# Design and implementation of a secret communication system based on TCP retransmission steganography

Javier Hospital Poncell

**Abstract–** This paper presents the design, implementation and experimental results of a communication system based on a network steganography method known as retransmission steganography (RSTEG). This technique, proposed by Mazurczyk et al. in 2009, is aimed at many network protocols that use a retransmission mechanism. Essentially, RSTEG works by not acknowledging a well-received package in order to invoke a retransmission, whose payload data will be exchanged by a steganogram. A client/server architecture has been built with a simple TCP implementation based on retransmission time-outs (RTO) that includes the RSTEG functionality. The application is able to communicate using an HTTP implementation built on top of it. The performed experiments evaluate the steganographic bandwidth and detectability of the implemented method. An average bandwidth of 6.4 kB/s was achieved for a 5% of retransmission probability on a 132 kB/s TCP throughput.

**Keywords–** information hiding, network steganography, RSTEG, TCP

**Resum–** Aquest article presenta el disseny, implementació i resultats experimentals d'un sistema de comunicació basat en el mètode d'esteganografia de xarxa conegut com a esteganografia de les retransmissions (RSTEG). Aquesta tècnica, proposada l'any 2009 per Mazurczyk et al., està enfocada a protocols de xarxa que utilitzen retransmissions. Concretament, RSTEG consisteix a no confirmar un paquet rebut per tal de forçar la seva retransmissió. Aquest paquet retransmès haurà substituït les dades d'usuari per un esteganograma. S'ha construït una arquitectura client/servidor mitjançant una implementació senzilla de TCP que inclou aquesta funcionalitat. L'aplicació es comunica mitjançant una implementació d'HTTP construïda a sobre. S'han realitzat experiments per tal d'avaluar la capacitat i la detectabilitat del mètode implementat, amb una banda ampla de 6,4 kB/s de mitjana per un 5% de probabilitat de retransmissió sobre una connexió TCP a 132 kB/s.

**Paraules clau–** esteganografia de xarxa, ocultació d'informació, RSTEG, TCP

## 1 INTRODUCTION

STEGANOGRAPHY is commonly defined as a set of techniques that undetectably alter a work (also known as cover) in order to embed a secret message. In particular, the term network steganography, first introduced in 2003 [1], encompasses the methods that embed secret data in a network communication without any third-party being aware of it. The concept of covert channels is a closely related topic, since they usually utilize similar techniques in order to establish a communication channel in a way it was not intended by design. Many applications have been devised for covert channels, both legitimate and illegitimate, such as censorship circumvention, whistleblowing, confidential information leakage or malware communication control.

Thus, these methods have attracted the attention of security researchers for a long time, since their analysis provides valuable information for improving network monitoring and malware detection. The use of static or dynamic analysis carried out by Network Intrusion Detection Systems (NIDS) has become an industry standard and as a result detection avoidance methods are a growing topic. For example, in 2013, the Fokirtor Trojan [2] was found using the SSH protocol as a covert channel for command

• E-mail: javier.hospital@e-campus.uab.cat
• Specialization: Information Technologies
• Work supervised by: David Megías Jiménez (DEIC)
• Academic Year 2020/21

and control while evading detection. In a similar way, another trojan known as Regin was found using several methods at once [3], such as embedding control commands into ICMP packets, HTTP cookies and TCP/UDP Protocol Data Units (PDUs).

In general, network steganography methods can be classified in the following taxonomy (Fig. 1):

- **Alteration of the Protocol Data Unit** (PDU), such as protocol headers or payload fields. An example for these is the HICCUPS technique [4], which works at layer 2 of the OSI stack and consists of deliberately sending MAC frames with bad checksum values. Those machines that are not part of the system reject those frames while the others accept them and thus use them to maintain a covert communication. Another technique known as YARSTEG is based on the same idea while using TCP [5].
- **Modification of the PDU stream** by altering the sequence order or introducing intentional losses or time delays. For example, IP fragmentation steganography works by splitting and arranging fragmented packets in such a way that a secret fragment can be hidden between them [6].
- **Hybrid methods** that modify PDU data and structure. Here, we can find methods that are a combination of the previous categories. An example for these are RSTEG or the Lost Audio pACKets steganography (LACK) method [7], that are applied to the TCP and VoIP protocols respectively.

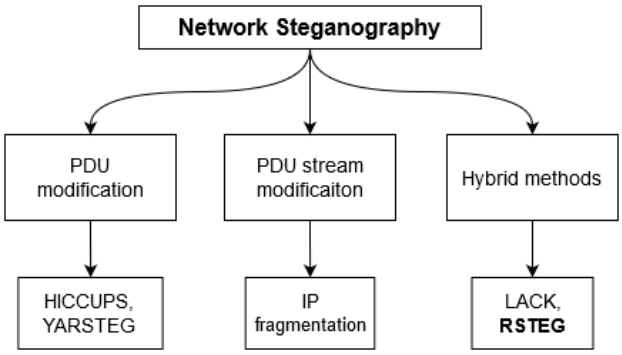



Fig. 1: Network steganography taxonomy with examples.

