Superimposing Permutational Covert Channels onto Reliable Stream Protocols

Jamie Levy
Dept. of Math. & Comp. Science
John Jay College, CUNY
New York, NY 10019, U.S.A.
jamie.levy@jjay.cuny.edu

Jaroslaw Paduch
jpaduch@jjay.cuny.edu

Bilal Khan
bkhan@jjay.cuny.edu

Abstract

In this paper, we present a new implicit encoding technique that makes use of lower-layer packet reordering to superimpose covert messages onto a reliable data stream. In particular, since the TCP layer provides a reliable in-order data stream over the unreliable network layer's IP datagram service, we can encode covert messages by artificially permuting IP packets before they leave the source and reading the permutation at the destination prior to delivering the payload to TCP. Applying such permutations will not adversely affect TCP's ability to reconstitute the transport layer data stream, since TCP is designed to be robust against out of order network layer packet delivery. We describe the design and operation of PERMEATE*, an open-source covert channel toolkit which implements such a permutational covert channel over TCP, and we provide a quantitative assessment of it's efficacy and efficiency as a covert channel.*

1. Introduction

A covert channel is a mechanism for steganographically superimposing illegitimate data onto a legitimate network data stream. Although the illegitimate message is sent unencrypted, it remains unnoticed since it is being carried "inside" a legitimate flow in such a manner that it does not essentially alter the semantics of the legitimate data when it arrives at the intended recipient. To date, most covert channels devised can be placed in one of two broad categories: they either (i) *explicitly* encode secret information in unused portions of packet headers, or (ii) *implicitly* encode secret information in inter-packet timing intervals.

- Explicit encodings (storage channels) are more commonly seen in literature and are easier to implement. The basic strategy is to embed covert information into fields that are either not used or can be readily changed with little damage to packet processing and semantics. To encode the covert message, appropriate bits are changed in the packet headers; to decode, these bits are read off the header fields at the destination. Several implementations of storage channels built using TCP, IP and UDP headers are readily available on the web [1, 2, 3, 4, 5, 6, 7]. There are also implementations that use ICMP [4, 8, 9, 10, 11], HTTP [12, 13, 14, 15], DNS [16, 17] and MSN [18] protocols.

- Implicit encodings (timing channels) convey a message based on the time in between successive packet transmissions. According to the "Orange Book" [19], a timing channel is possible whenever "one process is allowed to signal information to a second process by modulating its own use of system resources" . For example, one design might follow Morse code: three packets sent across the wire in a short amount of time might be conveyed as an 'S' (dot dot dot), while sending out three more packets spaced out with a larger amount of time in between them might convey the message 'O' (dash dash dash), etc. In order for timing channels to be successful, both parties must be operating in a steady state with respect to network and CPU load, since otherwise abrupt changes in the traffic environment will be yield errors in the covert channel. More problematically, channels error rates degrade when instrumented in the wide area, since traffic shaping/multiplexing at intermediate (network layer) routers has the effect of reducing the variance of inter-packet time distributions. Thus, in order for the bimodal distribution of interpacket timings to remain separable at the destination, one requires extreme differences in the interpacket timings used to encode covert bits. This endows the shape of the legitimate traffic with a very low entropy, making it appear bursty

and suspicious, and drawing attention to the possible presence of a covert channel. Timing channel implementations are not as readily available as storage channel implementations [20]. An early example was seen in 1976 in the Multics machine where two processes communicated by manipulating intervals between page faults [21]. More modern day implementations are found as part of tools like Squeeza [22].

1.1. Significance

Covert channels have gained notoriety because of their prevalence and application in malware, such as those used in password and information theft. Specifically, more and more trojans and rootkits found in the wild instrument backdoor access to the infected host using covert channels rather than connections to specific ports. The covert channels employed are typically implemented over various legitimate protocols such as ICMP, UDP, TCP, as well as application layer protocols such as http, DNS and email. Because these protocols are required by legitimate users, they are usually not filtered aggressively by network security elements such as firewalls and network intrusion detection systems. Thus, covert channels provide a mechanism for malware to instrument backdoor access that bypasses security measures.

