap files in order to see if they were able to detect our secret transmissions: neither Suricata nor Zeek ultimately raised any alert. It should be noted that intrusion detection systems of this kind are only as

[4]https://zeek.org/

good as their configuration, meaning with better packages or rules the results might have been different. Nevertheless, for now we feel safe to confirm what Iglesias et al. had already pointed out one year ago [8]: while many techniques aimed at defeating network steganography have already been proposed, many existing network security tools have not necessarily implemented them just yet.

Chapter 4

Defeating our own covert channel

In the last chapter we finished our analysis by showing how a generic IDS with a general purpose configuration likely does not have a very good chance of spotting our covert channel. In this chapter we will show how security measures that are more specifically tailored to counter the type of transmission used in our channel can be far more effective at detecting or suppressing it.

4.1 TCP proxying

Ever since the first time we introduced ISN modulation based covert channels, we always claimed the most effective way to stop them would be proxying all TCP connections in order to ensure initial sequence numbers are always rewritten, as first suggested by Murdoch and Lewis in 2005 [16]. In Chapter 3 we also mentioned how what was originally considered only a theoretical possibility might have already turned into a reality in light of the significant progress made by hardware and computational power in the last twenty years.

Since our tests remained successful even when conducted on three different real networks we do not believe proxying all TCP connections has become an internet standard in the general case just yet. Unfortunately, we cannot entirely rule out the possibility that middle-boxes aimed at proxying TCP connections are actually used more frequently than we think and that we were simply lucky enough not to stumble in any of them during our own tests. Nevertheless, proxying all TCP connections would improve network security at the cost of reducing network efficiency due to the additional overhead: in light of this, we do not expect it to ever become a default policy in middle-boxes that need to handle high volumes of traffic (unlike other policies such as reassembling packet fragments [10], which are effectively useful at improving network quality).

Rewriting initial sequence numbers using a randomized value for security reasons is however already a relatively popular feature offered by certain firewalls and security devices aimed at guarding individual networks: a good example of this is Cisco ASA[1]. We ultimately believe deploying one of these devices would be the most effective way to prevent our own covert channel from operating: similarly to other measures generally adopted by active wardens, the operation of rewriting initial sequence numbers can be simply applied as a default policy in all situations, leaving our channel no room to adapt.

4.2 RITA (Real Intelligence Threat Analytics)

RITA (Real Intelligence Threat Analytics) is an open source software designed to analyze network traffic in order to detect signs of malicious activity, in particular beacons associated with command and control software[2]. A beacon is a transmission pattern that consists of communications that look strangely regular and programmatic: while not all beacons are necessarily associated with malicious activities, command and control programs often generate such patterns because they instruct the infected host to send regular pings to the attacker's server in order to signal it is ready to receive remote commands. While our own covert channel is not a command and control software itself, it does nevertheless resort to a communication that follows a very regular pattern due to packet retransmissions, ultimately making it very vulnerable to RITA's analysis.

RITA is designed to scan input data that is stored in a database where entries can be either "simple" or "rolling"[3]. Simple entries consist of data generated from a single .pcap file, whereas rolling entries consist of data generated from multiple .pcap files stored over time, called "fragments" : these fragments are "rolling" because once

[1]https://www.cisco.com/c/en/us/td/docs/security/asa/asa99/configuration/firewall/asa-99-firewall-config/conns-connlimits.html
[2]https://www.activecountermeasures.com/free-tools/rita/
[3]https://github.com/activecm/rita

they reach a maximum number, the next .pcap file stored in the database entry automatically replaces the oldest. In RITA's default configuration a rolling database entry stores up to twenty-four fragments, each one covering one hour of network traffic (for a total of twenty-four hours). During the scan process itself, individual connections are first gathered into different sets: each one contains all connections that went from a same source address to a same destination address. Each of these sets (i.e. each *(Source, Destination)* tuple) is then assigned a global score that ranges from 0 (most definitely not malicious) to 1 (most definitely malicious) based on statistical analysis. This analysis only takes place if the number of connections included in a given set surpasses a certain "grace threshold", otherwise the set is simply ignored and its connections are ultimately considered not malicious (in RITA's default configuration this threshold is set to 20). The statistical analysis takes into account four main aspects[4], each one associated with its own individual score: the global score is none other than their mean, though RITA can be configured to assign different weights as well. In order:

- **Timestamp score** is based on the frequency and regularity of network traffic over time. Repeated transmissions with fixed intervals between them score higher, while sporadic transmissions with irregular intervals between them score lower. According to RITA's documentation[5], timestamp score is calculated as $(\frac{1}{3}(1 - \mathit{TSBowleySkew}) + \frac{1}{3}(\max(1 - \frac{\mathit{TSMADM}}{30}, 0)) + \frac{1}{3}(\mathit{TSConnCount}))$ where "TSConnCount" is the ratio of the number of connections to the number of 10 second periods in the whole dataset, "TSBowleySkew" is the Bowley Skew of intervals[6] and "TSMADM" is Median Absolute Deviation around the median of intervals[7].