In this paper, we create a secret communication system using the hybrid method known as retransmission steganography (RSTEG) [8]. Specifically, we develop a simple implementation of the Transmission Control Protocol (TCP) with RSTEG running over it. Then, we build an HTTP client-server architecture on top of that. The client and server applications work as a typical REST web service, while also being able to transfer secret data with the underlying RSTEG mechanism. The system has been developed with Python 3.8 and the Scapy library for packet manipulation and decoding [9].

The rest of this work is organized as follows. Section 2 explains the RSTEG in detail. Section 3 presents the system architecture design. Next, Section 4 details the implementation of the system. Then, Section 5 presents the experimental methodology and results. Finally, Section 6 concludes this article.

## 2 The RSTEG method

In a typical time-out retranmission mechanism, the sender starts a countdown after sending each packet, which will then be reset upon receipt of the receiver's acknowledgment; if the countdown expires, the packet retransmission is triggered. RSTEG takes advantage of this mechanism by making the receiver not acknowledge a successfully received packet in order to expire the sender's time-out and, consequently, trigger a retransmission. Then, the sender, knowingly modifies the packet payload data and embeds the secret that will be sent as a retransmission. In order for this to be successful, both the sender and receiver must be aware of this exchange, thus being able to reliable transmit the secret without disrupting the cover data.

For example, as shown in Fig. 2, an expected RSTEG exchange using TCP would be as follows: the client establishes a connection with the server in order to transmit a file, which will act as a cover. Then, the client proceeds to send the file, split into several segments. At some point during the data transfer, the client will signal a segment such that the server knows that an RSTEG exchange should follow. Upon its reception, the server does not issue the expected acknowledge segment and waits for the retransmitted segment with the secret payload. Finally, the server receives and acknowledges the secret and the connection proceeds as usual until the connection closure.

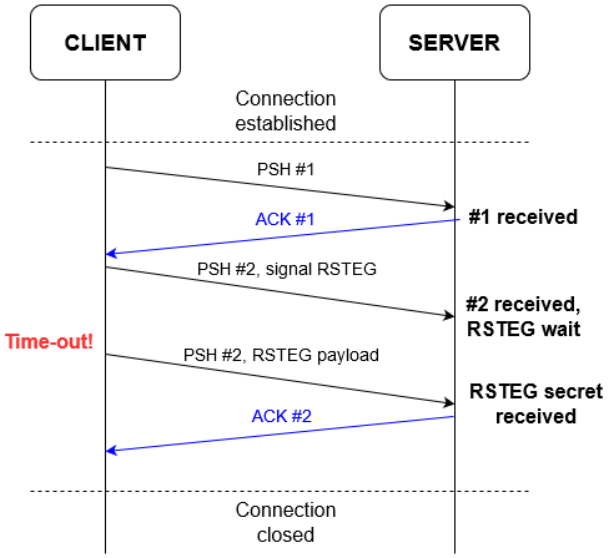



Fig. 2: RSTEG idea applied to a TCP stream.

However, the RSTEG idea presents several challenges to address, such as how would the sender signal the receiver, or what would happen if the segment with the signal or secret payload is lost. Moreover, this method introduces artificial retransmissions that impact the connection throughput and they are also a detection key factor. We address these

issues in more detail in the following sections.

### 2.1 Signaling

There are several methods for RSTEG to signal an exchange. A straightforward way would be to choose a PDU header field and modify its value as needed. Since we only need one bit to distinguish between a normal packet or a signaling one, there are several fields we could use. For instance, when using the IPv4 protocol, we could modify fields such as the Don't Fragment or the Type of Service fields, or we could also use the header padding bits as a signal flag. However, this way would be relatively easy to identify by comparing headers from the same stream and a procedure to modify the headers must also be implemented for the sender.

In order to avoid this, another procedure would be to embed the signal flag into the payload data. The original RSTEG paper proposes the identifying sequence for marking a TCP segment as described in Expression (1). The $IS$ is computed using a cryptographic hash function using the following parameters: $SK$ is a shared secret key between the sender and the receiver, followed by the TCP sequence number, the segment checksum and finally, the signal flag bit.

$$IS = H(SK + Seq.No. + TCPChecksum + SignalingBit \quad (1)$$

Once computed, the sender inserts the $IS$ into the payload data in a predefined position. The receiver will check that position in every payload and compare the received data versus the computed sequence.

### 2.2 Packet loss

Since RSTEG modifies the retransmission mechanism, some edge cases must be addressed regarding packet loss and the subsequent retransmissions:

- Loss of the signal packet: the sender packet marked with the signal flag is lost due to the network context but, given the RSTEG behavior, it is unable to discern this loss from the expected lack of acknowledgment. Thus, in this scenario, the receiver will read an unexpected packet containing the steganogram. To avoid the disruption of the connection, the receiver should not issue an acknowledgment until he receives the lost signal packet. As a result, the sender will not receive the expected confirmation and realize that there is a packet loss situation, at which point the sender will retransmit the signal packet until it receives its acknowledgment.
- Loss of the steganogram: this scenario is very similar to the previous one, since the sender is not able to recognize which packet was lost (the signal or the steganogram). First, the sender will try to retransmit the signal packet in case it was lost, but, in this situation, the receiver should not issue an acknowledgment. Therefore, the sender should retransmit the steganogram until its reception is acknowledged.