As a specific example, ICMP backdoors have been readily available since 1997 when daemon9 released Loki in a Phrack article [23, 24]. The idea behind these is to have a non-promiscuous sniffer running in the background waiting for either a specially crafted ICMP packet or a specially crafted sequence of packets to cross the network interface. Once the correct packet pattern is seen, a backdoor shell is opened to the attacker. This approach arose because most firewalls still allowed ICMP packets at the time when these tools were first developed [25]. These backdoors were not invincible, however since there is a small window of opportunity where they can be detected during the established connection [25]. Loki became a popular tool of choice, and gave way to other tools based on its implementation including 007shell [25, 26] as well as Cd00r [27], which in turn influenced Sadoor [28, 29] and Helldoor [30]. ICMP backdoor implementations evolved to kernel modules and an implementation by Bioforge was soon seen in a 2003 Phrack article [31] showing how to collect passwords from a compromised server by sending a carefully crafted ICMP packet. Other more sophisticated variations have been seen in the wild which are not as readily available to the public [25]. One such improvement is the release of information (or shell) over an ICMP covert channel or other such storage channel [25]. This helps to avoid detection by open connections in the earlier versions.

Besides ICMP backdoors, many other descriptions of covert channels can be readily found, both in reports from the hacker community [31, 32] and in academic research papers [33, 34, 35].

2. Background

IP packet headers are often used to implement real-world covert channels because there are several IP header fields that are not mandatory and can be easily manipulated. Fields that are most often usurped for covert bandwidth include the IP identification (IP ID), Type of Service (TOS), time-to-live (TTL) and the Options fields [36]. Before describing how some of these fields are used to implement covert channels, we review their legitimate usage.

At the network layer, IP datagrams can be split up into fragments that are reassembled by IP at the destination [36]. This allows IP to send out large PDUs, even if the size exceeds the MTU of a downstream router, since the routers are able to break the packets into smaller fragments as needed. When a large IP packet is broken into multiple IP fragments, the first IP fragment will have an *offset* of zero, and its *flags* will be set to `MF` ("more fragments"). Then, each successive fragment p (where p ranges from $1, 2, 3, 4, \ldots, n$) will have offset $1500p/8$ (assuming that the maximum size for each packet is 1500 bytes and n is the total number of packets). The IP ID of all fragments, however, remains the same as the IP ID of the original large IP packet. When the last fragment is sent, its flags are set to 0x00 to indicate that this is the last fragment that should be expected. It is up to a downstream IP stack to reassemble the fragments into the original datagram. There are several fields in the IP header that help us in out covert channel design, including:

- The Type of Service (TOS) field in the IP header consists of eight bits, and allows the specification of different types of IP datagrams.

- The IP identification (IP ID) field in the IP header is a sixteen bit field that helps with fragmentation. Each time an machine sends out an IP datagram, it sets the datagram's IP ID value. The IP ID value changes predictably over time; most implementations of IP increment it by some fixed value after each IP packet transmission. If an IP datagram is fragmented by a router, each fragment retains the same IP ID as the large IP packet from which it is derived. This allows a downstream IP protocol stack to reassemble the fragments.

2.1. Prior Approaches over IP

An example of a storage channel over IP is Joanna Rutkowska's covert channel using the IP ID field [32]. The channel uses eight bits of the sixteen bit IP ID field to encode covert messages. For example, suppose we want to

send the covert message “JAMIE ” which when expressed in ASCII (hex and binary) is:

```
  J         A         M         I         E
 0x4a      0x41      0x4d      0x49      0x45
 0100 1010 0100 0001 0100 1101 0100 1001 0100 0101
```

then this message would be sent using Rutkowska’s toolkit over a sequence of six IP packets as follows:

```
[tcp] 4510 0064 [4A00] 4000 4006 4370 ...]
[tcp] 4510 0034 [4100] 4000 4006 42a0 ...]
[tcp] 4510 0034 [4D00] 4000 4006 43a0 ...]
[tcp] 4510 0064 [4900] 4000 4006 4270 ...]
[tcp] 4510 0034 [4500] 4000 4006 43a0 ...]
```

By manipulating the sequence of IP IDs that appear in the packets (4A00, 4100, 4D00, 4900, 4500), Rutkowska’s toolkit uses the high order eight bits to encode successive ASCII characters of the covert message.

Murdoch and Lewis [37] describe the pitfalls of storage channels implementations and the factors leading to their discovery. They note that while one may encode covert messages using unused bits in fields of the IP packet header, these fields often take predictable values. For example, while it is possible to encode a message using the options field of the IP header, however since the options field is not used very often, its usage could raise a red flag in network intrusion detection systems. IP IDs are often predictable, too, depending on the system being used, since most systems increment IP IDs for each datagram using a set cycling value. TOS is also somewhat predictable, since it is not used as much either. Therefore, implementing a covert storage channel alone may not be the safest thing to do if one needs to ensure that the channel remains undetected.