- **Data size score** is based on how how much data is transferred with each connection. Transmissions that always send the same amount of data score higher, while transmissions that send different amounts of data score lower. According to RITA's documentation, data size score is calculated as $(\frac{1}{3}(1 - \mathit{DSBowleySkew}) + \frac{1}{3}(\max(1 - \frac{\mathit{DSMADM}}{32}, 0)) + \frac{1}{3}(\max(1 - \frac{\mathit{DSMode}}{65535}, 0)))$

[4]https://github.com/activecm/rita/blob/master/pkg/beacon/analyzer.go
[5]https://github.com/activecm/rita/tree/master/pkg/beacon
[6]https://mathworld.wolfram.com/BowleySkewness.html
[7]https://en.wikipedia.org/wiki/Median_absolute_deviation

where “DSMode” is data size mode (i.e. the data size that appears the most often), “DSBowleySkew” is the Bowley Skew of data sizes and “DSMADM” is Median Absolute Deviation around the median of data sizes.

- **Duration score** is simply the ratio between the timespan in which transmissions persist and the timespan covered by the database entry as a whole. For example if all transmissions occur within a single hour and the database entry covers one single day, the score will be approximatively 0.0417 (i.e. one divided by twenty-four).

- **Histogram score** can be assigned through two different possible calculations: in both cases the total timespan covered by the database entry is first divided in twenty-four buckets in order to create a class distribution of all connections (hence the name “histogram score”). Exploring these calculations in detail goes beyond the purpose of this work: the most relevant detail of this score is that a very low duration score automatically implies a very low histogram score.

We ultimately used RITA to scan the same three .pcap files we had originally created for our tests with Suricata and Zeek (for each .pcap file we created a dedicated “simple” database entry in RITA to scan them individually). As we feared, RITA assigned a quite high score to all three .pcap files: **0.935** to N.1, **0.825** to N.2 and **0.743** to N.3 respectively. It should be noted however that RITA did not ultimately identify any of our transmissions as a covert channel and merely harbored suspicions related to their communication pattern. Moreover, since RITA’s logic is that virtually any transmission could be a beacon it is not uncommon for regular, non-malicious network traffic to show up as a false positive. This allowed us to conclude that despite its initial bad performance, our channel could be adapted to slightly improve its stealthiness against RITA (at least when RITA follows its default configuration). More specifically, our channel can change its behaviour in the attempt to lower its score as much as possible and pretend it is simply one of the many false positives. This behaviour change ultimately consists of four modifications that must be applied to the Sender program (their combined effect is shown in Figure 4.1).

The first modification consists in the Sender always operating just one single retransmission of each SYN packet, rather than a number aimed at reflecting the compromised host’s Windows settings: by reducing retransmissions the Sender’s communication becomes more sporadic, which supposedly improves its timestamp score. Moreover, if the secret text is short enough the Sender might be able to convey it using a number of packets that does not exceed RITA's "grace threshold", thus avoiding detection altogether. For example, considering how each forged packet's sequence number can transport a total of 28 secret bits, exfiltrating a 256-bit key requires no more than 10 packets (assuming packet loss does not occur): if each packet is only retransmitted once, the total number of outgoing SYN packets is exactly 20, meaning RITA will not raise any alert unless its settings were edited to lower its grace threshold. Eliminating retransmissions altogether would furtherly increase the Sender's chances of avoiding detection entirely, but doing so would also penalize robustness too much: we thus concluded the best compromise is likely to keep one single retransmission for each forged SYN packet.

The second modification consists in the Sender refraining from creating one single real connection towards the Receiver in order to scout the values of fields such as Identification (IP) or Maximum Segment Size (TCP option) and later use them to initialize forged packets. The logic behind this modification is once again that reducing the total number of transmissions is vital in order to reduce RITA's timestamp score as much as possible. The forged packets' fields shall ultimately be initialized with default values that look as realistic as possible (for example it is extremely unlikely that Maximum Segment Size has a value different from 1460).