### 2.3 Detection

Since retransmissions are intrinsic to the network protocols where we apply RSTEG, the addition of artificial retransmissions should not be a major detection vector if they are kept below a reasonable threshold. If possible, the sender should analyze the target network for the average retransmission rate to select the appropriate retransmission probability ($R_P$) that blends the connection with the network background traffic. Note that the selected $R_P$ value will also influence the data throughput for both secret and cover streams.

In addition to retransmissions, the modified packet checksum could be another detection vector for RSTEG. During the insertion procedure, the payload data is swapped by the steganogram but the header is left unmodified. Thus, a checksum validation would not match the one from the original packet. Despite this, there are ways around this issue. For instance, in TCP, the checksum can be made to match a desired value with the addition of a compensation code in the steganogram data. A more detailed overview on this follows in Section 3.

The major weakness of RSTEG against detection are passive wardens that implement any mechanism of payload comparison between retransmitted segments. Although simple and effective, this detection method struggles with performance on high-traffic networks where all streams must be monitored. Since a representation of the last observed payload has to be kept in memory for future comparison, the memory requirements can be high for a typical network device.

## 3 System Architecture

A client/server architecture has been designed and implemented in order to illustrate the potential of RSTEG applied to TCP in a fairly common configuration. As illustrated in Fig. 3, the system is composed of the following modules:

- **RstegTCP**: includes all the transport layer logic. It is in charge of accessing the network layer and contains the essential TCP logic with the added RSTEG behavior and data buffers. It offers a set of primitives so other modules can establish and manage a TCP connection.
- **RstegSocket**: a wrapper that encapsulates the primitives from RstegTCP and offers a socket-like interface similar to other libraries found in many programming languages. Besides calls such as Open, Bind or Receive, this module offers an rSend call that manages the cover and secret data, together with the retransmission probability provided by the user.
- **Custom HTTP client/server**: a simple HTTP implementation built over the RstegSocket module. The server offers a REST API with root and '/upload' endpoints that reply to GET and POST requests. The latter endpoint is capable of storing the data received from requests in disk. In particular, the server not only stores normal POST data, but also the secret data received from the RSTEG method, if any.

- **GUI**: a user interface that manages the client application. With this, the user is able to select the configuration parameters, such as IP addresses, ports, URLs, and so on. Also, the user is able to select a connection for RSTEG and input the desired retransmission probability. In addition, the module outputs the log registry to the user and can show the HTML responses using the local browser.

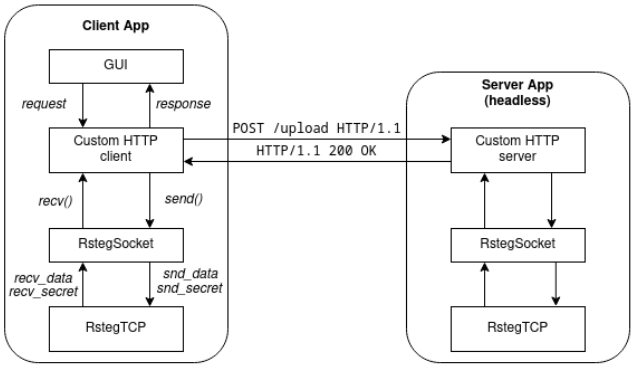



Fig. 3: Overview of the implemented communication system using RSTEG.

## 4 IMPLEMENTATION

### 4.1 RstegTCP

This module is built around a Python class of the same name which uses an L3RawSocket object from the Scapy library in order to send and receive datagrams from the network layer. The library is also used to build the packet data structures, such as IP or TCP headers. The TCP implementation follows the RFC 793 specification but is far from being as extensive and efficient as current implementations. That would be beyond the scope of this work and, consequently, functionalities such as the sliding window, cumulative ACKs, and congestion avoidance mechanisms are not implemented. In order to use the L3 layer, RstegTCP uses its method start to initialize all class parameters and spawn a thread that polls the L3 socket for datagrams including a TCP segment with configured destination. This thread will keep polling the network layer until the internal state is switched to "closed".

Upon reception of a valid datagram, the thread calls the handle_packet method with the packet as a parameter. This method selects the appropriate handler function for the packet, according to the class internal state and the TCP flag of the received segment. This is carried out following the connection state specifications from the RFC 793. Next, the handler function parses the received segment, update the internal parameters, and send the suitable response segment. In order to initiate or terminate connections, as well as to manage the data transfer, the methods connect, close and send_data are implemented.

To implement the RSTEG functionalities, several additions have been made. Firstly, we compute and append the $IS$ to the payload while it is being assembled in order to invoke the artificial retransmission. The sequence is digested with a SHA-256 hash function, thus resulting in a 32-byte hash. The payload extraction procedure is aware of this and knowing the sequence length and position it can successfully extract and evaluate it. Note that the user data length must be reduced in order to fit the sequence and comply with the maximum segment size. Secondly, while assembling the artificial retransmitted segment, if the steganogram length is less than the original payload length, padding must be added to maintain the same segment size. Thirdly, to manage the steganogram data, a second data buffer is created, parallel to the TCP data buffer. This makes it easier to manage and receive the secret data across several retransmitted segments.