3. Our Approach

Our approach is based on the simple observation that TCP provides a reliable in-order transport layer data stream over the unreliable network layer IP datagram service. Moreover, since TCP is designed to be robust to out of order network layer packet delivery, we can artificially permute IP packets before they leave the source, and read the permutation at the destination. The choice of permutation is used to encode a covert value, and the sequence of chosen permutations, in turn, encodes the covert message. Applying such permutations does not adversely affect TCP’s ability to reconstruct the higher layer transport layer data stream.

A coarse upper bound on the capacity of such a covert channel can be computed by noting that if the covert channel queues n IP packets and then permutes them before sending them, it can encode $\log_2(n!)$ bits of information for every n packets sent in its choice of permutations. It is an easy exercise to prove that (for n sufficiently large),

$$\left(\frac{n}{3}\right)^n \leqslant n! \leqslant \left(\frac{n}{2}\right)^n,$$

and so the covert channel conveys $O(\frac{n \log_2 n/2}{n}) = O(\log_2 n)$ covert bits per legitimate packet.

In what follows, we describe three concrete systems we developed in the process of exploring the systems-level issues involved in instrumenting such cover channels.

3.1. Scheme I

Outgoing IP packets are queued in the kernel, based on their destination IP. Every time the queue Q_A of IP packets to destination A contains n IP packets, we determine the next $k = \lfloor \log_2(n!) \rfloor$ bits of the covert message intended for the agent at A, and will use this k bit number

$$C = C_0 \cdot C_1 \cdots C_i \cdots C_{k-2} \cdot C_{k-1}$$

to determine how to permute the n packets before sending them, as follows. First, note that the lower bound inequality in expression (1) and our choice of k together imply that $k \leqslant \log_2(n!)$, or equivalently, the set of all k bit numbers is smaller in cardinality than the set of permutations of n packets. Myrvold and Ruskey have demonstrated [38] an $O(n)$ algorithm for evaluating a permutation ranking function

$$r : S_n \to \{0, 1 \ldots, n! - 1\}$$

where r is a (very particular) bijective map between the symmetric group on n elements (as a set) and the set of integers between 0 and $n! - 1$ (inclusive). In the same paper, they showed that the corresponding unranking function r^{-1} can also be computed in $O(n)$ time. Thus, considering C to be a k bit number, we can use Myrvold and Ruskey’s unranking function to quickly obtain a permutation $\pi = r^{-1}(C)$ of the n packets queued in Q_A. Upon receiving n packets, the destination considers the sequence of IP IDs of the packets to determine the permutation π. The recipient then computes $r(\pi)$ to obtain C.

Analysis. Clearly, this covert channel conveys $\Omega(\frac{n \log_2 n}{n}) = \Omega(\log_2 n)$ covert bits per legitimate packet, and hence is optimal with respect to throughput achieved. On the down side, the covert channel scheme introduces significant additional latency because each packet has to wait until all n packets have been obtained before they are sent out in permuted order. The Myrvold and Ruskey unranking function r^{-1} is, in practice, fairly complex way to encode an integer as a permutation. Most importantly, because the ranking functions r and r^{-1} are bijections of sets, they are extremely efficient and contribute severe sensitivity to errors, in the following sense: Any error in the received permutation (caused by a reordering or loss of IP packets in transit) is effectively undetectable since the corrupted permutation is guaranteed to lie in the domain of r, and therefore, to be syntactically correct. Thus, while this scheme is optimal with respect to throughput, its adoption

requires the design of an error *correction* mechanism (over the covert channel) to recover from permutation corruption. This in turn, mandates that a bi-directional covert channel to be established, since errors must result in solicitations for retransmission. The technical difficulties in doing this led us to consider less efficient (i.e. lower throughput) permutational covert channels which would be less sensitive to permutational corruption due to packet loss and re-ordering. In the next section we describe the design and implementation of one such scheme.

3.2. Scheme II

Outgoing IP packets are queued in the kernel, based on their destination IP. Every time the queue Q_A of IP packets to destination A contains 16 IP packets, we determine the next 16-bits of the covert message

$$C = C_0 \cdot C_1 \cdots C_i \cdots C_{14} \cdot C_{15},$$

viewing C as a number in $\{0, 1 \ldots, 65, 535\}$. The number C is converted into a permutation of 16 packets, using the map

$$\pi \quad : \quad \{0, 1 \ldots, 65, 535\} \rightarrow S_{16},$$

which is defined as follows. Let:

$$\begin{aligned} S_0(C) &= (i \mid 0 \leqslant i \leqslant 15, C_i = 0) \\ S_1(C) &= (i \mid 0 \leqslant i \leqslant 15, C_i = 1). \end{aligned}$$

Then $\pi(C)$ is the permutation obtained as follows:

(i). If $\min S_0(C) < \max S_1(C)$: then we take $\pi(C)$ to be the elements of $S_1(C)$ in increasing order followed by the elements of $S_0(C)$ in increasing order.

