International Journal of Information Security (2024) 23:3481–3512
https://doi.org/10.1007/s10207-024-00879-z

REGULAR CONTRIBUTION

A TCP-based covert channel with integrity check and retransmission

Stefano Bistarelli[1] · Andrea Imparato[1] · Francesco Santini[1]

Published online: 12 August 2024

Abstract
We propose a covert channel and its implementation in Windows OS. This storage channel uses the *Initial Sequence Number* of TCP to hide four characters of text and the *identification* field to "sign" the message and thus understand if it has been altered during the transmission. The secret is sent in the first SYN segment to open a connection, and an ACK-RST response acknowledges the receipt. Designed error-correction codes make the protocol more robust and able to handle (IP) packet drops and transmission errors. In this paper, we provide a detailed discussion of the implementation and an evaluation of the stealthiness of the proposed channel: we inspect the generated traffic with two IDSs and RITA, a tool performing statistical analysis to detect malware beaconing.

Keywords Network steganography · Covert channel · Robustness

1 Introduction

Steganography comprises the science and art of hiding information transfer and storage [28]. Steganography is not to be confused with cryptography: they both share the ultimate goal of protecting information, but the former attempts to hide it to make it "difficult to notice". At the same time, the latter tries to scramble it to make it "difficult to read" [17]. Hence, cryptography is defeated if someone successfully reads the communication's content, while steganography is defeated when someone notices the information exchange itself.

Steganography predates the development of complex machines and exists independently of computer science. However, the arrival of computers and the internet introduced a plethora of new domains where steganography can be applied. Examples include hiding information in file systems or digital media such as JPEG images, MP3 audio and video streams. Generally speaking, a covert transfer of information always features the following elements regardless of its specific domain [28]:

- *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 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 [28].

Many existing works call this type of communication a covert channel, referencing a concept first introduced by Lampson in 1973 [16]: according to Lampson, a covert channel is any channel "not intended for information transfer at all". The US Department of Defense provided an alternative definition in 1985 [26]: a covert channel is "any communication channel that a process can exploit to transfer information in a manner that violates the system's security policy. This definition emphasises well both the benefits and the dangers associated with the ever-increasing use of network steganog-

✉ Francesco Santini
francesco.santini@unipg.it

Stefano Bistarelli
stefano.bistarelli@unipg.it

Andrea Imparato
andrea.imparato@studenti.unipg.it

1 Dipartimento di Matematica e Informatica, University of Perugia, Perugia, Italy

raphy 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 this paper, we design a covert channel by using TCP fields to embed a message in the protocol header of a TCP segment.[1] Each SYN packet used to open a connection secretly carries four characters of text: longer messages can be sent by opening multiple connections. More in detail, we apply network steganography by embedding the message into the *Sequence Number* field[2] of TCP. The paper's novelty resides in the channel's robustness and the integrity enforced on data. Some error codes are shared as well and used to retransmit the message in case of loss (we use SYN packets to establish a new connection to send a piece of message, so we do not inherit the robustness of TCP ourselves), and we "sign" the message by applying a random function on the id field of the TCP header and XORing it with the message. The purpose is to hide the message better (for example, vowels would have a higher frequency of appearance) and also have a way to check if the message has been altered during the transmission.

Since the channel only needs to interact with TCP/IP headers to transmit data, it can be adapted to function on any underlying OS. However, the implementation we used to carry out our experiments was designed to run on Windows OS specifically, as it mocks the behaviour of this specific TCP/IP stack and exploits Windows commands to retrieve vital information to establish the channel. The generated traffic, in the form of *pcap* files (i.e., packet capture), is passed to the inspection of two *Intrusion Detection Systems* (*IDS*) and *RITA* (*Real Intelligence Threat Analytics*: RITA is an open-source framework for detecting command and control communication through network traffic analysis. The *Beacon Detection* module of RITA identifies signs of beaconing behaviour in and out of a network. The covert channel we propose could produce regular transmission in dimension (normal packet size) and time (regular interval between sending different chunks of the same message). For this reason, we adapt the channel to reduce the attention-score RITA assigns to it and make it appear as a false positive together with other completely "legit" internet sessions. As far as we know, RITA is used for the first time to perform statistical analysis over a covert channel.

We want to underline the advantages of network steganography concerning other *data obfuscation* techniques, such as data encryption.[3] Network steganography hides the fact that communication is taking place; in contrast, encryption only protects the content of the communication: this makes it less likely for an unauthorised party even to be aware that a secret exchange is occurring. Moreover, steganography can be effective against traffic analysis, as the hidden information does not follow recognisable patterns that might be detected through monitoring network traffic; in contrast, encrypted traffic may still reveal patterns that could be subject to analysis.[4] Data encryption might also be susceptible to cryptanalysis and known-plaintext attacks and require accurate and secure key management. Finally, even when data is securely encrypted, it may attract attention due to its encrypted nature; steganography, on the other hand, does not leave a decryption footprint, as the hidden information is not immediately apparent. Even if steganography has these advantages in specific scenarios, it also has disadvantages (for example, a cost in terms of longer connections or increased resource usage [19]), and it is not a substitute for encryption. The two can be complementary, and in many cases, a layered approach that combines encryption with steganography may provide a more robust solution for securing communication.

Moreover, we would also highlight the differences between steganography and enforcing communication anonymity, for example, through high-latency systems as *mix networks* [3] or low-latency ones as *Onion Routing* and *Tor* [4].[5] According to [22], communication anonymity prevents an observer from linking a message to a sender/receiver, while steganography does not. Moreover, systems implementing anonymous communications usually employ many additional resources, such as several hops in an overlay network using encryption of messages between hops. With encryption, we fall into the same considerations provided above.

Network steganography can be employed in various legitimate and potentially malicious applications. Some examples of possible applications include data exfiltration in espionage and Intelligence, confidential business communication, malicious activities (e.g., malware communication or command-and-control traffic to evade IDSs), anonymous communication and anti-censorship, and copyright protec-

[1] Even if the word "segment" would correspond to a more precise terminology when referring to the *Protocol Data Unit* of TCP, in the paper, we will also use a more general "packet".

[2] Or *Initial Sequence Number*, since we always use SYN packets to transmit the message.

[3] We stick to the definition about *obscured data* in NIST-CSRC glossary: "data that has been distorted by cryptographic or other means to hide information. It is also referred to as being masked or obfuscated".

[4] For example, some encryption protocols may produce fixed-length ciphertext, such as block-ciphers as *AES* (i.e., *Advanced Encryption Standard*), or have distinctive headers, such as *AES-GCM* (*Galois/Counter Mode*), where a 96-bit nonce is included as part of the ciphertext.

[5] We mean the latency in the delivery of the message from the sender to the receiver: in case of web browsing, for example, latency needs to be low to provide good enough user experience.

tion (e.g., embedding information to identify the origin or ownership of digital content).

After this introductory section, Sect. 2 surveys the most important properties and some examples of TCP channels proposed in the literature. Section 3 describes the hypotheses we suppose for our covert channel to come into play. Section 4 introduces our covert channel by motivating our choices and illustrating the communication protocol, while Sect. 5 illustrates the most critical steps of the implementation process, particularly those aimed at creating forged packets that need to look as realistic as possible to a warden in a Windows system. Section 6 describes how we improved the robustness of the communication in case of message error or drop. In Sect. 7, we test the implementation and the stealthiness of the covert channel by successfully bypassing two different IDS (*Suricata* and *Zeek*). Section 8 shows how to make our channel survive statistical analysis of "beaconing" detection tools as RITA: our Sender may frequently beacon the Receiver to send a message, thus appearing as malware. Finally, Sect. 9 presents conclusive thoughts and ideas about future work.

2 Covert channels and their properties with a focus on IP and TCP

One of the pioneering works on contemporary information hiding is due to the prisoner's problem described in [25], which 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 to escape. They can exchange written messages, but a warden constantly watches their communications to ensure they are not exchanging notes for malicious purposes. To hamper their plan, the warden could either analyse 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 prison; they might as well be the same person (for example someone who installed an application on a particular host that needs to send data to another one covertly), and the warden consists in any hardware or software guarding the network (for example, a firewall).

Covert channels need to show some good properties. The three most recurring concepts found in the literature are:

– *Bandwidth* expresses a channel's transmission capacity [17], usually measured either in bits per second (Raw Bit Rate, RBR) or bits per packet (Packet Raw Bit Rate, PRBR) [20]. In the following sections of this paper, we shall focus mainly on channels that operate at levels 3 and 4 of the OSI reference model, meaning by "packets" we refer to TCP/IP packets specifically.
– *Stealthiness* expresses how successfully a channel evades the warden's attention. This property is complex to evaluate objectively because it depends on how sophisticated a given steganography method is and how effective the warden's countermeasures are [17]. Regarding this last point, Iglesias et al. argued that while researchers have already conceived many techniques that could successfully detect network steganography, several existing security tools have not implemented them just yet [12].
– *Robustness* expresses how much alteration a channel can withstand without compromising the flow of secret information [17]. Much like stealthiness, this property is also complex to evaluate objectively because of the warden's potential impact. Generally speaking, the more robust a channel is, the less frequently errors and information loss occur.

While a good covert channel should strive to maximise its performance regarding all three of these properties, in reality, they are interdependent. For example, the bandwidth increase usually comes at the cost of reducing robustness and stealthiness [17]. Finally, many works classify covert channels into three comprehensive categories depending on their general communication strategy. Specifically [8]:

– *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 combines both.

On average, storage channels tend to have higher embedding capacity than timing and hybrid channels but also less robustness and stealthiness. On the other hand, channels based on timing require synchronisation on the Covert Receiver's side and are usually more challenging to implement [20].

In the following two subsections, we will revise some works in the literature by focusing on steganography techniques that exploit different fields in the IP and TCP headers (in Sects. 2.1, 2.2, respectively) since for our channel, we are interested in these two protocols (our motivations will be presented in Sect. 3).

2.1 Steganography in the IP header

Among existing works, several proposed embedding a secret message in the Identification field of the IP header (Internet Protocol). This 16-bit field contains a value initialised during packet creation that is unique for each packet and allows the correct identification of fragments belonging to the same packet whenever fragmentation occurs. Since the field can theoretically be set to any value, Rowland suggested that a Covert Sender could hide a single ASCII ("American Standard Code for Information Interchange") character by embedding it in the 8 most significant bits, assigning a random value to the rest 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 [23].

Murdoch and Lewis [21] argued that values usually assigned to the Identification field are not as random as they may seem. The only constraint contemplated by operating systems upon assigning new values consists of 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 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.

In light of this, Tommasi et al. [26] suggest a different strategy in which the Covert Sender does not directly manipulate the Identification field of a single packet but instead sends a continuous 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 the 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 customarily 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 contains secret information. This information consists of a single character belonging to a previously arranged alphabet. To extract it, the Covert Receiver has to use the modulo operator with the alphabet's size: in the 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. However, the abnormally frequent use of the MF bit and the absence of following fragments could alert it and cause it to block suspicious packets. The authors thus introduced an additional safety measure consisting of 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 can still deduce its embedded data, 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 the Covert Receiver misinterprets as secret information: an example could be a packet that was dropped simply because its TTL (Time to Live) had reached a value of zero. The bandwidth of this solution is ultimately challenging to estimate because it depends on how often the Identification field's value progression produces packets fit for carrying secret information.

Instead, Kartik Umesh Sharma [24] proposed encoding 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. 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 of four ASCII characters. Otherwise, it is just a regular packet. The four bytes of the hidden 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.

2.1.1 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 suggests, this field typically includes additional helpful information in certain situations but is generally not required for the protocol to operate. The main advantage of using this field to implement a covert channel is the 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 [21]. Moreover, since altering the field's content does not affect the protocol's correct functioning, an active warden could clear the field to prevent its possible misuse.

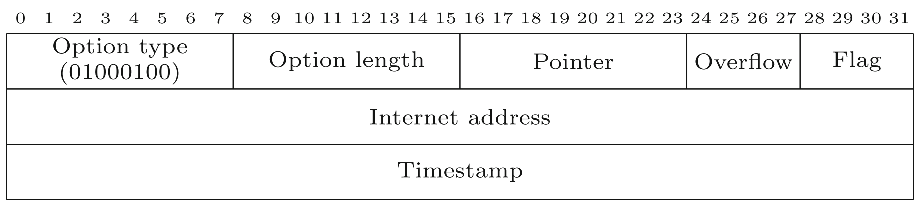


Fig. 1 Structure of the IP header's Timestamp option [1]

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 Fig. 1, 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.
- Pointer (8 bits) to keep track of the following available location for a new entry. No space is left if its value surpasses the Option Length's value.
- Overflow (4 bits), to track how many nodes could not 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 Fig. 2 for an example). If any node fails to add a new entry because no space is left, it must increase by one the Overflow counter. This field is typically initialised with a value of 0 and is precisely 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. However, there is the risk of a real overflow overwriting embedded data on the Internet. 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 [21]. 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 space is still available.