**Algorithm 1** Find compensation code

**Parameters**: Old payload ($oP$) and new payload ($nP$)
**function** FINDCOMPENSATION($oP, nP$)
  **if** $len(oP)\%2 == 1$ **then**
    $oP += b$"0"
  **end if**
  **if** $len(nP)\%2 == 1$ **then**
    $nP += b$"0"
  **end if**
  $o\_sum = sum(oP)$
  $n\_sum = sum(nP)$
  $comp = o\_sum - n\_sum$
  $comp = (comp >> 16) + (comp$ & 0xffff$)$
  $comp \mathrel{+}= comp >> 16$
  Return $comp$

Finally, when the artificial retransmission is triggered and the steganogram is assembled, the checksum must be recomputed to match the original value. Essentially, the TCP checksum is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header and payload. Thus, a 16-bit compensation word can be computed and appended to the new payload so as to obtain the exact same checksum value from the original payload (Algorithm 1).

### 4.2 RstegSocket

The RstegSocket class encapsulates an RstegTCP object and builds a set of methods that make its use easier for building application protocols. These methods are inspired by the socket interfaces in the standard library of Python and other high-level languages. The RstegSocket provides the following methods:

- **Bind (host, port)**: configures the RstegSocket class to use the supplied host and port values.
- **Listen**: starts the underlying module using the binded values.
- **Accept**: the RstegTCP thread waits until a three-way handshake is performed and the connection is established.
- **Connect (host, port)**: starts the module and initiates a three-way handshake with the indicated host on the supplied port.

- **Send (data)**: slices the data according to the Maximum Segment Size (MSS) and sends it to the send_data method in RstegTCP.
- **rSend (cover, secret)**: first, it slices the cover and secret data. Then, the artificial retransmission trigger is evaluated using the retransmission probability in the socket configuration. When the trigger is activated, the cover slice is marked for retransmission and, after timeout, the secret slice is sent to the send_secret method.
- **Recv**: polls the RstegTCP data and secret buffer searching for new received data. Returns a two-position array for new cover and secret data.
- **Close**: closes the TCP connection and stops the module, resetting its state and buffers.

### 4.3 Custom HTTP client/server

This module implements a simple HTTP 1.1 client/server using the RstegSocket methods. This version has been selected because it is still prevalent in the Internet and the Scapy library offers the data structures for the requests and response headers. The server follows a REST architecture as many web services do nowadays, while the client simply offers a straightforward method to make requests and return their responses. Two HTML resources are exposed by the / and /upload endpoints, giving clients the option of storing data (with or without the RSTEG mechanism) through POST requests. The HTTP server only provides partial functionality when using browsers as clients due to the lack of more advanced features of the TCP protocol that are not implemented in the underlying module.

### 4.4 GUI

A graphical user interface has been added to facilitate the use of the client application. The interface is implemented with the PySimpleGUI wrapper [10], using the Qt library for cross-platform user interfaces.

The interface frame is divided into 3 sections (Fig. 4). Firstly, a form gives the users the possibility to choose which layer do they want to communicate with (raw TCP or HTTP). Depending on their choice, the second section is rendered below with the suitable form and input parameters, such as IP addresses, ports, URL, HTTP method, etc. For instance, while using the TCP or HTTP POST form, a subform that lets the user pick a cover and secret data from the disk appears. Other parameters such as the retransmission probability or a checkbox for setting up the RSTEG mechanism are also present. Lastly, the third section is rendered as a text screen where the connection status and application log are shown. In addition, two buttons are rendered below the third section. The "Submit" button verifies the form parameters are correct and starts the user connection. Finally, the "Clear Log" resets the text screen from the third section. Also, the interface can output the received raw HTTP responses in a pop-up window or it will try to open the systems default browser if an HTML response is detected.

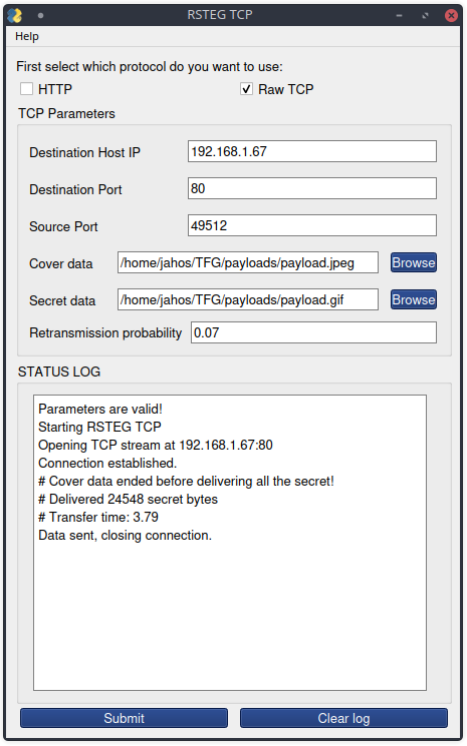



Fig. 4: GUI after an RSTEG connection using TCP.