(ii). If $\min S_0(C) > \max S_1(C)$: then since it follows that $\max S_0(C) > \max S_1(C) \geqslant \min S_1(C)$ so we take $\pi(C)$ to be the elements of $S_1(C)$ in decreasing order followed by the elements of $S_0(C)$ in decreasing order.

Clearly $\pi(C)$ is a permutation of the numbers $\{0, 1, \ldots, 15\}$, and the two-case definition ensures that the recipient can unambiguously identify from $\pi(C)$ which numbers are in $S_0(C)$ and which are in $S_1(C)$, and thus reconstruct C.

In practice, before a text message can be sent, it must be encoded in a particular scheme. We allowed for upper and lower case encodings using both ASCII and EBCDIC. For example, Table 1 shows the message "JAMIE" encoded in ASCII upper case. The first 16-bit block of the covert message "JAMIE" results in considering C to consist of the letters "JA" (which encoded as uppercase ASCII are `0x4a 0x41`). Translated this covert message into binary (network order) we get `0100 0001 0100 1010`, a 16-bit number in which bits 1, 3, 6, 8 and 14 are one (1) and all other bits are zero (0). Thus

Symbol	Value	7	6	5	4	3	2	1	0
J	0x4a	0	1	0	0	1	0	1	0
A	0x41	0	1	0	0	0	0	0	1
M	0x4d	0	1	0	0	1	1	0	1
I	0x49	0	1	0	0	1	0	0	1
E	0x45	0	1	0	0	0	1	0	1

Table 1. ASCII Uppercase "JAMIE".

$$\begin{aligned} S_0(C) &= \{0, 2, 4, 5, 7, 9, 10, 11, 12, 13, 15\} \\ S_1(C) &= \{1, 3, 6, 8, 14\}. \end{aligned}$$

Since $\min S_0(C) < \max S_1(C)$, the permutation is constructed according to rule (i), which mandates:

$$\pi \quad = \quad (1, 3, 6, 8, 14, 0, 2, 4, 5, 7, 9, 10, 11, 12, 13, 15).$$

Thus the 16 queued packets

$$P_0 P_1 \cdots P_i \cdots P_{14} P_{15}$$

are written out according to the order dictated by π:

$$P_1 P_3 P_6 P_8 P_{14} P_0 P_2 P_4 P_5 P_7 P_9 P_{10} P_{11} P_{12} P_{13} P_{15}.$$

We also need to let the recipient of the message to know which of the encoding schemes has been used and which packets are of interest. We used the TOS field of the IP datagrams to do this, by setting the TOS to be:

$$\begin{aligned} AU : `A' + `U' = 65 + 85 \quad &== \quad 150 \\ AL : `A' + `L' = 97 + 117 \quad &== \quad 214 \\ EU : `E' + `U' = (197 + 228)\%256 \quad &== \quad 169 \\ EL : `E' + `L' = (133 + 164)\%256 \quad &== \quad 41 \end{aligned}$$

The values for the initials of the encoding scheme are added together as an 8 bit value. The recipient deduces the encoding by reading the TOS field value.

Analysis. Each bit in the covert message is represented by the placement of *one* IP packet. The covert channel thus exhibits a throughput of one covert bit per legitimate packet. While this is significantly less efficient compared to the theoretically optimal Scheme I, it provides certain advantages:

- The π function defined in Scheme II is not a bijection, but rather maps $2^{16} = 65, 536$ messages injectively into the $16! = 20, 922, 789, 888, 000$ permutations. Thus, if errors introduced by packet reordering result in a random permutation reaching the destination, this permutation has only a

$$\frac{65, 536}{20, 922, 789, 888, 000} \approx 3 \cdot 10^{-9}$$

probability of being syntactically correct[1] (i.e. in the range of π). Thus, compared to Scheme I, the less efficient Scheme II enjoys error *detection* capabilities and a more straightforward encoding process.

Evaluation. Initial trials were conducted using a program called Scapy [39] to construct packets using our covert channel scheme. Scapy is written in Python and enables the construction of any type of packet, as well as sending and receiving of packets. Packets were constructed by hand and collected at the recipient's machine using the Wireshark [40] packet sniffer. It was not possible to gauge the efficiency of the Scapy based trials, since the packets were constructed by hand and sent one at a time.