Trabelsi and Jawhar opted instead to implement a covert channel using the Traceroute option [27]. 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 Fig. 3 and is relatively similar to the one used by the Timestamp option in that the first three fields are used similarly. The most notable difference is the absence of the Overflow and Flag fields because 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.

In [27], 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. Suppose the Pointer field is initialised with a value greater than the Option length field. In that case, no node can add any entry, 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 become redundant and can be used to expand the covert channel's bandwidth even more. Since the authors implemented two-way communication, they chose to reserve these bits for traffic control functionalities similar to the ones used in TCP to improve robustness. Compared to the Timestamp option, this solution is more robust and offers more bandwidth but is still prone to quickly drawing the warden's attention. Specifically, the warden could realise 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 travelled by the packet (especially if the warden is very close to the Covert Sender).

2.1.2 Steganography in the fragmentation process

Fragmentation is a process that occurs whenever an IP packet encounters a segment along the path that lacks a *Maximum Transmission Unit* (*MTU*) big enough to carry the packet as a whole: as a consequence, the packet is split into smaller parts (called Fragments) that are individually sent along the segment and later reassembled by the receiver. Table 1 shows an example of how an IP packet is fragmented. Every sequence fragment 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 determines which part of the original packet is carried by each fragment. All

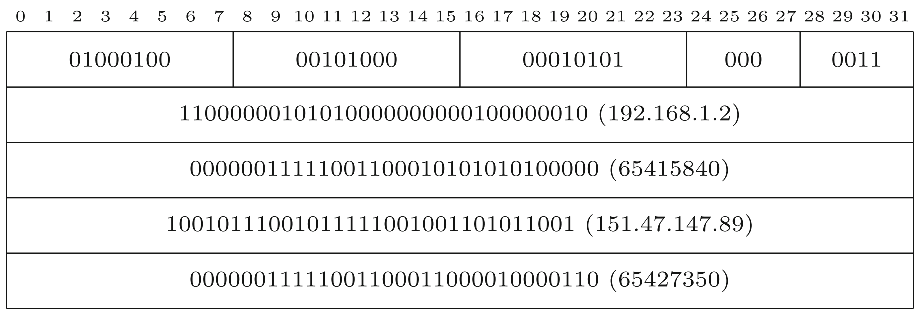


Fig. 2 Example of IP Timestamp option with Flag field set to 3 and two entries added (each consisting of IP address followed by timestamp value)

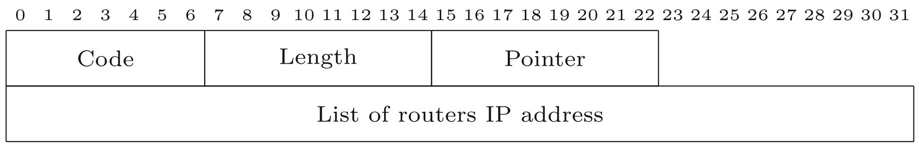


Fig. 3 Structure of the IP header's Traceroute option [27]

the fragments of a given packet share its Identification field value so that they can be easily distinguished from fragments of other packets.

Mazurczyk and Szczypiorski [18] proposed various steganography methods based on forging fragment sequences that, at first glance, look like the mere product of a natural fragmentation process but carry hidden data:

- The first method conveys information through the total number of fragments created: an even number represents a 0, and an odd number represents a 1, for an unlimited bandwidth of 1 bit per packet.
- The second method consists of embedding information in the Fragment offset of each fragment: once again, even values represent 0 and odd values represent 1. This method's bandwidth ultimately depends on how many fragments are created, but since, as shown in Fig. 1, 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 more challenging to spot.
- The third method consists of injecting in an authentic sequence of fragments some forged ones containing secret information in their payload instead of parts of the original packet. 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 and 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 Fig. 1, Fragment offset values belonging to the 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 alter a sequence by deleting a fragment to represent a 1 or leaving it intact to represent a 0, resulting in a total bandwidth of 1 bit per packet.

To defeat these steganography methods, a passive warden can use statistical analysis to detect anomalies in the fragmentation process. An even better solution is deploying an active warden that reassembles the fragments and randomly splits them again to jam any covert information. According to Klein [14], 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.

2.1.3 Steganography in other fields of the IP header

IP packets can also be labelled to prevent fragmentation by setting their DF (Don't Fragment) bit to 1. Kundur and Ahsan exploited this aspect to develop yet another covert channel [15]. In this case, the embedded information consists of the exact value the Covert Sender chose for the DF bit. Naturally, this also implies the Covert Sender must create packets small enough to fit the MTU of all segments along the path, lest the impossibility of fragmenting any packet with DF set to 1 makes it impossible to deliver. Such a channel is easy to implement overall 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 [21].

Table 1 Example of IP fragmentation [18]

Original packet	Sequence	Identifier	Totale length	DF	MF	Fragment offset
	0	345	5140	0	0	0
Fragment 1	0–0	345	1500	0	1	0
Fragment 2	0–1	345	1500	0	1	185
Fragment 3	0–2	345	1500	0	1	370
Fragment 4	0–3	345	740	0	0	555

Another steganography method analogous to embedding information in the DF bit is to embed it in the Delay bit, as suggested by Hintz in [11]. The Delay bit generally belongs to the ToS (Type of Service) field, which was initially intended to carry additional information related to the quality of service parameters but is rarely used with its original semantic nowadays. 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 [21]. 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 [29]. This field is of fundamental importance in regular network traffic because it prevents congestion caused by packets trapped in infinite routing loops. The original sender of any packet must thus always initialise it with a specific value, and every node that receives and retransmits the packet decreases it by one. If the value ever drops to zero, the packet is deleted without retransmission, ensuring it will only travel for a limited number of hops. Naturally, this means the field must always be initialised 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 difficult because its value is purposefully designed to be altered upon each hop, resulting in a high risk of information loss. Since the field can be initialised 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 initialising the TTL field. The authors thus analysed existing network traffic to discover what changes to TTL values could effectively blend in well enough to avoid detection. Their analysis ultimately suggested that a good covert channel should use at most two different TTL values immediately adjacent to each other and avoid changing values more often than once for every 2–3 packet pairs. An effective strategy to defeat such a channel would be to deploy an active warden that normalises the TTL value of all transiting packets [20].

Table 2 summarises the proposal of network steganography in the IP header by showing their advantages and disadvantages regarding bandwidth, stealthiness and robustness.

2.2 Steganography in the TCP header

The TCP header (Transmission Control Protocol) 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 with the URG flag to signal the presence of urgent data in the payload [6]. Since this field is rarely used nowadays, Hintz suggested using it to develop a covert channel [11]. Such a channel would be easy to implement and grant a total bandwidth of 16 bits per packet. Still, Hintz recognised 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 clear the field's value without consequences) [11].

Giffin et al. [9] 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. The Options field of the TCP header is generally similar to the one already seen in the IP header. It once again contains information that can be occasionally useful but is typically unnecessary for the protocol to operate. In TCP, the Timestamp option measures the time gap between a packet's departure from a sending host and its arrival at the receiving one [13]. To do so, the sending host appends its local clock's timestamp in the 32 bit TS value field shown in Fig. 4. The receiving host then copies this information in the acknowledgement packet to confirm the original one's arrival and adds its timestamp in the TS echo reply field before sending it.

The authors suggested modulating the least significant bit of the sender's timestamp to convey secret information for a total bandwidth of bit per packet. Since timestamp values must be strictly monotonic [21] this modulation must always be done through an increment: the authors thus recommended

Table 2 Summary of some IP-based steganography methods (**A** and **D** stand for "Advantage" and "Disadvantage" respectively)

Method	Bandwidth	Stealthiness	Robustness
IP identification modulation (1997) [23]	8 bits per packet	**A** Warden cannot clear the field	No relevant features
		D Warden can detect the channel through statistical analysis	
DF bit modulation (2003) [15]	1 bit per packet	**D** Warden can notice the field's alteration by looking at the context	No relevant features
Delay bit modulation (2003) [11]	1 bit per packet	**D** Warden can clear the field	No relevant features
		D Warden can detect the channel through statistical analysis	
TTL modulation (2007) [29]	1 bit per packet	**A** Based on a vital field	**D** Packet travelling alters the field (risk of information loss)
		A Can be hard to spot (depending on implementation)	
		D Warden can reset the field to a default value	
IP traceroute option modulation (2010) [27]	Up to 296 bits per packet	**D** Warden can clear the field	**A** Thanks to its large capacity, some bits can be used to implement a retransmission feature
		D Warden can notice the field's inconsistency	
Fragmentation based methods (2012)[18]	Changes with each specific method (often 1 bit per packet)	**D** Fragmentation is rarely used	**D** Limited range due to devices such as CGNATs
		D Warden can destroy the message by reassembling the fragments	
Using IP identification as a pointer (2016) [24]	Up to 32 bits per packet	**A** Does not alter header fields	No relevant features
		D Alters the payload	
IP timestamp option modulation (2020) [1]	4 bits per packet	**D** Warden can clear the field	**D** Limited range due to how the IP timestamp option works
		D Warden can notice the field's inconsistency	
COTIIP (2022) [26]	Difficult to estimate (affected by alphabet size and message structure)	-**D** Sender is forced to transmit a continuous stream of packets	**A** Can resist certain forms of packet filtering
		D Use of the MF bit could alert the warden	**D** Receiver misinterprets normal packet drops as relevant information

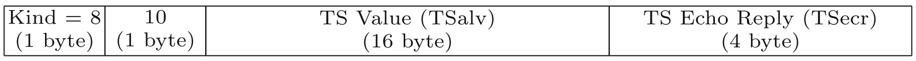


Fig. 4 Structure of the IP header's Traceroute option [27]

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 be expensive. Moreover, including the Timestamp option in many TCP packets is the norm in some Linux versions and other Unix-like operating systems [21], 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 measuring the total ratio to different timestamps or applying a complex randomness test to the least significant bits, depending on connerecognisedd [11].

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 [21]. The sending host decides the field's initial value and can theoretically be any number ranging from 0 to $2^{32} - 1$. The main advantage of any steganography method that modulates this field is that the warden cannot prohibit its use or arbitrarily alter its value without affecting regular network traffic. An active warden's best shot at neutralising such a covert channel would be to proxy all outgoing TCP connections so that ISN values are always overwritten. Still, Hintz argued doing so would be difficult, possibly hinting at the resulting overhead negatively impacting network performance [11].

In [23], 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 of reserving some bits (usually the most significant ones) to encode an ASCII character and randomising the rest to create more realistic value distributions. The resulting bandwidth would be 8 bits per TCP connection.

Still in [23], 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, allowing them to synchronise on the counterpart's initial sequence number [6]. 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. To continue the handshake, the recipient replies with a packet containing its initial sequence number, both the SYN and ACK flags set, 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 of 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 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 containing the Covert Sender's ISN increased by one in the Acknowledgement number field. Therefore, the Covert Receiver can read this packet's Acknowledgement number value, decrease it neutralising and ultimately extract the secret information embedded initially by the Covert Sender [23]. The most relevant aspect of this strategy is that having a legitimate server "bounce" secret information lets the Covert Sender bypass a typical limitation of the warden, allowing outgoing connections only towards certain specific destination addresses. However, the warden could 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 randomising to apply *SAV (Source Address Validation)* as a default policy [14].

Ganivev et al. [8] created their variant of Rowland's original strategy using the TCP header's ISN field and the IP header's Identification field simultaneously. Their method consists of 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.

Upon considering the alleged randomness of ISN values, Murdoch and Lewis [21] 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 considers the ISN generation policies contemplated in operating systems such as Linux and OpenBSD to produce field values that look as realistic as possible while still carrying hidden information.

Table 3 summarises the proposals above of network steganography in the TCP header by showing their advantages and disadvantages regarding bandwidth, stealthiness and robustness.