## 5 Experimentation

Three tests have been devised in order to evaluate the implemented RSTEG mechanism. They consist on the following aspects: measuring the steganographic capacity, testing its detectability, and also a theoretical comparison with another state-of-the-art steganographic method based on digital imagery.

### 5.1 Steganographic Capacity

#### *5.1.1* Methodology

The steganographic capacity can be defined as the maximum number of bits a method can embed on a determined cover for secret communication purposes. In particular, we evaluate the RSTEG capacity with the steganographic bandwidth ($S_B$) parameter. The $S_B$ shows the transferred amount of steganographic data on a determined RSTEG connection. This parameter is expressed in bits per second and can be calculated using the following Expression (2), where $N_s$ is the number of retransmitted segments with an steganogram, $S_s$ is the payload data size, and $T$ is the connection time length in seconds. Other relevant parameters are the retransmission probability ($R_P$) and the TCP cover connection throughput ($TCP_T$).

$$S_B = \frac{N_s \times S_s}{T}[Bps] \tag{2}$$

Two experimental scenarios with different network topologies have been constructed in order to evaluate $S_B$ in different implementations (Fig. 5). On the one hand, the scenario A is an Ethernet LAN with low latency and low background traffic, whereas the scenario B is an Internet connection with a geographically remote host and high latency. On both scenarios, a file will be transmitted between

the client and the server, acting as the cover data. The secret data will be sent with the RSTEG mechanism as long as the cover is still being transmitted. For both scenarios, the average $S_B$ for ten connections is computed for $R_P$ values between 0 and 10%.

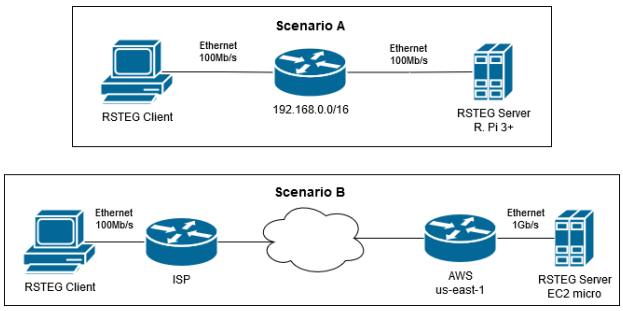



Fig. 5: Experimental network topologies.

| Scenario | File size | MTU | RTT | Hardware |
|---|---|---|---|---|
| A | 1,2 MB | 1500 B | 0,50 ms | R. Pi 3+ |
| B | 220 KB | 1500 B | 150 ms | EC2 t2.micro |

TABLE 1: EXPERIMENTAL PARAMETERS

### 5.1.2 Results

As expected, both scenarios show how the $S_B$ grows as we increase the $R_P$ value. With a higher retransmission probability, more segments with a secret payload are sent, but this also increases the detection probability. Thus, the $R_P$ value is a key parameter to maintain a balance between capacity and detectability. At the same time, we observe how the cover connection throughput degrades as a result of the increased artificial retransmissions and secret transfer. The maximum $TCP_T$ achieved without the RSTEG mechanism was of 132 kB/s on average. This is a modest result considering that the implemented TCP version is not as efficient and extensive as a modern implementation, with many advanced features missing. For instance, the addition of sliding window and window scaling mechanisms would vastly improve the $S_B$ and $TCP_T$ numbers.

In the first scenario (Fig. 6), an $S_B$ of about 6.4 kB/s for an $R_P$ of 5% was measured. This result is six and twelve times faster than other similar RSTEG implementations for the same retransmission percentage [11, 12]. Meanwhile, on the second scenario (Fig. 7), the high latency connection plus the remote host localization made the performance vastly decrease, with an $S_B$ of only 500 B/s for the same $R_P$. The lack of advanced TCP features, coupled with a round-trip time of 150ms on average, are the reasons for this decrease in performance.

## 5.2 Detection with Snort

The previous first scenario was modified in order to deploy a Network Intrusion Detection System (NIDS) capable of testing the detectability of the RSTEG method so as to imitate a real scenario of a monitored network. In particular, we deployed Snort 2.9.17 [13], the latest available version

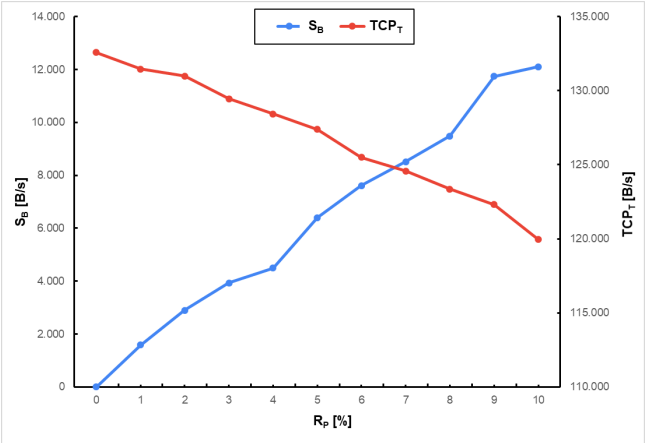



Fig. 6: Steganographic bandwidth ($S_B$) and TCP throughput ($TCP_T$) for scenario A.

