A Protocol for Building Secure and Reliable Covert Channel

Baishakhi Ray
University Of Colorado, Boulder
Department Of Computer Science
ECOT 717, Boulder, CO 80309, USA
baishakhi.ray@cs.colorado.edu

Shivakant Mishra
University Of Colorado, Boulder
Department Of Computer Science
ECOT 717, Boulder, CO 80309, USA
mishras@cs.colorado.edu

Abstract

This paper presents a secure and lightweight protocol for reliable data transfer through moderate bandwidth covert channels. Though data transfer through covert channels is not unprecedented, existing covert channels have been restricted to covert transmission of only small amounts of data. This paper demonstrates that it is possible to transmit large amounts of data covertly with sophisticated support such as security and reliability. The proposed protocol exploits ICMP Echo Request as covert medium, and uses OS finger-printing techniques to simulate real TCP/IP stack behavior for further security enhancements.

1 Introduction

A covert channel is a logical link between two compromised systems through which two end applications can secretly exchange information without being detected. A covert channel remains undetectable to an intermediary, despite the fact that the intermediary may have privileges such as an ability to intercept and observe all communication traffic along this channel. A covert channel is designed to be hidden within the normal communication traffic of a legitimate logical channel, such as TCP or UDP. Secret information is embedded in the legitimate channel packets in such a way that only the end applications can detect and retrieve this information. Anyone else watching the network traffic is unable to detect the presence of such information in the legitimate channel packets. Because a covert channel hides within a legitimate logical channel, it is a very simple yet effective mechanism for exchanging information between two end applications without alerting any firewalls or intrusion detectors on the network.

While a large number of covert channels have been designed and implemented over the last fifty years, they have largely been restricted to covertly exchanging only small amounts data and that too only via infrequent message exchanges. The key reason for these restrictions is that it is very difficult to hide large amount of data in legitimate logical channels without raising suspicion in the intermediate observers such as intrusion detectors. Furthermore, exchanging frequently even small amounts of data raises the possibility of suspicion. Because of these restrictions, it has not been possible so far to maintain extended sessions of intelligent information exchange between two cooperating end-applications via a covert channel.

In this paper, we describe the design, implementation and evaluation of a protocol to establish a covert channel that not only maintains a high degree of stealthiness, but also allows end applications to maintain prolonged communication sessions and provides support for reliability as well as data confidentiality. An important feature of our proposed protocol is that it can be embedded in any legitimate logical channel that is based on IP. In this paper, we describe our protocol using ICMP. This protocol satisfies the following important properties:

- **Stealthiness:** An intermediary cannot detect the presence of the covert channel despite observing all communication traffic. This property is achieved by ensuring that the statistical property of the channel with covert data embedded is same as the statistical property of the underlying legitimate channel without any covert data. Stealthiness is important because if an intermediary can detect the existence of a covert channel, he/she can simply destroy that channel as well as the entire covert communication. In particular, statistical similarity between a covert channel and a legitimate channel prevents a network intrusion detector from detecting the existence of a covert channel.

- **Lightweight:** Difference between the resources (CPU, memory and bandwidth) required to operate the proposed covert channel and those required to operate the legitimate channel is minimal. This property is important to ensure that a host-based intrusion detector or other users who are using the end hosts do not become

978-0-7695-3390-2/08 $25.00 © 2008 IEEE
DOI 10.1109/PST.2008.26

suspicious of a covert activity by observing deviations in the resources being consumed.

- **Confidentiality and integrity**: Only the end applications can decipher the secret information being exchanged through the covert channel. Furthermore, end applications can detect any tampering with the secret information being exchanged.

- **Reliability and Ordering:** End applications are able to exchange secret information via the proposed covert channel, despite intermediate data loss. Furthermore, the protocol ensures with high probability that the secret information is received in the same order in which it was sent, even though the underlying legitimate channel may reorder different packets.

We have implemented a prototype of this protocol using ICMP in Linux kernel, and performed a number of experiments. Results from these experiments show that there is no statistical difference in the communication data pattern resulting from our protocol embedded in ICMP and standard ICMP. Furthermore, the results show our protocol does not result in any significant changes in resources consumed. To demonstrate that our protocol can be used for intelligent information exchange, we have implemented an application wherein an end-application can open a shell on a remote machine using our covert channel.

The rest of this paper is organized as follows. Section 2 summarizes the current and past work done in building covert channels and identifies the differences between past work and the protocol proposed in this paper. Section 3 describes the design of our protocol. Section 4 describes a prototype implementation of our protocol over ICMP within the Linux operating systems and provides an evaluation of our protocol along with some performance measured from our prototype implementation. This section also discusses the application we have built using our covert channel. Finally, Section 5 concludes the paper.