Finally, Table 4 reports the advantages and disadvantages of our proposal. As we can see, the main advantage of our proposal consists of all the precautions designed to improve the Robustness of the transmission: additional information is hidden in the same message to manage errors and packet loss. In all the other proposals, the Robustness is provided by the nature of the adopted header/protocol, but it is not pushed into the covert channel itself. In the following section, which is dedicated to the design and implementation of the channel, we will detail these features. Besides the previously mentioned robustness advantages of our proposal, we also need to say that the approaches presented in Tables 2 and 3 do not target a specific OS and do not provide any direct reference to an implementation. At the same time, in this paper (see Sect. 5), we also point the reader to available code

Table 3 Summary of some TCP-based steganography methods (**A** and **D** stand for "Advantage" and "Disadvantage" respectively)

Method	Bandwidth	Stealthiness	Robustness
ISN modulation (1997) [23]	8 bits per TCP connection	**A** Warden cannot clear the field	No relevant features
		D Message is not encrypted (could lead to suspicious value distributions if a symbol is used too often)	
		D ISN field is overwritten if the TCP connection is proxied	
TCP handshake bounce (1997)[23]	8 bits per TCP connection (message embedded in ISN field)	**A** Warden cannot clear the field	No relevant features
		A Works in networks that restrict destination IP addresses	
		D Can be neutralized with SAV (Source Address Validation)	
		D ISN field is overwritten if the TCP connection is proxied	
TCP timestamp option modulation (2002) [9]	1 bit per packet	**A** Based on a field that is used quite often	No relevant features
		D Warden can clear the field	
		D Warden can detect the channel through statistical analysis	
URG pointer modulation (2003) [11]	16 bits per packet	**D** Use of this field is suspicious (rarely used in normal circumstances)	No relevant features
		D Warden can clear the field	
OS sensitive ISN modulation (2005) [21]	Varies on each OS	**A** Warden cannot clear the field	No relevant features
		A Mocks true sequence numbers generated by each OS	
		D ISN field is overwritten if the TCP connection is proxied	
Encryption in ISN with decryption key in IP identification (2021)[8]	8 bits per TCP connection	**A** Warden cannot clear the fields	No relevant features
		A Message is encrypted (ensuring ISN values have a less suspicious distribution)	
		D IP identification values can look suspicious	
		D ISN field is overwritten if the TCP connection is proxied	

to be downloaded and used. Section 2.3 surveys software tools instead.

2.3 TCP/IP steganography tools

Table 5 collects a survey of network steganography tools that use TCP and IP headers to hide the message. These tools are not directly supported by scientific work, but we included them for completion.

To conclude this section, we briefly introduce these tools one by one. The *covert_tcp* program is a simple utility developed only for Linux systems and has been tested with kernel version 2.0.[6] This software uses raw sockets to assemble spoofed packets and encapsulate data extracted from a file provided on the command line. The transmission rate is reduced to one packet per second to ensure that packets arrive in the correct order since the reliability characteristics of TCP/IP cannot be used. Three data injection methods are available: the software inserts data into the IP ID field and the TCP Initial Sequence Number and Sequence Number fields. The channel encodes characters in the above fields by using their ASCII values. The tool is the closest to what we

[6] covert_tcp: https://github.com/zaheercena/Covert-TCP-IP-Protocol.

Table 4 Summary of our own network steganography method (**A** and **D** stand for "Advantage" and "Disadvantage" respectively)

Method	Bandwidth	Stealthiness	Robustness
Encryption in ISN with integrity check and retransmission	28 bits per TCP connection	**A** Warden cannot clear the field	**A** Can detect the loss of a packet and retransmit the message accordingly
		A Message is encrypted (ensuring ISN values have a less suspicious distribution)	**A** Can determine if the message arrived correctly or was scrambled by the warden
		A Mocks the behaviour of the real Windows TCP/IP stack	
		D ISN field is overwritten if the TCP connection is proxied	

Table 5 Software tools that implement a covert channel using TCP and IP protocols

Name	License	Lang.	Description	Release	Update
covert_tcp	MIT	C	Using 3 fields of TCP/IP	2016	2016
IPv6teal	–	Python	Steganography using IPv6 Flow Labels	2019	2019
NETsteg	GPL-3.0	Python	Steganography using TTL IP	2019	2019
Vtun	Copyright	C	Tunnel over TCP/IP	1999	2016
AckCmd	–	Visual C++	Covert channel with TCP ACKs	2000	2000
syn-file	–	C	Steganography using TCP SYNs	2017	2017

The columns respectively report the name of the tool, the adopted software license, the main programming language in which they have been developed, a brief description, and finally, the year of the first release and the last update of the software

propose in this paper; however, our implementation targets Windows OS (versions 10 and 11), and it also implements a retransmission protocol in case of errors or packet drops.

IPv6teal[7] is a tool written entirely in Python that exploits the Flow Label field. The tool consists of two modules, *exfiltrate.py* and *receiver.py*, and transmits 20 bits for each segment. Of course, both machines must support and have an IPv6 address and the Scapy library installed.

NETsteg[8] is a tool for carrying out a covert transfer of information via the most significant bits (MSB) of the TTL field. The project consists of two modules for sending and receiving data, written in Python and distributed under the GPL-3.0 license; the software module dedicated to reception also produces a .pcap file.

For Linux, FreeBSD and Solaris environments, the *Vtun*[9] tool is available. It provides a method for creating virtual tunnels on TCP/IP networks. It also allows a user to shape, compress and encrypt/decrypt the hidden traffic sent over these tunnels.

AckCmd was written by Arne Vidstrom for Windows 2000 OS. Although traditional remote shells also communicate via TCP, this tool only uses TCP ACK segments. It adopts port 80 on the client and 1054 on the server to increase the probability of crossing network firewalls. However, the packets often appear incorrectly formatted for HTTP, and the software is visible in the task list. This tool is now dated, and finding a downloadable web link is problematic. For the same reasons, we do not report *Back Orifice 2k*, probably the most famous data exfiltration tool on the Windows OS platform (Windows 95 and 98, Windows NT, 2000, XP).

Syn-file[10] is a tool available on GitHub and entirely written in C for data exfiltration. This technique exploits the sequence numbers synchronised by the TCP SYN field. It consists of a module that runs on the victim's computer and one that monitors network traffic to decode information.

For a more extended survey of network steganography methods at OSI layers 2, 3, and 4, the interested reader can refer to [2].

3 Establishing a context

In this section, we characterise the context in which we are supposed to design and develop our covert channel. Of course, tens of different scenarios can be considered due to the heterogeneity of networks, systems, and requirements in communication. Some scenarios can contrast: for example, whether the communication must be hidden in the same net-

7 IPv6teal: https://github.com/christophetd/IPv6teal.

8 NETsteg: https://github.com/chrispetrou/NETsteg.

9 Vtun: https://vtun.sourceforge.net.

10 Syn-file: https://github.com/defensahacker/syn-file.

work or if the two endpoints are located in two different networks. Different protocols could be involved (e.g., ARP in the first case). For this reason, we were forced to make some initial assumptions to provide a working solution for a scenario since a channel that works for any scenario is not possible.

The basic scenario we are picturing revolves around a host served by an Internet Service Provider (ISP) or, in general, networks without TCP proxying (see Sect. 4.2), which are very common in small/medium companies not protected by *Content Delivery Networks* (CDNs), for example. In this situation, we play the role of a data exfiltrator (Sender), and as such, we are interested in secretly and silently communicating with a Receiver outside this network. We may interpret an attacker who has gained administrator rights on a machine inside that network or the owner of a host who wants/needs to conceal its communication. The key takeaways that are most relevant to our work are the following:

- We focus on the *Transport* and *Network* layers of the OSI model. Working with layers below (such as *Data link*) would limit communication range to the local network and often requires tampering with the hardware. Working with layers above (such as *Application*) would reduce versatility in that using a protocol (e.g., FTP) to embed a message implies both the Sender and Receiver must run applications that rely on such a protocol (not guaranteed).
- We assume the information to be exchanged consists of 7-bit characters from the original ASCII set. However, any other content could be encoded into text.
- We aim to hide short pieces of information. In our tests, we sent up to 712 characters of text, which is enough, for example, to transfer a quite large cryptographic key, a username/password couple, or a short text message. Sending more characters requires "beaconing" the Receiver more times: our channel can do this, but the obtained traffic could be detected by specific tools (see Sect. 9).
- Some software or hardware-based protection is guarding the network (e.g., firewall and IDS).
- No (human) network-security operator will generally check network traffic unless alerted by the warden.
- We have admin rights on both the Sender/Receiver physical machines.
- On the Sender side, we assume internet navigation is authorised, meaning port 80 and port 443 are allowed (as often happens). However, since the channel works below the Application layer, we can easily suppose different ports are not outbound filtered.
- We assume the Receiver can receive packets on the chosen port from the network, as in our specific case, port 80 and port 443.
- The receiver's IP address is entirely unknown, and no host on the network would typically ever try to make contact with it. Traffic associated with our information exfiltration activity will be the only traffic related to it (in both directions). The IP address is not blocked (e.g., blacklisted by the sender network)
- The operating system running on the compromised host is Windows (either version 10 or 11). We choose Windows because it represents the preferred OS in desktop and laptop computers (more than 70%),[11] whose connection is frequently served by an ISP or a small/medium company, as in the scenario supposed at the beginning of this section. Considerations about different OSs are provided in Sect. 6.3.

Concerning the three good qualities of covert channels introduced in Sect. 2, our objective is to prioritise stealthiness first and then robustness, and to sacrifice bandwidth consequently as the minor goal. Despite 28 bits of information carried by each message, we will appreciate in Sects. 7 and 8 that common IDSs and statistical analysis cannot discover the channel. Error codes, checksums and retransmission will ensure a reasonable degree of robustness.

4 Defining a steganography method

After having outlined in Sect. 3 the characteristics of the scenario we consider, in this section, we motivate and describe the covert channel designed to work in such a scenario effectively. We identify the packet field(s) we use to hide the message in Sect. 4.1.

4.1 Choosing the most appropriate field

The first and most important aspect is choosing the field(s) in which the secret message is embedded. The problem with most fields is that traffic normalisers and active wardens are the countermeasures we fear the most. This is because the kind of action they need to operate is generally straightforward to implement and has a very low computational cost, usually consisting of 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 be applied as a default policy. For example, Handley et al. [10] suggested a traffic normaliser can delete all IP options from packets because their removal hampers diagnostic tools at most and generally does not affect higher layers' semantics at all. We ultimately expect the vast majority of wardens

[11] Desktop Operating System Market Share Worldwide Feb 2024, *statscounter.com*: https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-202402-202402-bar.

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 it too risky to hide the secret message in any field that can be overwritten with no adverse consequences. This rules out the possibility that we base our method on any of the following:

- TCP timestamp option modulation [9].
- DF bit modulation [15].
- Delay bit modulation [11].
- URG pointer modulation [11].
- TTL modulation [29].
- IP traceroute option modulation [27].
- Fragmentation-based methods [18] (while they do not necessarily rely on hiding the message in a field, traffic normalisation is still effective at neutralising them).
- IP timestamp option modulation [1].
- COTIIP [26] (since it relies on the MF bit).

Considering further methods in the literature, some involve altering the IP identification field, which cannot be easily accomplished in our case. The reason for this is how Windows handles the IP identification field. An in-depth explanation on this topic was provided by Klein [14], 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). Windows keeps a dedicated counter for each *(Source address, Destination address)* tuple related to outgoing traffic to initialise the IP identification field. 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 sent from a given source address to a given destination address, a new Path object is created and its identification counter is initialised with a random value.

Then, for each new packet sent from this source address to this exact destination address, the counter is incremented by one every time. Path objects are ultimately removed from the PathSet only on infrequent 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 have to always increment said value by one, lest we produce forged packets that display a substantial difference compared to legitimate ones. This means we cannot modulate the field as we please, and thus, we do not want to base our method on any of the following either:

- IP identification modulation [23].
- Using IP identification as a pointer to embedded data [24].
- Encryption in ISN with decryption key in IP identification [8].

Taking into account how assessing the possibility of using the IP identification field quickly led to the discovery of limitations that would excessively restrict our margin of action, a brief overview of the remaining fields can easily allow us to conclude that only a single field allows alteration with a sufficient degree of flexibility in our situation: **the ISN field**. In fact:

- *Source address (IP)* cannot be altered, as most firewalls nowadays adopt *Source Address Validation* as their default policy to prevent source address spoofing. If the field is instead modulated using a private address, its 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 the 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 limited flexibility in 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).

In Sect. 4.2, we evaluate the risks that may originate from using the ISN field.