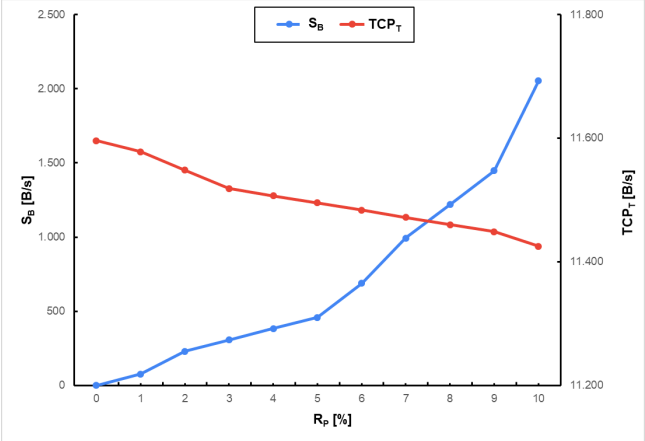



Fig. 7: Steganographic bandwidth ($S_B$) and TCP throughput ($TCP_T$) for scenario B.

at the time of writing. Snort is one of the most widely used open-source solutions for NIDS, commonly found in network devices such as firewalls or hubs. Using the port mirroring technique, we duplicated the RSTEG server Ethernet traffic into the Snort machine.

Snort is built as a collection of modules known as preprocessors and a detection rule database used to perform static analysis. We used the latest available database for registered users and the default Snort configuration values. From the RSTEG point of view, the most interesting preprocessors are Stream5 and Normalize_TCP because these modules are in charge of analyzing the TCP and UDP streams for any anomalies.

The same tests from the previous scenario were repeated, this time with Snort deployed. The logs reported an alert with the 129-5 code for 83% of the segments that included an steganogram. In particular, the 129-5 alert is described as "Bad segment, adjusted size less than 0" and classifies the stream as "potentially bad traffic" with a level 2 priority out of 4 (where 1 is the highest priority). According to the Snort database documentation, alert 129-5 is raised by the Normalize_TCP preprocessor but no further detail or warning regarding its nature is stated. Upon inspecting the preprocessor source code, we confirmed that the

preprocessor option known as IPS performs a payload comparison between retransmitted segments inside a TCP stream, thus defeating the steganographic mechanism.

Despite this, further research showed that some Snort implementations in network devices, such as the firewall software known as Cisco Firepower Management Center, ignore the 125-5 alert by default [14]. Presumably, this could have been done because of false positives being triggered by corrupted segments, an increased performance load for high-traffic networks, or simply a lack of risk awareness due to almost non-existent documentation.

### 5.3 RSTEG vs JPEG steganography

Nowadays, most modern steganographic methods based on digital imagery work by embedding the secret data inside the most complex regions of the image. For JPEG images, the steganogram is embedded inside the quantized DCT coefficients derived from the compression algorithm. However, the quality factor ($QF$) used during the compression algorithm is a key component in order to establish the steganographic capacity and detectability rate.

Despite this, we can compute the capacity of a given image according to the number of alternating non-zero DCT coefficients ($nzAC$) and the amount of bits embedded into them ($bpnzAC$). The exact number of available $nzAC$ depends on the image content and the $QF$ used during the compression phase. According to the study [15], the average nzac on a 5000 image dataset of $512 \times 512$ pixels with $QF = 80\%$ is 67408 $\approx$ 67500. Also, according to [16], the suitable $bpnzAC$ amount to guarantee indetectability on a $QF = 75\%$ image must be 0.1 or lower.

Assuming these values, the nzac per pixel ratio ($R_{nzAC/p}$) for an image compressed with a $QF$ of $80\%$ would be about $0.25$ nzAC/pixel, as shown in Expression (3). Thus, if we have an image with a megapixel resolution, we compress it with the aforementioned $QF$, and embed 0,1 bpnzac, the image capacity ($C_n$) is about 25000 bits (3125 bytes), as detailed in Expression (4). In addition, if we assume a 25:1 compression rate [17], the cover image size would be around 120 kB.

$$R_{nzAC/p} = \frac{67500}{512 * 512} \approx 0.25\ [nzAC/pixel] \quad (3)$$

$$C_n = 0.1 \times R_{nzAC/p} \times (1000000) \approx 25000\ [bits] \quad (4)$$

In order for the previously estimated JPEG capacity to match the implemented RSTEG capacity for a 5% $R_P$, the cover image should be transmitted at a rate of 244800 B/s (Table 2). Considering that the TCP throughput of the RSTEG implementation is about 132 kB/s, this means that if we were to transmit the same steganogram, the RSTEG method is 1.85 times faster than the JPEG method. Note that these results are computed using the implemented TCP version in order to compare both methods under the same conditions. Thus, if we were to use an optimal TCP implementation we would observe a greater throughput value but the 1.85 times difference would persist.

| $R_P$ [%] | $C_{RSTEG}$ [B/s] | Bandwidth for JPEG [B/s] |
|---|---|---|
| 1 | 1593.71 | 62776.22 |
| 2 | 2903.07 | 114153.35 |
| 3 | 3931.40 | 154588.92 |
| 4 | 4497.75 | 176858.72 |
| 5 | 6396.28 | 251511.96 |
| 6 | 7607.27 | 299130.02 |
| 7 | 8512.96 | 334743.20 |

TABLE 2: MEASURED CAPACITY FOR RSTEG ($C_{RSTEG}$) AND THE REQUIRED BANDWIDTH FOR JPEG TO MATCH THE SAME CAPACITY AS RSTEG USING A 80% OF $QF$, 0,1 BPNZAC AND A 25:1 COMPRESSION RATE.