The third modification consists in the Sender limiting its communication to a very specific time window, which ideally should be no wider than a twenty-fourth of the total timespan covered by RITA's database entry. Unlike other modifications, this one did not require us to change the Sender's code but rather record our future .pcap files in a way that properly reflects this behaviour. The ultimate goal of this modification is to bring down RITA's duration score to approximatively 0.05 (which would also guarantee a histogram score of almost 0). In this sense, the three .pcap files we originally recorded had a bias: since we started our captures right before beginning the secret transmission and finished them right after the last forged packet's arrival, the resulting duration score was always very high.

The fourth modification consists in the Sender waiting different time intervals after sending each packet (both the original and its single retransmission). This modification is aimed at making transmissions look more erratic in the hope of reducing

RITA's timestamp score.

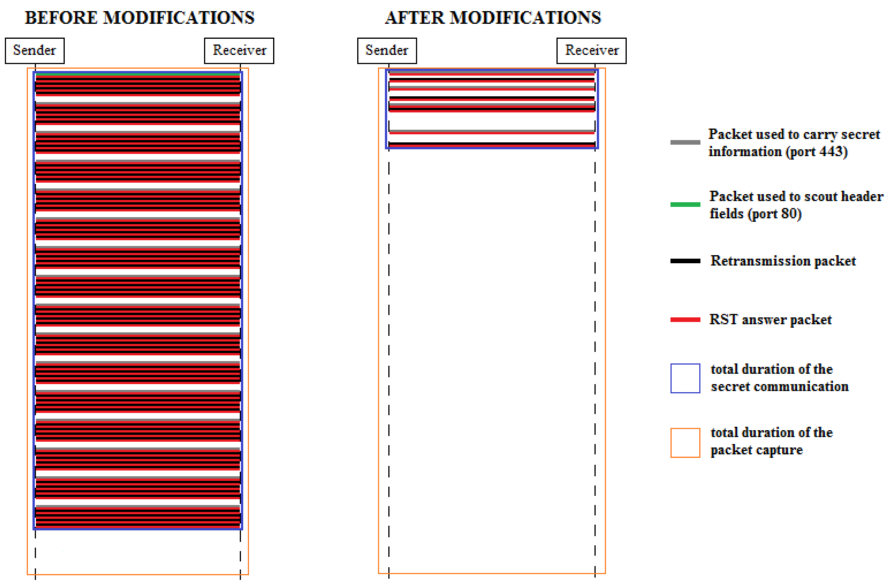

BEFORE MODIFICATIONS
Sender
Receiver
AFTER MODIFICATIONS
Sender
Receiver
Packet used to carry secret information (port 443)
Packet used to scout header fields (port 80)
Retransmission packet
RST answer packet
total duration of the secret communication
total duration of the packet capture

Figure 4.1: Visual representation of the Sender's communication before (left) and after (right) modifications.

Changing the channel's behaviour unfortunately caused it to lose a bit of stealthiness in regards to how much a forged transmission of our own is akin to a real three-way handshake failure in Windows, but fortunately this did not cause neither Zeek nor Suricata to raise any alert in any of the preliminary tests we ran once these modifications were complete. These preliminary tests ultimately produced mixed results. Specifically:

- RITA's duration and histogram scores were consistently reduced as intended: both dropped to almost 0 in all our preliminary tests.
- RITA's timestamp score was on average reduced, but inconsistently: in most cases it fell into the [0.5, 0.6] range but in some cases it remained as high as 0.9.
- If the secret text was small enough to fit in a total number of outgoing SYN packets that did not exceed RITA's grace threshold (taking into account the single retransmission of each packet and how each packet drop slightly increases

this total), our transmission effectively always managed to avoid detection altogether.

Despite the inconsistencies in our attempts to reduce RITA's timestamp score, improvements in the duration and histogram scores alone were enough to ensure the global score would never be higher than at most 0.515 in any of our preliminary tests. We thus decided to conduct one final test aimed at assessing whether our transmission's new score was good enough for it to blend in among other false positives. Due to the timestamp score's inconsistencies we decided to record ten different .pcap files and draw our conclusions from the average score obtained across all of them. Each .pcap file covered a total timespan of 60 minutes and included one secret transmission used by the Sender to exfiltrate an average of 550 bits within a time window of approximatively 220 seconds (always resulting in a duration score of approximatively 0.06 and a histogram score of 0).