In order to fully evaluate the efficiency of this scheme we attempted to implement it on a Linux machine by taking advantage of Netfilter, which allows us to catch and process packets en route through the kernel [41]. The code was designed as a transparent proxy over the `ip_queue` module. In practice, however, we found that queueing of 16 IP packets prior to their transmission would cause TCP layer to block the application layer because too many unacknowledged packets had been sent, exhausting the transmission window. TCP would retransmit the same segment repeatedly, and since RFC 2988 follows Karn's algorithm for taking RTT samples (which mandates that RTT must not be estimated using segments that were retransmitted), TCP is deprived of an estimate of round trip time (RTT). The TCP stack interpreted the delay as extreme network congestion, and following directive 5.5 of the RFC specification, it "backs off the retransmission timer (RTO)" exponentially (doubling it each time), following Van Jacobson's algorithm for adjusting RTO [42]. The net effect of this is that the TCP layer retransmits packets with exponentially growing interpacket intervals, until 16 packets have been delivered to IP, at which point they burst out, and the process repeats in this manner. The throughput (of the legitimate channel) is reduced tremendously because of the interactions between TCP and the transparent proxy implementing the permutation scheme. Indeed, this difficulty was foreseen by Paxson and Allman in Section 6 "Security Considerations", where they write:

> This document requires a TCP to wait for a given interval before retransmitting an unacknowledged segment. An attacker could cause a TCP sender to compute a large value of RTO by adding delay to a timed packet's latency, or that of its acknowledgment. However, the ability to add delay to a packet's latency often coincides with the ability to cause the packet to be lost, so it is difficult to see what an attacker might gain.. [43].

[1] An 8-bit blocksize yields a much higher probability of misparsing, since $2^8/8! = 6.3 \cdot 10^{-3}$; this was the reason we chose a 16-bit blocksize for our channel design.

The next permutational covert channel scheme we describe attempts to circumvent the above problem by reducing the maximum number of IP packets queued for each covert bit transmitted.

3.3. Scheme III

Scheme III is a further simplification of Scheme II, designed to circumvent the TCP layer stalling we encountered in that system. The new scheme operates as follows. Suppose we want to transmit a covert message C consisting of a sequence of n bits

$$C = C_0 \cdot C_1 \cdots C_i \cdots C_{n-1} \cdot C_n,$$

to machine A. We begin by determining bit C_i of the covert message to A. For each covert bit, we allow two IP packets to destination A to go through. If $C_i = 0$, we ensure that the two packets are in ascending order by IP ID; if $C_i = 1$, we ensure that the two packets are in descending order by IP ID. Thus, we transmit C by taking the next $2n$ IP packets submitted by TCP for A:

$$P_0, P_1, \ldots, P_{2i}, P_{2i+1}, \ldots, P_{2n-2}, P_{2n-1}.$$

The decoding of the covert message can be deduced by considering the sequence of IP IDs and computing the first derivative of successive pairs: a negative first derivative signifies a 1 while a positive first derivative signifies a 0.

For example, let us see how letter 'J' (hex `0x4a`, binary `0100 1010`) can be encoded in a suitable permutation of 16 IP packets. Sixteen queued packets

$$P_0 P_1 \cdots P_i \cdots P_{14} P_{15}$$

are sent out (2 at a time) according to the order:

$$P_0 P_1 P_3 P_2 P_4 P_5 P_6 P_7 P_9 P_8 P_{10} P_{11} P_{13} P_{12} P_{14} P_{15}.$$

The question of whether packets $2i$ and $2i + 1$ arrive in ascending or descending IP ID order determines the value of covert bit i. Figure 1 shows the character 'J' being transmitted over a sequence of 16 IP packets.

Analysis. Each bit in the covert message is represented by the placement of *two* IP packets. The covert channel thus exhibits a throughput of $1/2$ covert bit per legitimate packet. While this is significantly less efficient compared to the theoretically optimal Scheme I, and only 50% the throughput of Scheme II, it provides certain advantages:

- A concrete implementation of Scheme III only requires delaying at most one IP (when the covert bit that needs to be transmitted is a 1, and does not require delaying any IP packets when the covert bit that needs