## 6 CONCLUSION

In this work we have designed and created a communication system based on the retransmission steganography method (RSTEG). This method is oriented to network protocols that provide a retransmission mechanism and it works by not acknowledging a successfully received segment in order to trigger an artificial retransmission, whose payload has been replaced by an steganogram. In particular, the TCP protocol has been chosen for this system and thus we have implemented a simple version with the added RSTEG mechanism using Python and the Scapy library. On top of that, a client/server architecture has been built using the modified RstegTCP, wrapped inside a socket-like interface. In the application layer, a simple HTTP 1.1 implementation has been developed using the underlying RSTEG socket, which it has been used as a REST service on the server application. In addition, the client application has been enhanced with a graphical user interface such that the user can test several network parameters and use the REST service while performing RSTEG exchanges.

The experimental results show a steganographic bandwidth of 6.4 kB/s and 8.5 kB/s for a 5% and 7% of retransmission probability respectively, which is widely superior than in other published implementations of RSTEG. The implemented TCP connection measured a throughput of only 132 kB/s. This is due to many modern functionalities and optimizations of a typical TCP implementation which were out of the scope for this work, such as the sliding window, the window scaling options, cumulative ACKs and congestion avoidance mechanisms. As future work, it would be interesting to consider a much more in-depth implementation of RSTEG with these other features of TCP.

In addition, we tested the detectability of the implementation against a Network Intrusion and Detection System (NIDS). For this, we deployed the latest version of Snort, a widely used open-source NIDS solution. Our tests show that a preprocessor known as the TCP normalization engine is able to raise an alert for the majority of artificial

retransmitted segments, as it performs a payload comparison between the newer segment and the original one. In particular, the preprocessor flags the segments as bad traffic and raises the alert 129-5, which is barely documented in the Snort alert database. It should be noted that we found some network devices implementing Snort that have this specific alert ignored by default.

Finally, we performed a theoretical comparison of RSTEG against another steganographic method based on JPEG images, while using parameters that guarantee a good level of indetectability. We calculated the theoretical capacity of the JPEG method and its bandwidth equivalent to match the implemented method. The results pointed out that, for the same steganogram data and connection throughput, RSTEG is 1.85 times faster than its counterpart.

## ACKNOWLEDGMENT

The author gratefully acknowledge the invaluable help and experience of D. Megías, who proposed and supervised this work. I would also wish to thank the extraordinary effort performed by all the University members during the COVID-19 pandemic.

## REFERENCES

[1] J. Lubacz, W. Mazurczyk, and K. Szczypiorski, "Principles and overview of network steganography," *Communications Magazine*, vol. 52, no. 5, pp. 225-229, 2014.

[2] B. Prince, "Attackers hide communication with linux backdoor," Security Week. https://www.securityweek.com/attackers-hide-communication-linux-backdoor (accessed Jan. 8, 2021).

[3] A. L. Johnson, "Top-tier espionage tool enables stealthy surveillance," Symantec Enterprise. https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=b2a9256f-0486-4819-a833-a7260d7b77e8& (accessed Jan. 8, 2021).

[4] K. Szczypiorski, "HICCUPS: hidden communication system for corrupted networks," in *Proc. of ACS 2003*, Miedzyzdroje, Poland, Oct. 2003, pp. 31-40.

[5] A. M. Brodzki and J. Bieniasz, "Yet Another Network Steganography Technique Basedon TCP Retransmissions," in *5th International Conference on Frontiers of Signal Processing (ICFSP)*, Marseille, France, 2019, pp. 35-39, doi: 10.1109/ICFSP48124.2019.8938085.

[6] R. M. Goudar, S. J. Wagh, and M. D. Goudar, "Secure data transmission using steganography based data hiding in TCP/IP," in *Proceedings of the International Conference & Workshop on Emerging Trends in Technology (ICWET '11)*, Association for Computing Machinery, New York, NY, USA, 974–979. doi:https://doi.org/10.1145/1980022.1980233

[7] W. Mazurczyk, J. Lubacz, K. Szczypiorski, "Hiding data in VoIP," *In Proceedings of the 26th army science conference (ASC 2008)*, Orlando, Florida, USA, Dec. 2008.

[8] M. Smolarczyk W. Mazurczyk and K. Szczypiorski, "Retransmission steganography and its detection," *Soft Computing*, vol. 15, no.3, pp. 505-515, Nov. 2009, doi: 10.1007/S00500-009-0530-1.

[9] P. Biondi, "Scapy, Packet crafting for Python2 and Python3," Scapy.net. https://scapy.net/ (accessed Jan. 9, 2021).