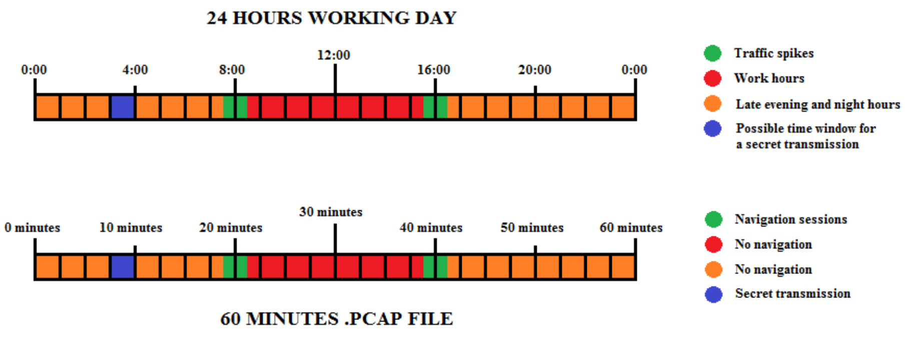


Figure 4.2: Scale comparison between an average working day (above) and one of our 60 minutes .pcap files (below).

Within the 60 minutes timespan we also instructed the compromised host to browse various popular websites in order to recreate as realistically as possible a portion of the typical network traffic that normally generates false positives in RITA's scans. Since in RITA's default configuration a database entry covers a total timespan of twenty-four hours and our .pcap files only cover one single hour, we ultimately scaled down both RITA's grace threshold (lowering it from 20 to 5) and our navigation's frequency to reflect as much as possible the traffic spikes that would normally occur during the average working day. Our navigation ultimately only occurred in two specific moments separated by a 20 minutes timespan between them: this timespan is supposed to represent the eight hour window during which employees are busy working and do not engage in internet navigation (which we assume to occur more frequently at the beginning and at the end of their daily shift). No navigation shall instead occur in the remaining 40 minutes because they are supposed to represent late evening and night hours, during which there is little to no network traffic (refer to Figure 4.2 for a visual comparison). Each navigation session ultimately consisted of visiting (in random order):

- **Google webmail** (to access a mailbox, open two new emails, delete one old email, send one new email).
- **Facebook** (to scroll the news feed for a couple of seconds, open the profile of two friends, check the "photos" section of one of them)
- **New York Times** (to scroll the main page, open two articles to read them)
- **Youtube** (to scroll the main page, visit the "shorts" section, view the first two shorts offered by the feed)

The test results are shown in table 4.1, which includes entries that summarize all recorded network traffic originating from the compromised host. As we can see, our covert channel's score is on average **0.412**, ranging from a minimum of **0.39** (our best performance) to a maximum of **0.512** (our worst performance). There are ultimately always at least a few false positives that score slightly higher compared to our own transmission, even in worst case scenario: they actually become the majority if we compare their score with our average performance instead. In light of this, we feel safe to conclude our covert channel does indeed disguise itself as a false positive relatively well: given how even the traffic generated from a single host produced at least fifteen false positives that scored higher, network traffic generated from a hundred hosts or more would ultimately make it very hard for someone to spot our channel inside the sample cloud unless some sort of filtering is applied preemptively. Unfortunately, we still do not consider this to be enough to automatically guarantee our channel's safety against RITA, but we were at least able to show how adapting the channel's behaviour depending on the specific context can have a positive impact on its stealthiness.

Table 4.1: Average RITA score of our covert channel (in bold) compared to other false positives. Reverse DNS lookup was used to determine each false positive's associated domain and autonomous system from its original IP destination address

Score	**Domain**	**Autonomous system**
0.706	ec2-52-34-182-216.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.7	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.699	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.687	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.656	ec2-35-160-233-103.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.625	a-0003.dc-msedge.net.	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.614	ec2-44-240-204-42.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.613	ec2-52-36-58-117.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.59	ec2-44-235-132-133.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.588	ec2-52-10-247-144.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.572	ec2-35-81-98-90.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.566	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.558	mil04s50-in-f3.1e100.net.	GOOGLE, US
0.521	mil04s51-in-f3.1e100.net.	GOOGLE, US
0.519	ec2-34-208-81-80.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.512	**OUR SECRET TRANSMISSION (worst performance)**	**Not available**
0.512	ec2-54-201-75-72.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.5	Not available	CLOUDFLARENET, US
0.498	Not available	FASTLY, US

Table 4.1: Average RITA score of our covert channel (in bold) compared to other false positives. Reverse DNS lookup was used to determine each false positive's associated domain and autonomous system from its original IP destination address