4.2 Assessing the risks associated with ISN modulation

A first reason for concern is that it might be possible for the warden to predict how a given operating system typically assigns new sequence number values, thus allowing the warden to notice the difference between legitimate and forged ones. This issue was first raised by Murdoch and Lewis [21], but compared to their original context, sequence number generation has, in general, improved if compared to twenty years ago [5, 7]. At the time of writing, no *Common Vulnerabilities and Exposures id* related to an ISN generation weakness on Windows can be retrieved.

For this reason, it is hard for a warden to check whether all the ISNs have been correctly generated on a network. However, any middlebox that operates TCP proxying on all connections could automatically rewrite the field and thus neutralise the channel. In 2005, Murdoch and Lewis argued that such an operation would be computationally expensive and is therefore not very likely [21]. However, we are sure some of today's hardware firewalls can rewrite the ISN field. For example, *CISCO ASA* (Adaptive Security Appliance) randomises by default the ISN of the TCP SYN passing in both the inbound and outbound directions to prevent an attacker from predicting the next ISN for a new connection and potentially hijacking the new session; this feature makes our channel useless. Nonetheless, randomisation may bring some problems[12, 13]: the same CISCO documentation suggests to deactivate it in some cases (e.g., multiple paths to exit/enter from the network).[14] Moreover, most of the hardware firewalls on the market do not perform ISN randomisation by default.[15]

Bandwidth is also a tricky aspect to consider: on the one hand, the sequence number field is 32 bits long, 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 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 following packet would correspond to a waste of bandwidth. In light of this, we concluded that to maximise bandwidth, the type of behaviour our channel should try to simulate consists of multiple connection attempts continuously rejected by the receiver.

Finally, embedding plain text makes ISN value distributions look suspicious in the case of statistical analysis. For example, English only uses a fraction of the 128 symbols the original ASCII offers. Certain ones are used a lot more often than others (in particular vowels since they are used most frequently).

4.3 Illustrating our steganography method

To conclude this section, we describe the steps performed by the Sender and the Receiver to communicate secretly.

The steps followed by our steganography method ultimately consider all the risks we mentioned in Sect. 4.2 and are illustrated in Figs. 5 and 6. In order:

List of Steps 1
1. *The Covert Sender creates a forged TCP packet and embeds the secret message in the Sequence number field of the TCP header. Up to four ASCII characters are embedded using the 28 least significant bits of the field (the four most significant bits are instead used to implement error codes and control codes aimed at improving robustness, as we will show in Sect. 6.1).*
2. *To improve their value distribution, a randomised 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 randomised 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 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 to recreate the same randomised 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.*
8. *The Receiver then sets the random function's seed to a value corresponding 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*

[12] Cisco ASA TCP Randomisation Issue. Routing via Cisco ASA is changing TCP sequence/ACK numbers. Several VPN firewalls may drop packets due to incorrect TCP sequence numbers. This happens when the ASA randomises the TCP sequence numbers and another device also performs the same randomisation of the TCP sequence numbers: https://www.tunnelsup.com/cisco-asa-tcp-randomization-issue.

[13] Routing via Cisco ASA is changing TCP sequence/ACK numbers. In case of different outgoing and incoming paths to the same network, only some of the TCP sequence numbers are rewritten by CISCO ASA, thus preventing the channel from establishing a correct TCP connection: https://serverfault.com/questions/511059/routing-via-cisco-asa-is-changing-tcp-sequence-ack-numbers.

[14] ASA/PIX 7.x and Later: Mitigating the Network Attacks. CISCO documentation suggests when to disable TCP ISN randomisation: https://www.cisco.com/c/en/us/support/docs/security/asa-5500-x-series-next-generation-firewalls/100830-asa-pix-netattacks.html.

[15] For the sake of fairness, we do not report the names of such firewalls here.

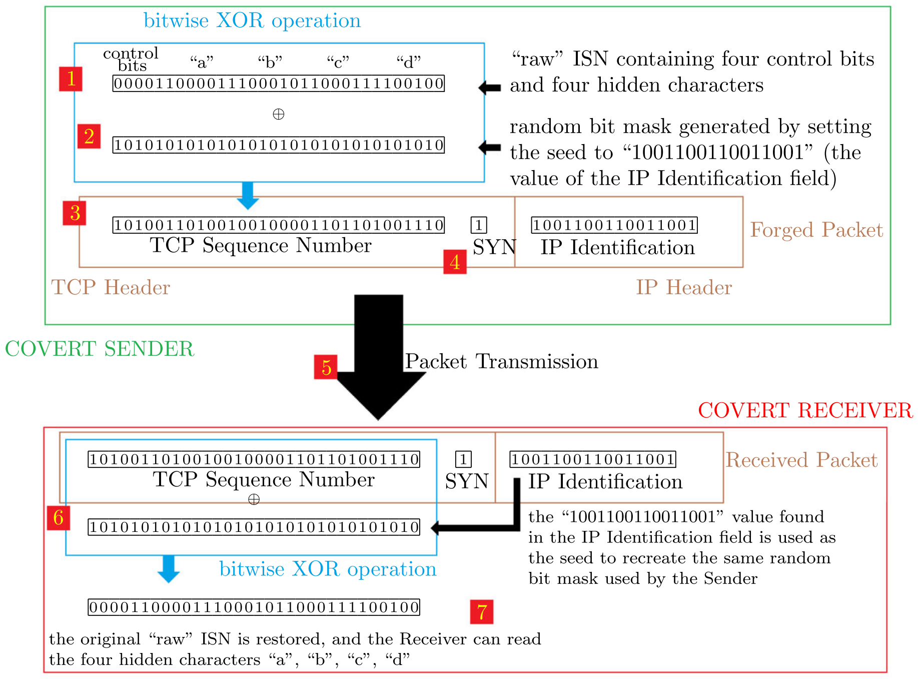


Fig. 5 Visual representation of how the Covert Sender embeds and transmits the secret message in our steganography method. The enumeration in the figure matches the steps illustrated in List 1 (colour figure online)

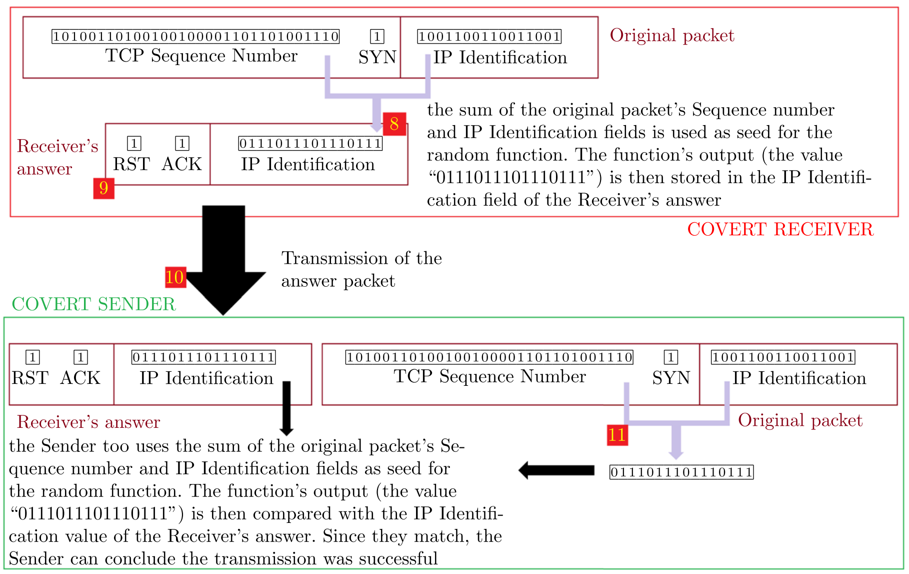

10100110100100100001101101001110
TCP Sequence Number
1
SYN
1001100110011001
IP Identification
Original packet
8
Receiver's answer
1 1
RST ACK
0111011101110111
IP Identification
9
the sum of the original packet's Sequence number and IP Identification fields is used as seed for the random function. The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

Fig. 6 Visual representation of how the Covert Receiver creates and transmits an answer in our steganography method. The enumeration in the figure matches the steps illustrated in List 1

randomised "signature" directly determined by the two fields' values.

9. *Finally, the Receiver assembles an answer consisting of yet another forged TCP packet. This packet's RST and ACK flags are set to 1 to pretend the Receiver is simply trying to reject the Sender's attempt to establish a three-way handshake. In reality, the Receiver confirms the original packet's reception and passes 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.*

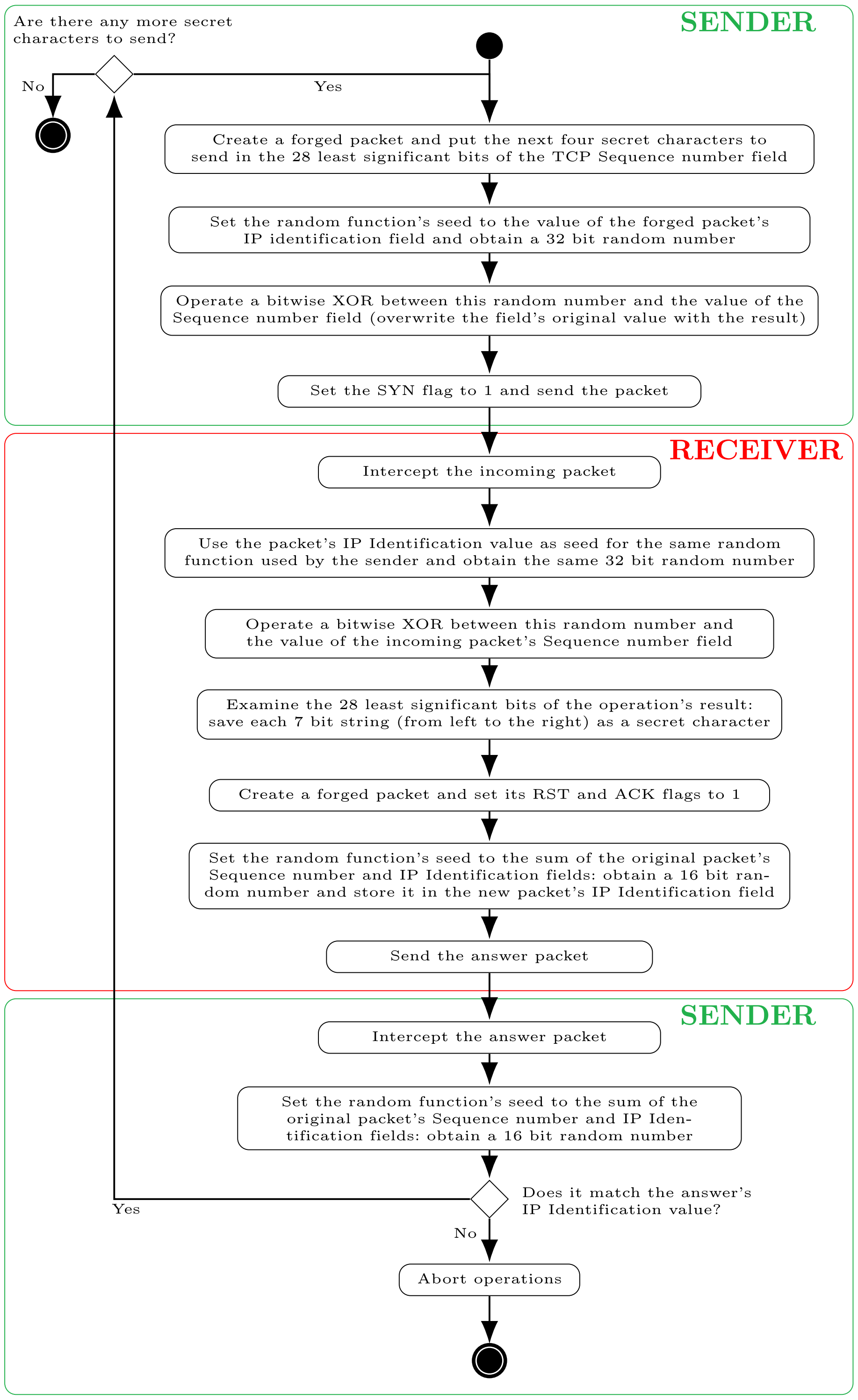

SENDER
Are there any more secret characters to send?
No
Yes
Create a forged packet and put the next four secret characters to send in the 28 least significant bits of the TCP Sequence number field
Set the random function's seed to the value of the forged packet's IP identification field and obtain a 32 bit random number
Operate a bitwise XOR between this random number and the value of the Sequence number field (overwrite the field's original value with the result)
Set the SYN flag to 1 and send the packet
RECEIVER
Intercept the incoming packet
Use the packet's IP Identification value as seed for the same random function used by the sender and obtain the same 32 bit random number
Operate a bitwise XOR between this random number and the value of the incoming packet's Sequence number field
Examine the 28 least significant bits of the operation's result: save each 7 bit string (from left to the right) as a secret character
Create a forged packet and set its RST and ACK flags to 1
Set the random function's seed to the sum of the original packet's Sequence number and IP Identification fields: obtain a 16 bit random number and store it in the new packet's IP Identification field
Send the answer packet
SENDER
Intercept the answer packet
Set the random function's seed to the sum of the original packet's Sequence number and IP Identification fields: obtain a 16 bit random number
Does it match the answer's IP Identification value?
Yes
No
Abort operations

Fig. 7 Activity diagram illustrating the steps mentioned in List of Steps 1

Figure 7 assembles all the steps performed by the Sender and the Receiver in a single picture for a more compact representation and to provide a better understanding.