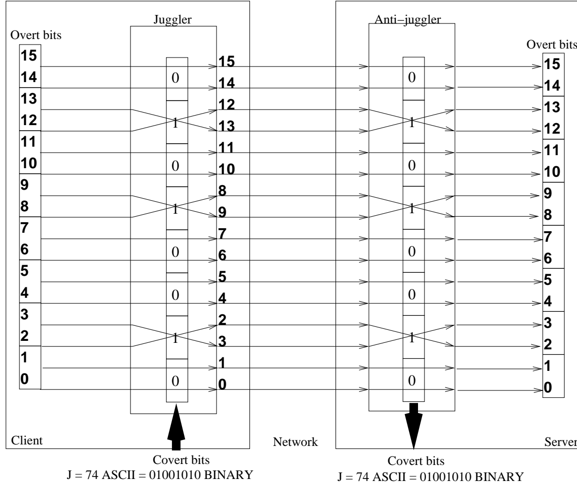

Figure 1. Transmitting 'J' over a sequence of 16 IP packets.

to be transmitted is a 0. Thus, amortizing, if the covert message consists of roughly an equal number of 0s and 1s, the expected queue depth is 0.5 packets.

- In terms of error resilience, we note that if packet $2i$ and $2i+1$ arrive in reversed order (because of network effects), this manifests a one bit error in bit i of the covert message at the receiver.

$$\begin{aligned} \text{IDs}: 23\ \text{sent(covert:0)} &\rightarrow 32\ \text{rcvd(covert:1)} \\ \text{IDs}: 32\ \text{sent(covert:1)} &\rightarrow 23\ \text{rcvd(covert:0)} \end{aligned}$$

However, if packet $2i$ and $2i-1$ arrive in reversed order (because of network effects), this also produces, on average a one bit error of the covert message received (bit i and/or $i-1$):

$$\begin{aligned} \text{IDs}: 2345\ \text{sent(covert:00)} &\rightarrow 2435\ \text{rcvd(covert:00)} \\ \text{IDs}: 2354\ \text{sent(covert:01)} &\rightarrow 2534\ \text{rcvd(covert:00)} \\ \text{IDs}: 3245\ \text{sent(covert:10)} &\rightarrow 3425\ \text{rcvd(covert:00)} \\ \text{IDs}: 3254\ \text{sent(covert:11)} &\rightarrow 3524\ \text{rcvd(covert:00)} \end{aligned}$$

Thus packet re-ordering produces *localized* corruption in the covert bitstream.

Single packet loss has reasonable side effects as well. If an even number of packets are lost, this causes the corresponding bits of the covert channel to be lost. In contrast, if an odd number of packets are lost, this causes a framing error which exhibits itself as a burst of 0-valued covert bits. For example, if the first packet P_1 in the permutation representation of 'J' is lost, then the residual stream:

$$P_3P_6P_8P_{14}P_0P_2P_4P_5P_7P_9P_{10}P_{11}P_{12}P_{13}P_{15}...$$

experiences a framing error which causes the covert message to become `0000 000....` Thus packet loss produces either *localized* corruption in the covert bitstream, or large scale corruption that is recognizable.

4. PERMEATE

PERMEATE is an open source project which implements the permutational covert channel described in Scheme III. It is available from Sourceforge at `http://permeate.sourceforge.net`.

4.1. Implementation

The architecture of PERMEATE is depicted in Figure 2 below. Covert bits are transferred from one host (referred to as the "client") to another host (referred to as the "server") by encoding the covert bits in permutations of IP packets en route from client to server. The interposition is achieved at the client by causing the Linux Netfilter to forward all outbound traffic destined to the server to the `ip_queue` module. PERMEATE's juggler application acts a transparent proxy queueing pairs of outbound IP packets from the `ip_queue` module before sending them back down in either the ascending or descending order (with respect to IP ID). At the server, interposition is instrumented by causing the Linux Netfilter to forward all inbound traffic destined from the client to the `ip_queue` module. PERMEATE's anti-juggler application acts a transparent proxy queueing pairs of inbound IP packets from the `ip_queue` module and checking to see if they are in ascending or descending order (with respect to IP ID) in order to reconstruct covert bits, after which it returns the packets back to `ip_queue` in ascending order (with respect to IP ID)[2].

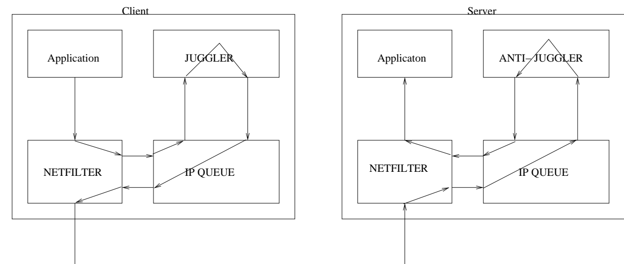

Figure 2. The architecture of PERMEATE.