Score	**Domain**	**Autonomous system**
0.497	edge-video-shv-01-fco2.fbcdn.net.	FACEBOOK, US
0.489	mil04s50-in-f13.1e100.net.	GOOGLE, US
0.485	Not available	FASTLY, US
0.475	ec2-54-189-201-34.us-west-2.compute.amazonaws.com.	AMAZON-02, US
0.472	mil04s43-in-f4.1e100.net.	GOOGLE, US
0.469	mil41s03-in-f10.1e100.net.	GOOGLE, US
0.468	mil04s50-in-f1.1e100.net.	GOOGLE, US
0.468	mil04s50-in-f5.1e100.net.	GOOGLE, US
0.464	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.463	a23-52-250-184.deploy.static.akamaitechnologies.com.	AKAMAI-AS, US
0.46	Not available	LEVEL3, US
0.46	mil04s51-in-f13.1e100.net.	GOOGLE, US
0.45	mil41s04-in-f22.1e100.net.	GOOGLE, US
0.447	mil04s43-in-f1.1e100.net.	GOOGLE, US
0.446	trn06s04-in-f10.1e100.net.	GOOGLE, US
0.446	edge-video-shv-01-mxp2.fbcdn.net.	FACEBOOK, US
0.444	mil04s50-in-f22.1e100.net.	GOOGLE, US
0.443	waw02s05-in-f10.1e100.net.	GOOGLE, US
0.436	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US

Table 4.1: Average RITA score of our covert channel (in bold) compared to other false positives. Reverse DNS lookup was used to determine each false positive's associated domain and autonomous system from its original IP destination address

Score	**Domain**	**Autonomous system**
0.435	mil04s44-in-f10.1e100.net.	GOOGLE, US
0.434	trn05s04-in-f1.1e100.net.	GOOGLE, US
0.434	waw02s05-in-f35.1e100.net.	GOOGLE, US
0.429	mil04s43-in-f10.1e100.net.	GOOGLE, US
0.428	ec2-52-54-49-121.compute-1.amazonaws.com.	AMAZON-AES, US
0.428	trn05s03-in-f3.1e100.net.	GOOGLE, US
0.425	mil41s04-in-f13.1e100.net.	GOOGLE, US
0.423	mil04s51-in-f14.1e100.net.	GOOGLE, US
0.418	ec2-52-3-42-214.compute-1.amazonaws.com.	AMAZON-AES, US
0.417	trn05s03-in-f10.1e100.net.	GOOGLE, US
0.417	mil04s50-in-f14.1e100.net.	GOOGLE, US
0.417	edge-star-shv-01-fco2.facebook.com.	FACEBOOK, US
0.417	mil04s51-in-f5.1e100.net.	GOOGLE, US
0.416	mil04s44-in-f22.1e100.net.	GOOGLE, US
0.413	trn06s03-in-f5.1e100.net.	GOOGLE, US
0.413	mil41s03-in-f14.1e100.net.	GOOGLE, US
0.413	mil04s50-in-f10.1e100.net.	GOOGLE, US
0.412	**OUR SECRET TRANSMISSION (average performance)**	**Not available**
0.412	mil41s04-in-f5.1e100.net.	GOOGLE, US

Table 4.1: Average RITA score of our covert channel (in bold) compared to other false positives. Reverse DNS lookup was used to determine each false positive's associated domain and autonomous system from its original IP destination address

Score	Domain	Autonomous system
0.41	trn05s04-in-f3.1e100.net.	GOOGLE, US
0.409	mil04s44-in-f3.1e100.net.	GOOGLE, US
0.407	edge-star-shv-01-mxp2.facebook.com	FACEBOOK, US
0.407	mil41s04-in-f10.1e100.net.	GOOGLE, US
0.405	trn06s03-in-f14.1e100.net.	GOOGLE, US
0.405	mil04s51-in-f10.1e100.net.	GOOGLE, US
0.402	mil04s44-in-f14.1e100.net.	GOOGLE, US
0.399	edge-dgw-shv-01-mxp2.facebook.com.	FACEBOOK, US
0.399	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.399	edge-dgw-shv-01-fco2.facebook.com.	FACEBOOK, US
0.39	**OUR SECRET TRANSMISSION (best performance)**	**Not available**
0.387	trn06s04-in-f3.1e100.net.	GOOGLE, US
0.384	mil04s44-in-f1.1e100.net.	GOOGLE, US
0.383	ec2-44-211-112-71.compute-1.amazonaws.com.	AMAZON-AES, US
0.382	a2-19-124-207.deploy.static.akamaitechnologies.com.	AKAMAI-ASN1, NL
0.379	mil41s04-in-f14.1e100.net.	GOOGLE, US
0.37	mil04s43-in-f5.1e100.net.	GOOGLE, US
0.363	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.362	mil04s43-in-f14.1e100.net.	GOOGLE, US