[10] The PySimpleGUI Organization, "PySimpleGUI Documentation," PySimpleGUI.readthedocs.io. https://pysimplegui.readthedocs.io/en/latest/ (accessed Jan. 9, 2021).

[11] W. Mazurczyk, M. Smolarczyk and K. Szczypiorski, "Retransmission Steganography Applied," *International Conference on Multimedia Information Networking and Security*, Nanjing, Jiangsu, 2010, pp. 846-850, doi: 10.1109/MINES.2010.179.

[12] J. Zhai, G. Liu and Y. Dai, "An Improved Retransmission Steganography and Its Detection Algorithm," *Third International Conference on Multimedia Information Networking and Security*, Shanghai, 2011, pp. 628-632, doi: 10.1109/MINES.2011.103.

[13] Snort, "Snort - Network Intrustion Detection & Prevention System", Snort.org. https://www.snort.org/ (accessed Jan. 9, 2021).

[14] Cisco Systems, "Inline normalization preprocessor - Cisco Firepower Management Center," Cisco.com. https://www.cisco.com/c/en/us/support/docs/security/firesight-management-center/117927-technote-firesight-00.html (accessed Jan. 9, 2021).

[15] H. Fangjun, J. K. Hyoung and Z. Dong, "Efficiency of Frequency Selection in JPEG Steganography," in *Proceedings of 3rd International Conference on Multimedia Technology (ICMT-13)*, 2013, pp. 1809-1816, doi: 10.2991/ICMT-13.2013.219.

[16] T. D. Denemark, M. Boroumand and J. Fridrich, "Steganalysis features for content-adaptive jpeg steganography," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 8, pp: 1736-1746, doi: 10.1109/TIFS.2016.2555281.

[17] Graphics Mill, "Compression ratio for different JPEG quality values," GraphicsMill.com. https://www.graphicsmill.com/blog/2014/11/06/Compression-ratio-for-different-JPEG-quality-values (accessed Jan. 10, 2021).

# APPENDIX

## A.1 Experimental results summary

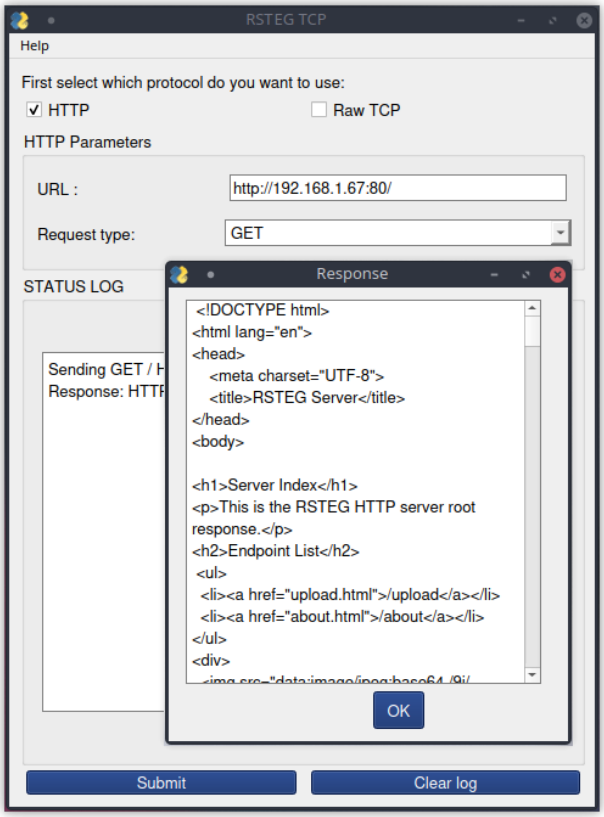
| $R_P$ [%] | $TCP_T$ [B/s] | $S_B$ [B/s] |
|---|---|---|
| 0 | 132574,08 | 0 |
| 1 | 131459,53 | 1593,71 |
| 2 | 130977,67 | 2903,07 |
| 3 | 129453,50 | 3931,40 |
| 4 | 128424,30 | 4497,75 |
| 5 | 127397,57 | 6396,28 |
| 6 | 125499,36 | 7607,27 |
| 7 | 124584,46 | 8512,96 |
| 8 | 123359,34 | 9486,68 |
| 9 | 122322,82 | 11743,73 |
| 10 | 119958,49 | 12104,33 |

TABLE 3: RESULTS FOR SCENARIO A.

| $R_P$ [%] | $TCP_T$ [B/s] | $S_B$ [B/s] |
|---|---|---|
| 0 | 11595,78 | 0 |
| 1 | 11577,91 | 77,04 |
| 2 | 11548,26 | 230,53 |
| 3 | 11518,75 | 306,59 |
| 4 | 11506,99 | 382,85 |
| 5 | 11495,26 | 458,95 |
| 6 | 11483,55 | 687,72 |
| 7 | 11471,86 | 992,37 |
| 8 | 11460,19 | 1220,13 |
| 9 | 11448,55 | 1447,44 |
| 10 | 11425,34 | 2052,71 |

TABLE 4: RESULTS FOR SCENARIO B.

## A.2 User interface



## A.3 UML class diagram