5 Forging packets and connections

In this section, we will instead illustrate the most critical steps of the implementation process, particularly those aimed at creating forged packets that need to look as realistic as possible to a warden.[16] To keep the implementation as simple as we could, we opted to write both the Covert Sender and the Covert Receiver in Python,[17] more specifically using Scapy. Scapy is a packet manipulation library 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[18]. We used such a library to handle both the creation, the transmission and the reception of forged TCP packets.

5.1 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 typically occurs in Windows 11 when an attempt to start a new three-way handshake fails. This section describes the approach to mock a failed three-way handshake in Windows.[19]

The basic behaviour we ultimately aim to replicate is displayed in Figs. 8 and 9, respectively representing in *Wireshark*[20] a failed attempt to create a three-way handshake in Windows due to the destination IP sending no reply (Fig. 8) or replying with RST (Fig. 9).

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 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. Still, in the "no answer" scenario, the initial waiting time is always doubled after each failure (as Windows assumes the answer might have taken too long due to bad network quality). Our transmission shall follow the same pattern: the same forged packet shall be retransmitted several times. The Covert Sender shall also spend a specific amount of time waiting for the Receiver's answer after each transmission (doubled in case the response 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 pre-emptively scout their value by calling the Windows **"Get-NetTCPSetting -Setting Internet"** command and adapt its behaviour accordingly. On the other hand, the Receiver must always consider that the Sender always retransmits every packet multiple times. Since, unlike the Sender, it cannot know how many times exactly, it will ultimately discard any packet with the same sequence number as the one that arrived immediately before.

5.2 Initialising the forged packet's fields

After focusing on the three-way handshake in Sect. 5.1, we focus on how to select the other fields of the TCP/IP packet carrying the message.

Before beginning transmission, the Sender must also ensure every forged packet field is initialised 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 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 initialised.

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 connected and capable of reaching the internet. The Sender shall check Scapy's routing table through the **"conf.route"** variable to ensure a chosen interface can reach the internet. To ensure 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 not have been necessary and goes a bit off-topic regarding our original

[16] The Python implementation of both the Sender and the Receiver endpoints presented in this paper can be found at: https://github.com/Aldwyn47/TCPCovertChannel.

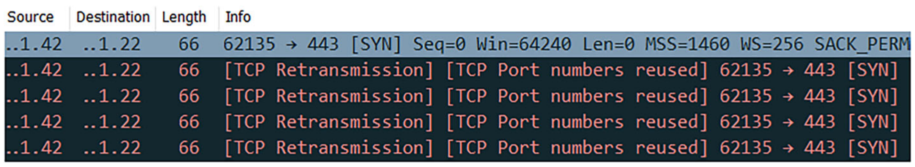
[17] Note that if the Python environment is not installed on the Sender machine, it can be easily zipped with all the required packages in a single .exe file (e.g., by using *PyInstaller*: https://pyinstaller.org/en/stable/).

[18] The Scapy Python package: https://github.com/secdev/scapy.

[19] In summary, our covert channel is based on failed three-way handshakes, and the scenario in Sect. 3 involves a victim using Windows OS.

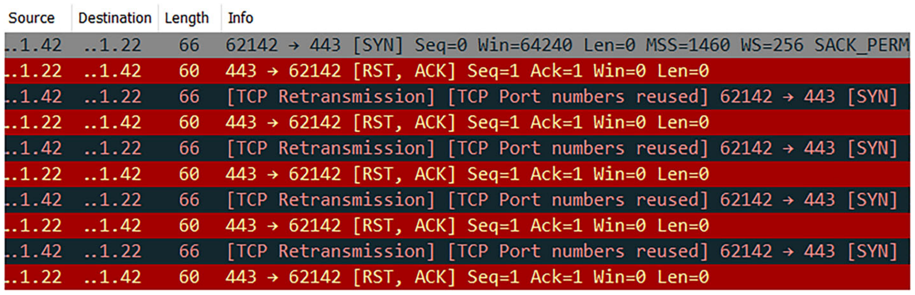
[20] Wireshark, a free and open-source packet analyser.: https://www.wireshark.org.

Fig. 8 Wireshark snapshot showing a failed attempt to create a three-way handshake in Windows (the destination host sent no reply)



```
Source  Destination Length  Info
..1.42  ..1.22      66      62135 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62135 → 443 [SYN]
```