Table 4.1: Average RITA score of our covert channel (in bold) compared to other false positives. Reverse DNS lookup was used to determine each false positive's associated domain and autonomous system from its original IP destination address

Score	**Domain**	**Autonomous system**
0.354	trn05s04-in-f10.1e100.net.	GOOGLE, US
0.353	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.35	mil41s03-in-f22.1e100.net.	GOOGLE, US
0.342	a2-19-124-200.deploy.static.akamaitechnologies.com.	AKAMAI-ASN1, NL
0.339	xx-fbcdn-shv-01-mxp2.fbcdn.net.	FACEBOOK, US
0.332	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.331	xx-fbcdn-shv-01-fco2.fbcdn.net.	FACEBOOK, US
0.322	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.319	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.313	edge-star-shv-01-mxp1.facebook.com.	FACEBOOK, US
0.311	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US
0.303	vip0x008.map2.ssl.hwcdn.net.	STACKPATH-CDN, US
0.301	Not available	MICROSOFT-CORP-MSN-AS-BLOCK, US

Chapter 5

Conclusions

When we first began our research we did so with the specific intent of creating a covert channel compatible with Windows that could offer a relatively good compromise between stealthiness, bandwidth and robustness. We also wanted our channel to be as versatile as possible, which is why our steganography method only interacts with layer 3 and layer 4 of the OSI reference model.

We feel overall satisfied with how much forged packets created by our program are akin to real ones generated by the TCP/IP stack of Windows, though we could have achieved even better results by reverse engineering the *tcpip.sys* driver itself in order to create our own slightly modified "duplicate" of it. We chose however not to follow this path both because of constraints related to time and tools available to us and because doing so would have implied violating proprietary software without permission.

Speaking of the channel's performance in regards to stealthiness, robustness and bandwidth we feel relatively satisfied but at the same time also learned some hard lessons along the way. Upon defining our own steganography method we mainly used storage based existing methods as a source of inspiration due to the higher bandwidth they offer on average. In retrospect, we unfortunately cannot help but feel that as time progresses, storage based channels will have increasingly more problems operating due to how efficient active wardens have become at altering header fields, even when a field is quite vital (such as TCP's initial sequence number): the most relevant example in this sense are firewalls such as Cisco ASA.

At the same time however, we do not believe following a different approach altogether (i.e. timing based or hybrid) would have necessarily produced significant improvements. Our experience with RITA is especially relevant in this sense, as in order to adapt our channel we ultimately had to alter its transmission pattern: such a modification would have been inherently a lot less compatible with a timing based channel. Speaking more generally, we believe the more a covert channel relies on timing as part of its strategy, the more vulnerable it becomes to statistical analysis and passive wardens. As time progresses we thus expect timing based channels to struggle increasingly as well, mainly due to the significant breakthroughs that are currently being made in the field of artificial intelligence (which we believe will become a tool very effective at improving the accuracy of passive wardens).

5.1 Future work

For each different network steganography approach or method it is ultimately always possible to conceive a countermeasure specifically tailored to neutralize it, confirming once again that one single general purpose solution capable of operating in every context and situation simply does not exist. Such a limitation however also affects to an extent the defender's options: in our experience the most effective tools were also the most specific, ultimately implying a truly tight and robust defense can only be achieved through the careful and capillary configuration of multiple security tools.

In light of this, we believe a good development direction for future covert channels could be to invest in redundancy in order to improve their ability to adapt in each specific context: rather than rely on one single steganography method, a channel of this kind could switch between several possible alternatives depending on information learned on the field. Naturally, such a channel is still not guaranteed to bypass the defender's countermeasures but has two advantages compared to a channel that uses just one single steganography method. The first advantage is that it causes the defender's costs to increase, since securing the network against multiple exfiltration techniques likely requires the deployment of additional tools (or at the very least a more capillary configuration of existing ones). The second advantage is that since the defender's set up becomes more complex, the possibility of a human error also increases: in this sense, if even just one exfiltration technique is successful, the attacker ultimately wins.

An adaptation focused approach should ultimately also be adopted in future versions of our own covert channel in order to improve its ability to identify different situations in which transmission fails, their potential causes and its most appropriate reaction to each. Some features would be relatively easy to add: for example, the channel could be instructed to transmit using a different destination port if using 443 turns out to be forbidden on the compromised host's network. Instructing the channel to change its steganography method altogether would be a bit more tricky, but the channel's ability to notice external alterations to sequence number values already provides a very good starting point for future developments. With a view to the possibility of enhancing our channel with additional exfiltration methods, we will conclude our presentation by providing a high-level illustration of some other alternatives to our method we had at first taken into account but then temporarily put aside for various reasons (mostly bandwidth related).