4.2. Covert Channel Throughput

Analyzing the design of Scheme III, we expect the *bit* rate of the covert stream to be roughly half the cumula-

[2]While this is not technically needed for TCP to operate correctly, we implemented it for reasons of "personal aesthetics".

tive IP *packet* rate corresponding to all transport layer traffic from the client to the server host. With IP packets of around 576 bytes, this would mean that the bit rate of the covert channel would be roughly $1/1152$ fraction of the bit rate of the overt channel. In actuality, TCP overhead caused the covert channel to achieve only approximately $1/3039$ of the overt channel throughput. We affirmed this analysis in several experiments–for example over an 7.45 Mb/s overt TCP connection, we were able to superimpose a 2.51 Kb/s covert channel having a bit error rate of 7.51%. We note that the overt channel's throughput without the overhead of packet processing in the transparent proxy was over 10.75 Mb/s–thus proxies (with their kernel-userspace transition) caused throughput reduction of $\approx 31\%$.

5. User Guide

In order for the user to be able use Permeate, he/she needs to have root access on both the client and the server. To run permeate on the client, type:

```
$ ./juggler.sh <file-to-send-covertly> <server-ip>
```

The script checks that the user is root and that `ip_queue` is loaded, then instruments the Netfilter rule

```
/sbin/iptables -A OUTPUT -p tcp -d <server-ip>
               -j QUEUE
```

and launches the juggler proxy (the permeate binary, with the "c" flag specifying for client operation)

```
$ ./permeate c <file-to-send-covertly> <server-ip>
```

To run permeate on the server, type:

```
$ ./antijuggler.sh <save-covert-file-as> <client-ip>
```

The script checks that the user is root and that `ip_queue` is loaded, then instruments the Netfilter rule

```
/sbin/iptables -A INPUT -p tcp -s <client-ip>
               -j QUEUE
```

and launches the anti-juggler proxy (the permeate binary, with the "s" flag specifying for server operation)

```
$ ./permeate s <save-covert-file-as> <client-ip>
```

The actual file transfer takes place from the client to the server as normal traffic flows between the two hosts (e.g SSH, HTTP, etc.)

6. Conclusion and Future Work

We have demonstrated our journey from the design of an idealized permutational covert channel to the implementation of a real permutational covert channel. The final implementation Scheme III channel operates at roughly $1/3000$ of the bit rate of the overt channel. It is also well-behaved in the face of in-transit packet loss and packet re-ordering. Our future efforts will involve extending PERMEATE to support a bidirectional permutational covert channel over bidirectional TCP streams, and then implementing a reliable data stream protocol using the covert channels. We also intend to migrate PERMEATE to a loadable kernel module (LKM) based architecture thereby circumventing latencies introduced by crossing the kernel-userspace boundary via `ip_queue`.

7. Acknowledgments

The authors would like to thank Joel Sandin for many spirited discussions during the early stages of this project.

References

[1] T. MacDermid, "StegTunnel." [Online]. Available: http://www.synacklabs.net/projects/stegtunnel/

[2] J. Rutkowska, "Nushu." [Online]. Available: http://invisiblethings.org/tools/nushu/nushu.tar.gz

[3] ——, "passivecc_ipid.c." [Online]. Available: http://invisiblethings.org/tools/passivecc_ipid.c

[4] R. Greenstadt and J. Giffin, "DEVCC." [Online]. Available: http://www.mit.edu/~gif/covert-channel/src/

[5] A. Vidstrom, "AckCmd." [Online]. Available: http://ntsecurity.nu/toolbox/ackcmd/

[6] C. H. Rowland, "Covert Channels in the TCP/IP Protocol Suite." [Online]. Available: http://www.firstmonday.dk/issues/issue2_5/rowland/

[7] S. Nomad, "NCovert." [Online]. Available: http://ncovert.sourceforge.net/

[8] T. M. Gil, "ICMPTX." [Online]. Available: http://thomer.com/icmptx/

[9] D. Stodle, "Ping tunnel." [Online]. Available: http://www.cs.uit.no/~daniels/PingTunnel/

[10] S. Chavlyuk, "Simple ICMP tunnel." [Online]. Available: http://sourceforge.net/projects/itun

[11] I. Zelenchuk, "Skeeve." [Online]. Available: http://www.gray-world.net/poc_skeeve.shtml

[12] P. LeBoutillier, "HTTunnel." [Online]. Available: http://sourceforge.net/projects/httunnel/