Fig. 9 Wireshark snapshot showing a failed attempt to create a three-way handshake in Windows (the destination host replied with RST)



```
Source  Destination Length  Info
..1.42  ..1.22      66      62142 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
..1.22  ..1.42      60      443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
..1.22  ..1.42      60      443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
..1.22  ..1.42      60      443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
..1.22  ..1.42      60      443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
..1.42  ..1.22      66      [TCP Retransmission] [TCP Port numbers reused] 62142 → 443 [SYN]
..1.22  ..1.42      60      443 → 62142 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

goal of only working with layers 3 and 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),[21] leaving us no choice but to handle the layer 2 transmission ourselves.

The remaining fields of the TCP and IP headers should be initialised with values 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 actual 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 primary process to intercept the packet created to fulfil the subprocess' request. 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 challenging 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 actual 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 the source port value immediately. Also, it prevents the possibility of another connection using the same value (since the TCP/IP stack effectively "reserved" it for the loopback connection).

5.3 Initialising interface dependent fields

To conclude the description of implementing the covert channel in Windows, this section focuses on the parameters related to the network interface more than the TCP/IP stack.

However, a few additional fields are interface-dependent and cannot be copied from the connection attempt on the loopback interface. Their values can either be set to their expected value in Windows (the most silent but least accurate choice) or be scouted using the same strategy. This time, however, the Covert Receiver's address is the destination for the subprocess connection (thus generating accurate values but also creating a connection attempt visible on the private network). In the case of the second option, the attempts shall occur on different destination ports to allow the Receiver to distinguish connection attempts that carry secret information from connection attempts used to scout these values.

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 the same involved IP addresses).

However, in case of a system reboot, all Identification values are always reinitialised, 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 also store this information. Before accessing the database, the Sender shall always repeat this command to compare its output to the stored value. If they

[21] A bug in the Scapy library concerning Windows operating system and the ARP protocol: https://github.com/secdev/scapy/issues/3474.

differ, it implies a system reboot occurred at some point, and the database must thus be cleared.

6 A word on robustness

The TCP protocol is well-known for its robustness; however, we cannot access such robustness features even though we are technically using TCP, we cannot access such robustness features. Hence, transport becomes unreliable, and the Sender is responsible for handling possible errors. This section will illustrate the adaptations we introduced in our code to address the most common issues, e.g., packet drops.

6.1 Termination character and error correction codes

This section presents all the additional information exchanged between the Sender and the Receiver to make transmitting the hidden message as reliable as possible.

In Sect. 5.1, we mentioned how the Sender program transmits each packet multiple times to replicate the typical sequence of events that usually occur 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 first 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 extract the secret characters correctly.

In light of this, the Sender can consider a packet's transmission successful only if the Receiver replies with RST to the first packet: what happens to any of the following retransmissions is entirely irrelevant. On the other hand, the Receiver must only accept information from the first packet and discard the information in any subsequent packet with 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 Sender and that each different Sender shall notify the Receiver when there is no more secret text to send.

Among symbols offered by original ASCII, we decided to commit the 0000100 bit-string, corresponding to the *End of Transmission* (*EOT*) character to this purpose; this means the Sender shall always append it as the 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. We 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 EOT intent, 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)	0 0 0 0	1 1 0 0 0 0 1	1 1 0 0 0 1 0	1 1 0 0 0 1 1	1 1 0 0 1 0 0
2)	0 1 0 1	1 1 0 0 1 0 1	1 1 0 0 1 1 0	1 1 0 0 1 1 1	1 1 0 1 0 0 0
3)	0 0 0 1	1 1 0 0 1 0 1	1 1 0 0 1 1 0	1 1 0 0 1 1 1	**1 1 0 1 0 0 0**
4)	0 1 0 1	1 1 0 0 1 0 1	1 1 0 0 1 1 0	1 1 0 0 1 1 1	**0 0 0 0 1 0 0**
5)	0 0 1 0	1 1 0 0 1 0 1	1 1 0 0 1 1 0	**0 0 0 0 1 0 0**	1 0 1 0 1 0 1
6)	1 1 1 1	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 1 0 0

Fig. 10 Visual representation of some "raw" ISNs, showing the possible uses of the four most significant bits

Figure 10 provides examples of different "raw" ISNs (i.e. before the randomised bitmask is applied to them) to illustrate the possible uses of the four most significant bits.

1. RAW ISN containing four regular characters. The four most significant bits are on the left and are set to 0000. The twenty-eight least significant bits are separated into 7-bit groups representing characters (in this case, "a", "b", "c" and "d").
2. This ISN contains four regular characters, too. The four most significant bits are set to 0101, and the embedded characters are "e", "f", "g", "h".
3. The most significant bits of this ISN are set to 0001, which points to the fourth character (in bold). However, this character is not the 0000100 string, so this ISN contains four regular characters.
4. One of the characters embedded in this ISN is the 0000100 string (in bold), but the first 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 (the first two characters) followed by the termination character (in bold), properly marked with the correct label (first four bits). The last seven bits (light grey) are ignored as they come after the termination character.
6. This ISN is an error correction code. The first four bits are 1111, and the remaining twenty-eight bits have a value that falls in the [0,5] range (in this case, 4).

In any other case, the message is instead a standard 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. One of the 4-bit labels is used in some scenarios, but the marked character is not the 0000100 string. However, one important exception is when the four most significant bits are all set to 1. 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. Instead, the Sender uses an error correction code to fix a transmission error.

After introducing how termination characters and error correction codes are encoded in the messages, Sect. 6.2 presents how this information is used in case of events such as message drops.

6.2 Common hazards

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 common scenario is one of the Sender's retransmissions receiving no answer. This happens because the packet or the Receiver's RST answer is dropped. As already said, retransmission packets are irrelevant since the Receiver ignores them by default. Another scenario corresponds to the Sender's packet never receiving any answer (both the first time it is transmitted and after each retransmission). We do not expect packet loss to be higher than 1% at worst for the average network. Such a situation likely implies the Receiver is either not operational or unreachable. The Sender shall thus stop all operations. Another scenario corresponds to the Receiver replying to the Sender's packet with an answer packet with the wrong signature in the Identification field. As we explained when we described the embedding process, this indicates either the original packet's sequence number or its identification value was altered while the packet was travelling (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 corresponds to the Sender's packet not receiving an answer the first time it is sent but receiving one (or more) during subsequent retransmissions. Since the Receiver answered at least once, it is operational and reachable. Either the original packet itself or its RST answer was dropped: these two possible causes lead to two different "variants" of this scenario.

The **first variant** occurs when the Sender's packet is dropped (Fig. 11. In this case, the Receiver effectively fails to receive the Sender's information. It 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 were part of the secret text sent by the Sender: most of the time, this incorrect information consists of four gibberish characters, but sometimes it could include the special "termination" character or in worst case scenario even be an error correction code.

Fortunately, the Receiver's first answer allows the Sender to detect this error because it always matches the packet accepted as relevant. Suppose such a packet is not the original one. In that case, the Sender can calculate its associated bit mask from its Identification value, remove it from the packet's sequence number field (as usual, through a bitwise XOR operation) and obtain the wrong information mistakenly accepted by the Receiver. Depending on the nature of this information, the Sender can 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 immediately 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%).[22]
- *Delete the last "termination" character and the two characters immediately 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 immediately before*, 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* 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 resume its operations. Still, the four characters that initially failed to arrive must be embedded again in the

[22] This and all the following probabilities have been computed by having 2^{32} as the denominator (all the possible bit combinations), and as numerator the number of configurations that satisfy the related code. In the first bullet, for example, the Receiver may receive *0001 - xxxxxxx - xxxxxxx - xxxxxxx - 0000100* (where x can be either 0 or 1), corresponding to 2^{21} possible alternatives: $2^{21} \div 2^{32} \simeq 0.0005$.

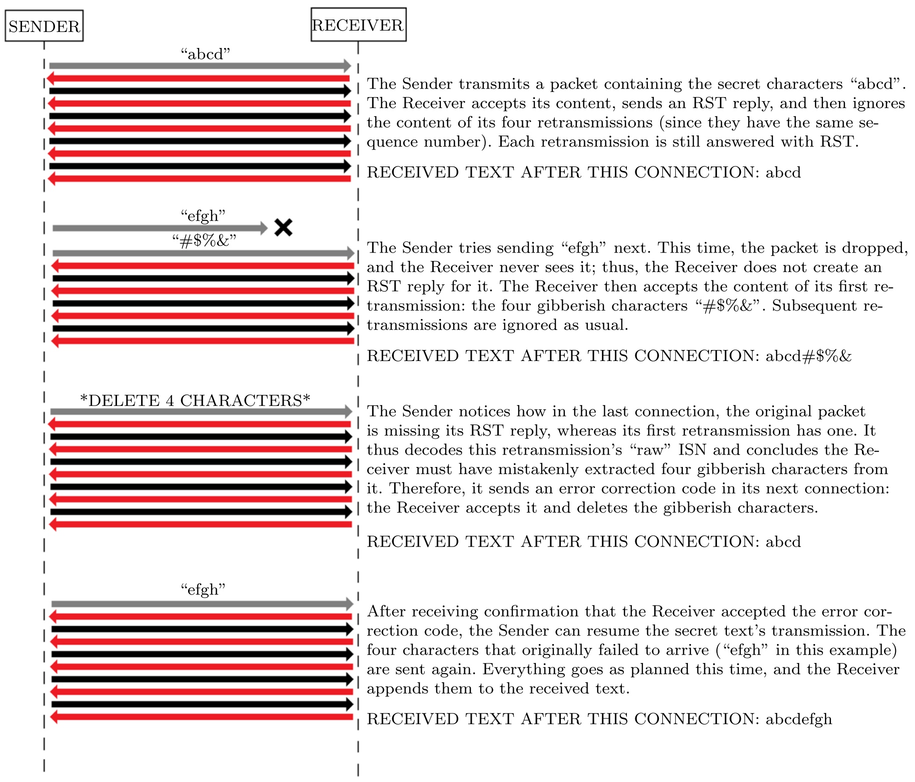

SENDER
RECEIVER
"abcd"
The Sender transmits a packet containing the secret characters "abcd". The Receiver accepts its content, sends an RST reply, and then ignores the content of its four retransmissions (since they have the same sequence number). Each retransmission is still answered with RST.
RECEIVED TEXT AFTER THIS CONNECTION: abcd
"efgh"
"#$%&"
The Sender tries sending "efgh" next. This time, the packet is dropped, and the Receiver never sees it; thus, the Receiver does not create an RST reply for it. The Receiver then accepts the content of its first retransmission: the four gibberish characters "#$%&". Subsequent retransmissions are ignored as usual.
RECEIVED TEXT AFTER THIS CONNECTION: abcd#$%&
DELETE 4 CHARACTERS
The Sender notices how in the last connection, the original packet is missing its RST reply, whereas its first retransmission has one. It thus decodes this retransmission's "raw" ISN and concludes the Receiver must have mistakenly extracted four gibberish characters from it. Therefore, it sends an error correction code in its next connection: the Receiver accepts it and deletes the gibberish characters.
RECEIVED TEXT AFTER THIS CONNECTION: abcd
"efgh"
After receiving confirmation that 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. Everything goes as planned this time, and the Receiver appends them to the received text.
RECEIVED TEXT AFTER THIS CONNECTION: abcdefgh

Fig. 11 Visual representation of the packet drop scenario's first "variant". Arrows labelled with text represent the first packet delivered by the Sender in each connection attempt (i.e. the one from which the Receiver extracts information), Sender-to-Receiver arrows represent its retransmissions, and Receiver-to-Sender arrows represent the Receiver's RST replies

following 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 (see Fig. 12). In this case, the Receiver effectively accepts the correct packet, but the Sender mistakenly convinces itself that a retransmission was accepted in its place. In such a situation, the Sender should theoretically do nothing at all. Unfortunately, the Sender ultimately has no way of knowing 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 of having it always assume the dropped packet is the original to 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. Suppose the lost packet is the Receiver's answer (i.e. the Sender's assumption is wrong). In that case, the Sender's error correction code causes the Receiver to delete four correct characters that are sent again anyway, ultimately slowing down the communication but not damaging it.

Unfortunately, there are still two extremely rare situations where communication is irreparably compromised. The first happens when the packet drop occurs while the Sender tries to transmit an error correction code, regardless of which variant. 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 make a blind guess as 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 should abort operations if such a situation occurs. Finally, 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 occurs once, the probability of such a situation is ultimately equal to that of the packet drop scenario happening twice in a row. The network's packet drop rate influences this probability: assuming no more than 1% of all packets are dropped and taking into account how anyone 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 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, which alters the Receiver's accepted text that is not automatically corrected with the Sender's subsequent transmission. Our program currently has no way to deal with such

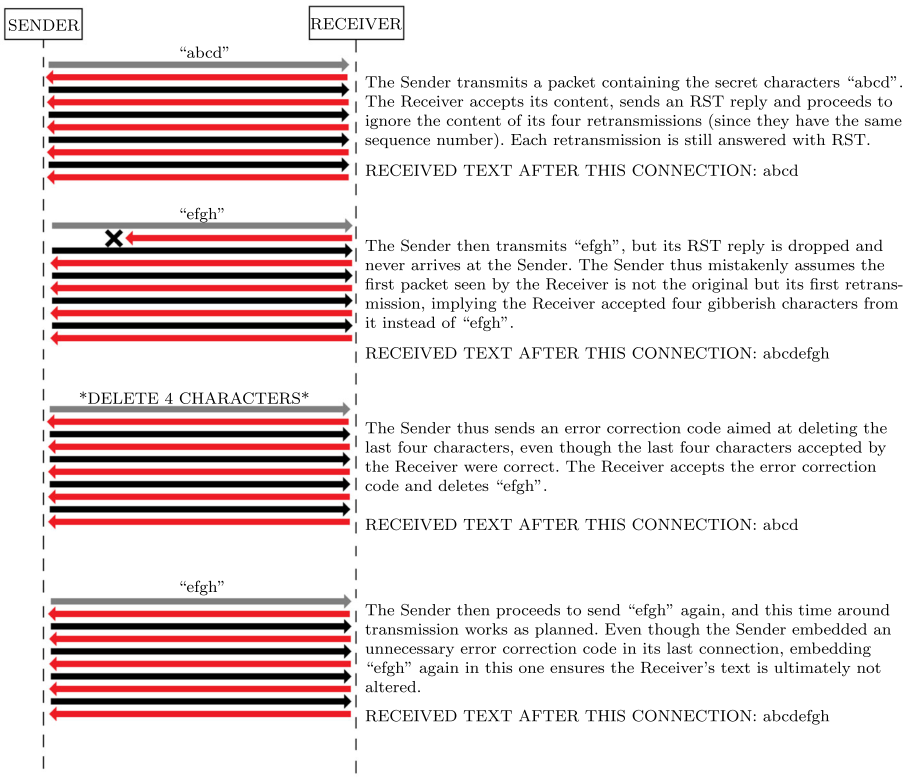

SENDER
RECEIVER
"abcd"
The Sender transmits a packet containing the secret characters "abcd". The Receiver accepts its content, sends an RST reply and proceeds to ignore the content of its four retransmissions (since they have the same sequence number). Each retransmission is still answered with RST.
RECEIVED TEXT AFTER THIS CONNECTION: abcd
"efgh"
The Sender then transmits "efgh", but its RST reply is dropped and never arrives at the Sender. The Sender thus mistakenly assumes the first packet seen by the Receiver is not the original but its first retransmission, implying the Receiver accepted four gibberish characters from it instead of "efgh".
RECEIVED TEXT AFTER THIS CONNECTION: 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 correct. The Receiver accepts the error correction code and deletes "efgh".
RECEIVED TEXT AFTER THIS CONNECTION: 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 last connection, embedding "efgh" again in this one ensures the Receiver's text is ultimately not altered.
RECEIVED TEXT AFTER THIS CONNECTION: abcdefgh