One first alternative consists in a storage based method revolving around modulation of the IP destination address. In this context, the attacker must possess multiple public IP addresses, each one corresponding to a different symbol: the Covert Sender ultimately transfers information by contacting the destination that matches the symbol it wants to transmit. For example, assume the attacker controls address A (representing 0) and address B (representing 1): in order to transmit the string 10010011 the Covert Sender shall make contact with B, A, A, B, A, A, B, B in this exact order.

Another alternative consists in a timing based method revolving around cyclic transmission intervals determined with the use of modular arithmetic (refer to Figure 5.1 for a visual representation). In this context, we imagine each 20 seconds time window to be divided into four 5 seconds intervals, each representing a different 2-bit symbol (00, 01, 10, 11 respectively). Assuming "T" is the result of Python's **time.time()** function, at any given moment the Sender can compute T modulo 20 to determine in which of the four time intervals it is located. If this interval matches the next symbol to send, the Sender can ultimately proceed to transmit a packet to the Receiver. Otherwise, it shall first wait an appropriate amount of time in order to cycle through the other intervals and reach the correct one. Upon receiving a packet, the Receiver shall also compute T modulo 20 to determine in which time interval the transmission occurred, ultimately deducing the corresponding symbol (in this example each interval has a duration of 5 seconds to account for possible latency issues).

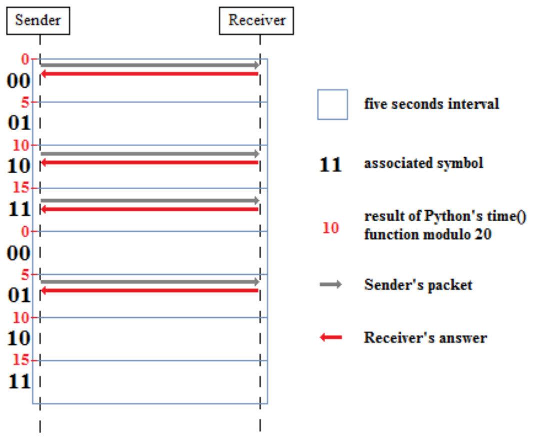


Figure 5.1: Visual representation of our timing based alternative method. In this example, the Sender uses four packets to transmit the string 00101101.

One final alternative consists in trying to hide the secret message in the traceroute option of the IP header. This steganography method (originally introduced by Trabelsi and Jawhar [22]) has been already illustrated in Section 2.2.2, at the end of which we also mentioned the risks associated with using it. Nevertheless, we cannot help but appreciate how much bandwidth it offers: if by any chance a transmission using this method turns out to be successful, one single packet has enough capacity to exfiltrate an entire 256-bit key alone (which would significantly enhance the channel's ability to evade statistical analysis).

Bibliography

[1] Punam Bedi and Arti Dua. Network steganography using the overflow field of timestamp option in an ipv4 packet. *Procedia Computer Science*, 2020.

[2] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP covert timing channels: design and detection. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick D. McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 178–187. ACM, 2004. doi: 10.1145/1030083.1030108. URL `https://doi.org/10.1145/1030083.1030108`.

[3] Wesley M. Eddy. Transmission control protocol (TCP). *RFC*, 9293:1–98, 2022. doi: 10.17487/RFC9293. URL `https://doi.org/10.17487/RFC9293`.

[4] Abduhalil Ganivev, Obid Mavlonov, Baxtiyor Turdibekov, and Ma'mura Uzoqova. Improving data hiding methods in network steganography based on packet header manipulation. In *2021 International Conference on Information Science and Communications Technologies (ICISCT), Tashkent, Uzbekistan, November 03-05, 2021*, 2021. doi: 10.1109/ICISCT52966.2021.9670109. URL `https://doi.org/10.1109/ICISCT52966.2021.9670109`.

[5] John Giffin, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts. Covert messaging through TCP timestamps. In Roger Dingledine and Paul F. Syverson, editors, *Privacy Enhancing Technologies, Second International Workshop, PET 2002, San Francisco, CA, USA, April 14-15, 2002, Revised Papers*, volume 2482 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2002. doi: 10.1007/3-540-36467-6_15. URL `https://doi.org/10.1007/3-540-36467-6_15`.

[6] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In Dan S. Wallach, editor, *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. USENIX, 2001. URL `http://www.usenix.org/publications/library/proceedings/sec01/handley.html`.

[7] Drew Hintz. Covert channels in tcp and ip headers. In *DefCon 10, Las Vegas, Nevada, August 2-4, 2003*, 2003.