[13] S. Castro, "Cctt." [Online]. Available: http://gray-world.net/pr_cctt.shtml

[14] A. Dyatlov, "Firepass." [Online]. Available: http://www.gray-world.net/pr_firepass.shtml

[15] D. Uid, "hcovert." [Online]. Available: http://sourceforge.net/projects/hcovert/

[16] T. M. Gil, "NSTX." [Online]. Available: http://thomer.com/howtos/nstx.html

[17] T. Pietraszek, "DNScat." [Online]. Available: http://tadek.pietraszek.org/projects/DNScat/

[18] W. Zheng, "MsnShell." [Online]. Available: http://gray-world.net/pr_msnshell.shtml

[19] "U.S. Department of Defense Trusted computer system evaluation," 1985. [Online]. Available: http://csrc.ncsl.nist.gov/publications/secpubs/rainbow/std001.txt

[20] H. Meer and M. Slaviero, "It's all about the timing..." [Online]. Available: http://www.sensepost.com/research/squeeza/dc-15-meer_and_slaviero-WP.pdf

[21] T. V. Vleck, "Timing Channels." [Online]. Available: http://www.multicians.org/timing-chn.html

[22] H. Meer and M. Slaviero, "Squeeza." [Online]. Available: http://www.sensepost.com/research/squeeza/

[23] daemon9, "Project Loki," in *Phrack Volume 49*. [Online]. Available: http://www.phrack.org/issues.html?issue=49\&id=6\&mode=txt

[24] ——, "LOKI 2(the implementation)," in *Phrack Volume 51*. [Online]. Available: http://www.phrack.org/issues.html?issue=51\&id=6#article

[25] E. Skoudis and L. Zeltser, "Malware: Fighting malicious code." O'Reilly, 2004.

[26] FuSyS, "007shell." [Online]. Available: http://packetstormsecurity.org/groups/s0ftpj/007shell.tgz

[27] FX, "Cd00r." [Online]. Available: http://www.phenoelit-us.org/stuff/cd00r.c

[28] R. Bejtlich, "Chained covert channels," in *The Tao of Network Security Monitoring*. Addison-Wesley, 2005.

[29] C. M. Nyberg, "Sadoor." [Online]. Available: http://packetstormsecurity.org/UNIX/penetration/rootkits/index7.html

[30] T. Redaelli, "Helldoor." [Online]. Available: http://utenti.gufi.org/~drizzt/codes/helldoor/

[31] Bioforge, "Hacking the linux kernel network stack," in *Phrack Magazine, vol 61*. [Online]. Available: http://www.phrack.org/archives/61/p610x0d_Hacking_the_Linux_Kernel_Network_Stack.txt

[32] J. Rutkowska, "The implementation of passive covert channels in the linux kernel." [Online]. Available: http://www.invisiblethings.org/papers/passivecovertchannelslinux.pdf

[33] I. S. Moskowitz and M. H. Kang, "Covert channels - here to stay?" in *Proceedings of the 9th Annual Conference on Computer Assurance*. National Institute of Standards and Technology, 1994, pp. 235–244. [Online]. Available: http://citeseer.ist.psu.edu/moskowitz94covert.html

[34] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn, "Information hiding — A survey," *Proceedings of the IEEE*, vol. 87, no. 7, pp. 1062–1078, 1999. [Online]. Available: citeseer.ist.psu.edu/petitcolas99information.html

[35] I. Moskowitz, R. Newman, D. Crepeau, and A. Miller, "Covert channels and anonymizing networks," in *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*, 2003. [Online]. Available: citeseer.ist.psu.edu/moskowitz03covert.html

[36] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. Addison-Wesley, 2006.

[37] S. J. Murdoch and S. Lewis, "Embedding Covert Channels into TCP/IP," in *Proceedings of the Information Hiding Workshop*, 2005.

[38] W. Myrvold and F. Ruskey, "Ranking and unranking permutations in linear time," *Information Processing Letters*, vol. 79, no. 6, pp. 281–284, 2001. [Online]. Available: citeseer.ist.psu.edu/myrvold00ranking.html

[39] P. Biondi, "Scapy." [Online]. Available: http://www.secdev.org/projects/scapy/

[40] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce, *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[41] J. Toledo, "EtherApe." [Online]. Available: http://etherape.sourceforge.net/

[42] V. Jacobson, "Congestion avoidance and control," *Computer Communication Review*, vol. 18, pp. 314–329, Aug. 1988.

[43] V. Paxson and M. Allman, "RFC2988: Computing TCP's retransmission timer," United States, 2000.