Fig. 12 Visual representation of the packet drop scenario's second "variant". Arrows labelled with text represent the first packet delivered by the Sender in each connection attempt (i.e. the one from which the Receiver extracts information), Sender-to-Receiver arrows represent its retransmissions, and Receiver-to-Sender arrows represent the Receiver's RST replies

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 likelihood 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.

6.3 The Sender and Receiver network architectures and operability

This section lists some additional assumptions related to the network and the machine from which the Sender and the Receiver operate.

Finally, we must also point out that our program is designed to work with some conditions on the network architecture, and it does not necessarily work if such an architecture is changed too radically. Specifically, we expect the Receiver to possess its public IP address or be on a private network where HTTPS packets are forwarded towards it (specifically by opening port 443). On the other hand, the Sender is expected to either be on a private network equipped with a default gateway that allows it to reach the outer Internet or possess its 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 believe the channel could theoretically be adapted to include some form of multiplexing based on the Identification field. Still, we chose not to prioritise such a feature at this time.[23] Please note that the described covert channel works for a Sender behind a NATted network, which is the case for most home networks served by an ISP and small/medium companies proposed in our context (see Sect. 3).

Even if we implemented the channel by considering the Sender and Receiver running on Windows machines, the design of the presented covert channel is wholly based on the TCP and IP protocols, which are common to all the nodes communicating together on the same network, despite the specific OS. The Python Scapy package adopted to manipulate packets can be used multi-platform, and all the configuration and *PowerShell* commands in Sect. 5 have a correspective in other OSs, such as Linux and MacOS. However, the presented implementation is tailored to the TCP/IP stack of Windows, as presented in Sects. 5.1, 5.3 in particular, to mock as much as possible the behaviour of the OS the warden knows that the machine runs (for example, Windows sends a SYN segment four times when a failure occurs in a three-way handshake). To summarise, porting the channel to a different OS is relatively straightforward if we decide to

[23] If the same user manages two or more Senders in the same gatewayed network, a trivial solution could be sacrificing some bits of the hidden message to represent a sort of Sender ID, for example.

put the Windows imitation aside, while adapting it to a new TCP/IP stack requires an investigation of the different OS.

7 Field tests

This section collects the experiments we performed to check if the implementation presented in Sects. 4, 5, and 6 correctly works stealthily (see Sect. 7.1).

Our implementation was ultimately tested across six different environments, always obtaining successful communication:

(i) Both the Sender and the Receiver were located on the same private network and gained internet access through UTP cables (Unshielded Twisted Pair). This environment was mainly used to conduct the first debugging.
(ii) The Sender and the Receiver were virtual machines connected to two 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 a gateway in both directions. All virtual machines and networks were hosted on the same computer.
(iii) The Receiver was connected to an ISP network. At the same time, the Sender was given internet access exclusively through a Wi-Fi hotspot created with a nearby mobile device (managed by a different ISP).
(iv) The Receiver and the Sender were connected to two different ISP networks in the same city. The Sender was launched from a virtual machine with internet access in Bridge mode.
(v) The Receiver and the Sender were connected to two different ISP networks in different cities.
(vi) The Sender was located in a university department network, and the Receiver was connected through an ISP. At least the Sender network was filtered by the department and university firewalls.

The length of the secret messages sent in these tests varied from a minimum of 4 to a maximum of 712 characters. Several packets were lost, but the error correction codes worked as intended and preserved the transmissions' integrity. Except for some initial failures caused by various bugs that needed correction, our tests were 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.

7.1 Assessing stealthiness

To evaluate the stealthiness of our transmissions, we first used Wireshark in three instances to capture traffic on the Sender's network. We ultimately produced three .pcap files to feed them later as input to programs designed to scan network traffic for anomalies.[24] All three .pcap files were recorded in *environment iii* and thus included only traffic related to a single workstation, with very sporadic activity aside from the single secret communication featured in each: a different user logged in to perform some email checking and some web-browsing to mock the behaviour of an ordinary workstation. The goal was to have a minimum amount of surrounding traffic to make a comparison, but not too much, to make the traffic channel stand out among the surrounding traffic as much as we could. In order:

– *PCAP N.1* featured the transmission of 712 characters. Each of the Sender's connection attempts included one original packet (containing four secret characters) and four retransmissions. After each connection attempt, the Sender was instructed to wait for a 1-s delay. Overall, the .pcap file covers a timespan of approximately 5 min and 15 s, during which a total of 180 TCP connections were created with the purpose of carrying secret information.
– *PCAP N.2* featured the transmission of 356 characters. Each of the Sender's connection attempts included one original packet (containing four secret characters) and four retransmissions. After each connection attempt, the Sender was instructed to wait for a 2-s delay. Overall, the .pcap file covers a timespan of approximately 7 min and 35 s, during which a total of 90 TCP connections were created with the purpose of carrying secret information.
– *PCAP N.3* featured the transmission of 36 characters. Each of the Sender's connection attempts included one original packet (containing four secret characters) and four retransmissions. After each connection attempt, the Sender was instructed to wait a 300-s delay. Overall, the .pcap file covers a timespan of approximately 43 min, during which a total of 9 TCP connections were created with the purpose of carrying secret information.

We then proceeded to install two different network security programs. The first one was Suricata (version 6.0.1),[25] an open-source IDS developed by Open Information Security Foundation (OISF) that uses rule-based and signature-based detection to monitor network traffic for signs of suspicious

[24] These three .pcap files and the ten ones in Sect. 8 are offered as a dataset on Kaggle: https://kaggle.com/datasets/dba976f795460c93279d3c95649742e18afdeb3f28f3aba6c0ed00721a3ccf74.

[25] Suricata, an open-source based IDS and Intrusion Prevention System (IPS): https://suricata.io/.

activity. Many online communities actively produce new rules to keep Suricata's performance up to date. In light of this, one of Suricata's major advantages is how easily it can be enhanced and customised by adding 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*, *stamus/lateral*, *tgreen/hunting*.

The second security software we installed was Zeek (version 5.1.1).[26] 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). Like Suricata, Zeek can be customised and extended to fit users' needs better. We thus extended our Zeek installation by adding the following packages: *conn-burst*, *ja3*, *HASSH*, *BZAR*, *dovehawk*, *spicy-analysers*, *pingback*, *icmp-exfil-detection*, *http_csp*, *http-stalling-detector*, *zeek-httpattacks*.

We then used both programs to scan our .pcap files to see if they could detect our secret transmissions: neither Suricata nor Zeek ultimately raised any alert. It should be noted that IDS of this kind are only as good as their configuration, meaning that the results might have been different with better packages or rules. Nevertheless, for now, we feel safe to confirm what was already pointed out in [12]. While many techniques aimed at defeating network steganography have already been proposed, many existing network security tools have not necessarily implemented them.

8 Our channel and statistical traffic analysis

In this section, we will show how security measures, more specifically tailored to counter the type of transmission used in our channel, can be far more effective at detecting or suppressing it. While the IDS-based approaches in Sect. 7.1 completely ignore our covert channel, this section considers a tool for statistical traffic analysis.

RITA (Real Intelligence Threat Analytics) is open-source software designed to analyse network traffic to detect signs of malicious activity, in particular beacons associated with command and control software[27]. A beacon corresponds to 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 to signal it is ready to receive remote commands. While our covert channel is not a command and control software, it resorts to communication that follows a regular pattern due to packet retransmissions, ultimately making it 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"[28]. Simple entries consist of data generated from a single .pcap file. In contrast, rolling entries consist of data generated from multiple .pcap files stored over time, called "fragments": these fragments are "rolling" because once 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 covering 1 h of network traffic (for 24 h). During the scan process, individual connections are first gathered into different sets: each contains all connections that went from the same source address to the same destination address.

Each of these sets (i.e. each *(Source, Destination)* tuple) is then assigned a global score that ranges from 0 (most not malicious) to 1 (most malicious) based on statistical analysis. This analysis only takes place if the number of connections in a given set surpasses a certain "grace threshold" otherwise, the set is ignored, and its connections are ultimately considered not malicious (in RITA's default configuration, this threshold is set to 20).

The statistical analysis considers four main aspects, each one associated with its score: the global score is none other than the mean of these four scores. However, RITA can be configured to assign different weights as well. Since RITA consists of a command line tool with no user interface, each resulting score is ultimately presented to the end user as a simple number printed on the terminal. The underlying computations are never shown to the end user either. Still, we were able to infer them by looking either at the source code itself[29] or at the official documentation[30]. We shall now illustrate how each of the four scores is determined. In order:

- *Timestamp score* is based on the frequency and regularity of network traffic over time. Repeated connections with fixed intervals between them score higher, while sporadic connections with irregular intervals score lower. According to RITA's documentation, the timestamp score

[26] Zeek, a free and open-source software network analysis framework: https://zeek.org/.

[27] RITA Website: https://www.activecountermeasures.com/free-tools/rita/.

[28] The source code of RITA: https://github.com/activecm/rita.

[29] The source code of the packet analyser module, producing all the alert scores of RITA: https://github.com/activecm/rita/blob/master/pkg/beacon/analyzer.go.

[30] The source code and description of the "beacon" package of RITA: https://github.com/activecm/rita/tree/master/pkg/beacon.

is calculated as,

$$((\frac{1}{3})(1-|TSBowleySkew|)+\frac{1}{3}(\max(1-(\frac{TSMADM}{30}),0))$$
$$+(\frac{1}{3})(TSConnCount))$$

where "TSConnCount" is the ratio of the number of connections to the number of 10-s periods in the whole dataset, "TSBowleySkew" is the Bowley Skew of intervals[31] and "TSMADM" is Median Absolute Deviation around the median of intervals.

- *Data size score* is based on how much data is transferred with each connection. Connections that always send the same amount of data score higher, while connections that send different amounts score lower. According to RITA's documentation, the data size score is calculated as

$$\frac{1}{3}(1-|DSBowleySkew|)+\frac{1}{3}(\max(1-\left(\frac{DSMADM}{32}\right),0))$$
$$+\frac{1}{3}(max(1-\left(\frac{DSMode}{65535}\right),0))$$

where "DSMode" is the data size mode (i.e., the data size that appears the most often), "DSBowleySkew" is the Bowley Skew of data sizes, and "DSMADM" is the Median Absolute Deviation around the median of data sizes.
- *Duration score* is simply the ratio between the timespan in which connections persist and the timespan covered by the database entry as a whole. For example, if all connections occur within a single hour and the database entry covers one day, the score will be approximately 0.0417 (i.e. one divided by twenty-four).
- *Histogram score* is determined by splitting the timespan covered by the database entry into an appropriate number of buckets (24 by default, reflecting a twenty-four-hour period). Each bucket is then assigned a frequency value representing how many connections occurred within its related timeframe. The result is a histogram on which RITA computes the coefficient of variation (CV). Coefficient of variation, defined as the standard deviation divided by the mean[32], is a measure of relative variability: a lower CV indicates that the frequencies are more tightly clustered around the mean, which could be a sign of regular beaconing activity. Histogram score is ultimately calculated as $1 - \mathrm{CV}$, capped at 1, so a lower CV results in a higher histogram score. It is worth noting that if all connections occur within a timespan small enough to fit inside one single bucket, the resulting coefficient of variation will be high. Hence, a very low duration score also implies a very low histogram score.

We ultimately used RITA to scan the same three .pcap files we had 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. However, it should be noted that RITA did not ultimately identify any of our transmissions as a covert channel and merely harboured suspicions related to their communication pattern. Moreover, since RITA's logic is that virtually any transmission could be a beacon, regular, non-malicious network traffic is not uncommon 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 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 Fig. 13).

The **first modification** the Sender continually operates just one retransmission of each SYN packet rather than a number aimed at reflecting the compromised host's Windows settings. Reducing retransmissions makes the Sender's communication more sporadic, supposedly improving its timestamp score. Moreover, if the secret text is short enough, the Sender can convey it using several packets that do not exceed RITA's "grace threshold", thus avoiding detection altogether. For example, considering how each forged packet's sequence number can transport 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 precisely 20: RITA will not raise any alert unless its settings are edited to lower its grace threshold. Eliminating retransmissions would further increase the Sender's chances of avoiding detection entirely. However, doing so would also penalise robustness too much: we thus concluded the best compromise is likely to keep one retransmission for each forged SYN packet.