[8] Félix Iglesias Vázquez, Fares Meghdouri, Robert Annessi, and Tanja Zseby. Ccgen: Injecting covert channels into network traffic. *Security and Communication Networks*, 2022, 05 2022. doi: 10.1155/2022/2254959.

[9] Van Jacobson, Robert T. Braden, and David A. Borman. TCP extensions for high performance. *RFC*, 1323:1–37, 1992. doi: 10.17487/RFC1323. URL `https://doi.org/10.17487/RFC1323`.

[10] Amit Klein. Subverting stateful firewalls with protocol states (extended version). *CoRR*, abs/2112.09604, 2021. URL `https://arxiv.org/abs/2112.09604`.

[11] Deepa Kundur and Kamran Ahsan. Practical internet steganography: Data hiding in ip. 04 2003.

[12] Józef Lubacz, Wojciech Mazurczyk, and Krzysztof Szczypiorski. Principles and overview of network steganography. *IEEE Commun. Mag.*, 52(5):225–229, 2014. doi: 10.1109/MCOM.2014.6815916. URL `https://doi.org/10.1109/MCOM.2014.6815916`.

[13] Xiapu Luo, Edmond W. W. Chan, and Rocky K. C. Chang. TCP covert timing channels: Design and detection. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, pages 420–429. IEEE Computer Society, 2008. doi: 10.1109/DSN.2008.4630112. URL `https://doi.org/10.1109/DSN.2008.4630112`.

[14] Wojciech Mazurczyk and Krzysztof Szczypiorski. Evaluation of steganographic methods for oversized IP packets. *Telecommun. Syst.*, 49(2):207–217,

2012. doi: 10.1007/s11235-010-9362-7. URL https://doi.org/10.1007/s11235-010-9362-7.

[15] Aleksandra Mileva and Boris Panajotov. Covert channels in TCP/IP protocol stack - extended version-. *Central Eur. J. Comput. Sci.*, 4(2):45–66, 2014. doi: 10.2478/s13537-014-0205-6. URL https://doi.org/10.2478/s13537-014-0205-6.

[16] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into TCP/IP. In Mauro Barni, Jordi Herrera-Joancomartí, Stefan Katzenbeisser, and Fernando Pérez-González, editors, *Information Hiding, 7th International Workshop, IH 2005, Barcelona, Spain, June 6-8, 2005, Revised Selected Papers*, volume 3727 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2005. doi: 10.1007/11558859_19. URL https://doi.org/10.1007/11558859_19.

[17] Jon Postel. Internet protocol. *RFC*, 791:1–51, 1981. doi: 10.17487/RFC0791. URL https://doi.org/10.17487/RFC0791.

[18] Craig H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2(5), 1997. doi: 10.5210/fm.v2i5.528. URL https://doi.org/10.5210/fm.v2i5.528.

[19] Kartik Sharma and Akshay Sharma. High bandwidth covert channel using tcp-ip packet header. 02 2016.

[20] Gustavus J. Simmons. The prisoners' problem and the subliminal channel. In David Chaum, editor, *Advances in Cryptology, Proceedings of CRYPTO '83, Santa Barbara, California, USA, August 21-24, 1983*, pages 51–67. Plenum Press, New York, 1983.

[21] Franco Tommasi, Christian Catalano, Alessandro Caniglia, and Ivan Taurino. Cotiip: a new covert channel based on incomplete ip packets. In *2022 7th International Conference on Smart and Sustainable Technologies (SpliTech), Split / Bol, Croatia, July 05-08, 2022*, 2022. doi: 10.23919/SpliTech55088.2022.9854307. URL https://doi.org/10.23919/SpliTech55088.2022.9854307.

[22] Zouheir Trabelsi and Imad Jawhar. Covert file transfer protocol based on the ip record route option. *Journal of Information Assurance and Security (JIAS)*, 5, 01 2010.

[23] Steffen Wendzel, Luca Caviglione, Wojciech Mazurczyk, Aleksandra Mileva, Jana Dittmann, Christian Krätzer, Kevin Lamshöft, Claus Vielhauer, Laura Hartmann, Jörg Keller, and Tom Neubert. A revised taxonomy of steganography embedding patterns. In Delphine Reinhardt and Tilo Müller, editors, *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, pages 67:1–67:12. ACM, 2021. doi: 10.1145/3465481.3470069. URL `https://doi.org/10.1145/3465481.3470069`.

[24] Sebastian Zander, Grenville Armitage, and Philip Branch. Covert channels in the ip time to live field. 01 2007.