The **second modification** consists of the Sender refraining from creating one single real connection towards the Receiver to scout the values of fields such as Identification (IP) or Maximum Segment Size (TCP option) and later use them to initialise forged packets. The logic behind this modification is that reducing the total number of transmissions is vital to reduce RITA's timestamp score as much as possible. The forged packets' fields shall ultimately be initialised 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 of 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 approximately 0.05 (which would also guarantee a histogram score of almost 0). In this sense, the three .pcap files we originally recorded were biased. Since we started our captures right before begin-

[31] A quick reference to how Bowley Skewness is computed: https://mathworld.wolfram.com/BowleySkewness.html.

[32] A quick reference to how the coefficient of variation is computed: https://mathworld.wolfram.com/VariationCoefficient.html.

ning 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 of the Sender waiting for different time intervals after sending each packet (both the original and its single retransmission). This modification makes transmissions look erratic to reduce RITA's timestamp score.

Unfortunately, changing the channel's behaviour caused it to lose a bit of stealthiness regarding how much a forged transmission of our own is akin to a real three-way handshake failure in Windows. Still, fortunately, this did not cause either Zeek or Suricata to raise any alert in any of the preliminary tests we ran once these modifications were complete. These initial 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 0.515 in any of our preliminary tests. We thus decided to conduct one final test to assess 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 recorded ten different .pcap files and drew conclusions from the average score obtained. Each .pcap file covered a total timespan of 60 min and included one secret transmission used by the Sender to exfiltrate an average of 550 bits within a time window of approximately 220 s (always resulting in a duration score of about 0.06 and a histogram score of 0).

Within the 60-min timespan, we also instructed the compromised host to browse various popular websites to recreate a portion of the typical network traffic that generates false positives in RITA's scans as realistically as possible. 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 typically occur during the average working day. Our navigation ultimately only occurred in two specific moments separated by a 20-min 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 the end of their daily shift). No navigation shall instead happen in the remaining 40 min because they are supposed to represent late evening and night hours, during which there is little to no network traffic (refer to Fig. 14 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, and 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 6, which includes entries that summarise 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 transmission, even in a worst-case scenario: they 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 filtering is applied preemptively. Unfortunately, we still do not consider this enough to automatically guarantee our channel's safety against RITA. Still, we were at least able to show how adapting the channel's behaviour depending on the specific context can positively impact its stealthiness.

9 Conclusion and future work

We have proposed a covert channel based on TCP SYN packets. The implementation runs in Windows and has been designed with stealthiness and robustness in mind by trying to mock the Windows TCP/IP stack and offering automatic retransmission of packets in case of errors and packet drops. Tests show that the channel works in different network envi-

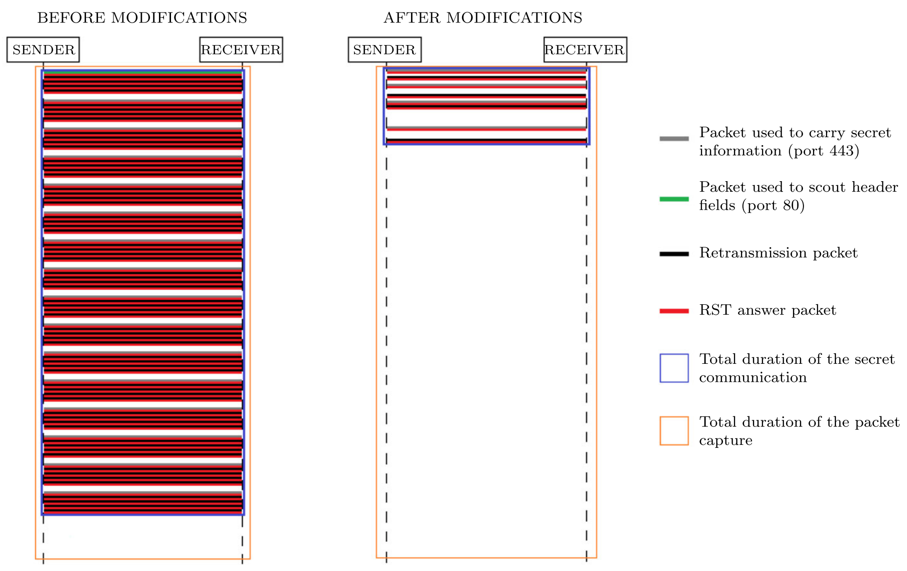

BEFORE MODIFICATIONS
AFTER MODIFICATIONS
SENDER
RECEIVER
SENDER
RECEIVER
Packet used to carry secret information (port 443)
Packet used to scout header fields (port 80)
Retransmission packet
RST answer packet
Total duration of the secret communication
Total duration of the packet capture

Fig. 13 Visual representation of the Sender's communication before (left) and after (right) modifications

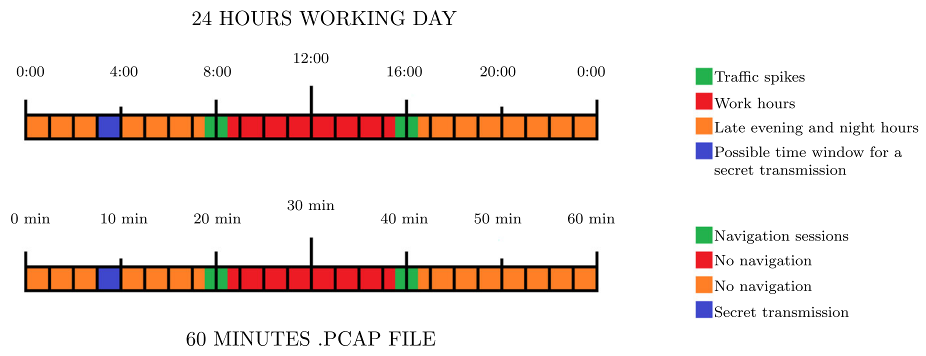


Fig. 14 Scale comparison between an average working day (above) and one of our 60 min .pcap files (below)

Table 6 Average RITA score of our covert channel (in bold) compared to other false positives

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 6 continued

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
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
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

Table 6 continued

Score	Domain	Autonomous system
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
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

Reverse DNS lookup was used to determine each false positive's associated domain and autonomous system from its original IP destination address

ronments, also against statistical-based traffic analysis, as performed by RITA.

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

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

Speaking of the channel's performance regarding stealthiness, robustness, and bandwidth, we feel relatively satisfied, but at the same time, we also learned some hard lessons along the way. Upon defining our steganography method, we mainly used storage-based existing methods as a source of inspiration due to the higher bandwidth they offer on average. However, we 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 (which comes at the cost of some network troubleshooting in some cases, see Sect. 4.2).

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: to adapt our channel, we ultimately had to alter its transmission pattern: such a modification would have been inherently 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 in artificial intelligence (which we believe will become a very effective tool for improving the accuracy of passive wardens).

9.1 Future work

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

In light of this, we believe a promising development direction for future covert channels could be to invest in redundancy 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 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 human error also increases: in this sense, if even one exfiltration technique is successful, the attacker ultimately wins.

An adaptation-focused approach should ultimately be adopted in future versions of our covert channel to improve its ability to identify different situations in which transmission fails, their potential causes, and the 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. Training the channel to change its steganography method altogether would be more tricky. Still, the channel's ability to notice external alterations to sequence number values already provides a perfect starting point for future developments. To enhance our channel with additional exfiltration methods, we will conclude our presentation by providing a high-level illustration of some other alternatives to the way we had initially taken into account but then temporarily put aside for various reasons (mostly bandwidth-related).

One first alternative consists of a storage-based method revolving around modulation of the IP destination address. In this context, the attacker must possess multiple public IP addresses, each corresponding to a different symbol: the Covert Sender ultimately transfers information by contacting the destination that matches the character it wants to transmit.

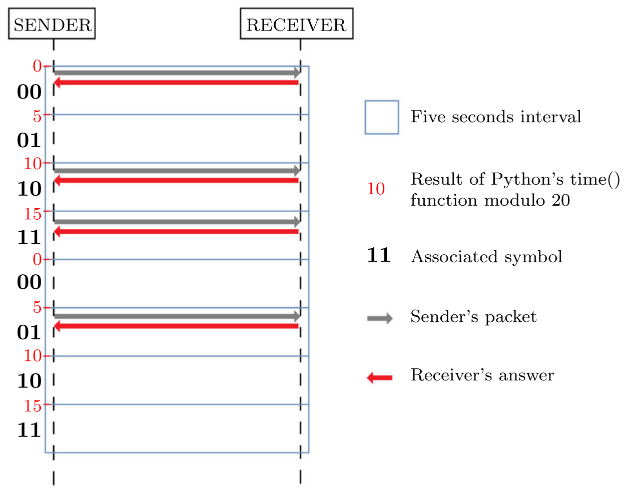


Fig. 15 Visual representation of our timing-based alternative method. In this example, the Sender uses four packets to transmit the string 00101101

For example, assume the attacker controls address A (representing 0) and address B (representing 1): to send the string 10010011, the Covert Sender shall contact B, A, A, B, A, A, B, B in this exact order.

Another alternative consists of a timing-based method revolving around cyclic transmission intervals determined using modular arithmetic (refer to Fig. 15 for a visual representation). In this context, we imagine each 20-s time window to be divided into four 5-s 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 transmit a packet to the Receiver.

Otherwise, it shall first wait an appropriate amount of time 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 s to account for possible latency issues).

One final alternative consists of trying to hide the secret message in the traceroute option of the IP header. This steganography method (initially introduced by Trabelsi and Jawhar [27]) has already been illustrated in Sect. 2.1.1, 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. Suppose, by any chance, a transmission utilising this method is successful. In that case, 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).

Funding Open access funding provided by Università degli Studi di Perugia within the CRUI-CARE Agreement.

Research Data Policy and Data Availability Statements No particular dataset was used in this research.

Declarations

Conflict of interest We declare we have no competing interests as defined by Springer or other interests that might be perceived to influence the results and discussion reported in this paper.

Ethical Standards Stefano Bistarelli and Francesco Santini were supported by project "Progetto Fondo Ricerca Ateneo WP3.2 - Fighting Cybercrime with OSINT (FICO)", project "Progetto Fondo Ricerca Ateneo WP4.1 - Argomentazione Astratta, Text Mining e Network Analysis per il Supporto alle Decisioni (RATIONALISTS)", and project "Progetto Fondo Ricerca Ateneo - Tecnologie Blockchain avanzate per una supply chain alimentare di cibi sani e affidabili (BLOCKCHAIN4FOODCHAIN)", financed by the University of Perugia. The same two authors were also supported by project "VITALITY" funded by MUR-Italy (cup: J97G22000170005), and project "Empowering Public Interest Communication with Argumentation - (EPICA)", funded by MUR-Italy (cup: J53D23007220006).

References

1. Bedi, P., Dua, A.: Network steganography using the overflow field of timestamp option in an IPv4 packet. Procedia Computer Science **171**, 1810–1818 (2020)
2. Bistarelli, S., Ceccarelli, M., Luchini, C., Mercanti, I., Santini, F.: A survey of steganography tools at layers 2-4 and HTTP. In: ARES, pp. 81:1–81:9. ACM (2023)
3. Chaum, D.: Untraceable electronic mail, return addresses and digital pseudonyms. In: Secure Electronic Voting, volume 7 of *Advances in Information Security*, pp. 211–219. Springer (2003)
4. Dingledine, R., Mathewson, N., Syverson, P.F.: Tor: The second-generation onion router. In: USENIX Security Symposium, pp. 303–320. USENIX (2004)
5. Dorrendorf, L., Gutterman, Z., Pinkas, B.: Cryptanalysis of the random number generator of the windows operating system. ACM Trans. Inf. Syst. Secur. **13**(1), 10:1-10:32 (2009)
6. Eddy, W.M.: Transmission control protocol (TCP). RFC **9293**, 1–98 (2022)
7. Ferguson, N.: The Windows 10 random number generation infrastructure. Technical report, Microsoft (2019)
8. Ganivev, A., Mavlonov, O., Turdibekov, B., Uzoqova, M.: Improving data hiding methods in network steganography based on packet header manipulation. In: International Conference on Information Science and Communications Technologies (ICISCT) (2021)
9. Giffin, J., Greenstadt, R., Litwack, P., Tibbetts, R.: Covert messaging through TCP timestamps. In: Dingledine, R., Syverson, P.F. (eds.) Privacy Enhancing Technologies, Second International Workshop, PET, volume 2482 of *LNCS*, pp. 194–208. Springer, New York (2002)
10. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: Wallach, D.S. (ed.) 10th USENIX Security Symposium. USENIX (2001)
11. Hintz, D. Covert channels in TCP and IP headers. In: DefCon 10, Las Vegas, Nevada (2003)
12. Iglesias, I.F., Meghdouri, F., Annessi, R., Zseby, T.: CCgen: injecting covert channels into network traffic. Secur. Commun. Netw. **2022**, 05 (2022)
13. Jacobson, V., Braden, R.T., Borman, D.A.: TCP extensions for high performance. RFC **1323**, 1–37 (1992)
14. Klein, A.: Subverting stateful firewalls with protocol states (extended version). *CoRR*, arXiv:2112.09604 (2021)
15. Kundur, D., Texas: Practical internet steganography: data hiding in IP. In: Proceedings of Texas wksp. security of information systems (2003)
16. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
17. Lubacz, J., Mazurczyk, W., Szczypiorski, K.: Principles and overview of network steganography. IEEE Commun. Mag. **52**(5), 225–229 (2014)
18. Mazurczyk, W., Szczypiorski, K.: Evaluation of steganographic methods for oversized IP packets. Telecommun. Syst. **49**(2), 207–217 (2012)
19. Mazurczyk, W., Wendzel, S., Villares, I.A., Szczypiorski, K.: On importance of steganographic cost for network steganography. Secur. Commun. Netw. **9**(8), 781–790 (2016)
20. Mileva, A., Panajotov, B.: Covert channels in TCP/IP protocol stack—extended version. Central Eur. J. Comput. Sci. **4**(2), 45–66 (2014)
21. Murdoch, S.J., Lewis, S.: Embedding covert channels into TCP/IP. In: Barni, M., Herrera-Joancomartí, J., Katzenbeisser, S., Pérez-González, F. (eds.) Information Hiding, 7th International Workshop, IH, volume 3727 of *LNCS*, pp. 247–261. Springer, New York (2005)
22. Pfitzmann, A., Köhntopp, M.: Anonymity, unobservability, and pseudonymity—a proposal for terminology. In: Workshop on Design Issues in Anonymity and Unobservability, volume 2009 of Lecture Notes in Computer Science, pp. 1–9. Springer (2000)
23. Rowland, C.H.: Covert channels in the TCP/IP protocol suite. First Monday **2**(5) (1997). https://firstmonday.org/ojs/index.php/fm/article/view/528
24. Sharma, K., Sharma, A.: High bandwidth covert channel using TCP-IP packet header. https://www.researchgate.net/publication/312170838_High_Bandwidth_Covert_Channel_using_TCP-IP_Packet_Header, 02 2016. Accessed 17 Dec 2023
25. Gustavus, J.: Simmons. The prisoners' problem and the subliminal channel. In: Chaum, D. (ed.) Advances in Cryptology. Proceedings of CRYPTO, pp. 51–67. Plenum Press, New York (1983)
26. Tommasi, F., Catalano, C., Caniglia, A., Taurino, I.: COTIIP: a new covert channel based on incomplete ip packets. In: 2022 7th International Conference on Smart and Sustainable Technologies (SpliTech) (2022)
27. Trabelsi, Z., Jawhar, I.: Covert file transfer protocol based on the IP record route option. J. Inf. Assur. Secur. (JIAS) **5**, 01 (2010)
28. Wendzel, S., Caviglione, L., Mazurczyk, W., Mileva, A., Dittmann, J., Krätzer, C., Lamshöft, K., Vielhauer, C., Hartmann, L., Keller, J., Neubert, T.: A revised taxonomy of steganography embedding patterns. In: Reinhardt, D., Müller, T. (eds.) ARES 2021: The 16th

International Conference on Availability, Reliability and Security, p. 67:1-67:12. ACM, New York (2021)

29. Zander, S., Armitage, G., Branch, P.: Covert channels in the IP time to live field. In: Australian Telecommunication Networks and Application Conference (ATNAC) (2006)