2 Related Work

Several methods have been proposed to build covert channels between two compromised hosts, exploiting redundancy in TCP/IP protocol. [7] shows how to use IP identification field and TCP initial sequence number field, by simply replacing them with clandestine ASCII characters. The method can further be improved by XOR-ing the data with some random numbers. Thus one can send up to 16 bits exploiting IP identification field, while up to 32 bits of covert data can be sent through TCP sequence number. Another possible data hiding approach is encoding the clandestine data in TCP acknowledgment sequence number field. This method relies on IP address spoofing. Both the sender and destination addresses are forged, the covert data is sent to a remote site (destination address), and the remote site then bounces back the packet to the intended target, i.e. the source address of the packet. Another simple but elegant deployment of covert channel is Loki [3]. It uses data part of ICMP Echo request and echo-reply packet for covert communications. This method increases the bandwidth considerably at the expense of relatively higher chances of getting detected.

[2] investigates the possibility of data hiding by packet header manipulation and packet sorting. In the first approach, [2] exploits the redundancy in DF Flag (Don't Fragment) flag in IP header, while MF flag is 0. But this method is less efficient, since in each packet, only one bit covert data can be sent, hence only results in nominal bandwidth utilization. A better approach can be hiding data in the 32 bit sequence number field of the authentication header (AH) and encapsulating security payload (ESP) in the IPSec protocol. This provides information on natural ordering of the packet stream, and is therefore utilized for data hiding.

Another potential field for data hiding is TCP timestamp, studied in detail in [5]. Low order bits of TCP time stamp depend on host machine, and are effectively random from outside detection point of view. Hence, data can be stored in these low order bits, without changing the statistical property of the timestamp, effectively. [4] IPv6 Destination option field can be another area of interest. If the option type in TLV field of IPv6 extension header is set to 00, the options are supposed to be skipped. Hence, a destination options extension header with TLV encode the message, the highest-order 2 bits of the option type to 00 and an option type with value not taken yet, can be a source of covert channel.

Though covert channels have been implemented over many different logical channels, they all share one similarity. They do not provide enough bandwidth for any real data transfer. Some implementations support data hiding of up to four bytes or more. However, they violate statistical property of the systems, and hence are easily detectable. For example, in TCP sequence number, one can embed the data of up to four bytes, but sequence numbers are supposed to be monotonoically increasing after a connection has been established. An alternative is to send TCP SYN everytime with new sequence number, but that creates too many TCP-SYN packets between two hosts, causing suspicion. A system administrator can prevent the two hosts to communicate based on such suspicion. Hence covert communication over a longer period of time is not be possible with TCP SYN packets.

We propose a protocol to build covert channel over any IP-based channels. In this paper, we focus on building a covert channel over ICMP Echo-Request packet using this protocol. There are two reasons for choosing ICMP echo

request packets. First, ICMP echo request packets are commonly used to check the status of a network. In fact, a majority of the users use ICMP to ping other machines to know whether it is up or not, or to check the status of a link. As such, firewalls and networks consider ping traffic to be benign and allow it to pass through [3]. Secondly, an intermediary cannot disable ICMP traffic without interrupting normal user services, because ICMP is widely used to communicate abnormal conditions during IP routing, and check network status.

Though [3] has already proposed method to send covert data over ICMP echo request and reply packets, they cannot keep the statistical property of the underlying ICMP channel. Our protocol is unique to the extent that instead of exploiting all the 56 bytes that are available as default for ICMP Echo request data [1], we use only five bytes for data. Ping implementation of Linux uses timestamp in first 8 bytes of data field as OS fingerprint. We embed the covert data in the last five bytes of the OS fingerprint to maintain the same statistical property as the normal ICMP data. In addition, to provide a reliable communication, we have designed an 8-bit protocol header that is embedded inside the ICMP identification field. Hence effectively we send 6 bytes of covert data per packets.

3 Protocol Design

We propose a protocol for constructing a secure and reliable covert channel. An important feature of this protocol is that it can be embedded in any IP-based channel. The protocol provides support for secure (confidentiality and integrity) and reliable bi-directional data exchange. The key challenge in building such a protocol is to ensure that it is computationally lightweight and consumes low bandwidth of internet.

3.1 Reliability

To establish reliability, we choose the simple stop-and-wait automatic repeat request (ARQ) mechanism. There are of course more sophisticated and popular reliability mechanisms available, e.g. go-back-n or selective repeat request. But the major problem with implementing them is that they tend to use much larger bandwidth than we can afford to avoid detectability. Since low bandwidth usage is of paramount importance for our design, these sophisticated mechanisms with large bandwidth requirements do not suit well.

For example, all modern operating systems support rate limiting of ICMP echo request/reply traffic. If the "limiting rate" is lower than the transmit window size of go-back-n or selective repeat request ARQ then the window size of the protocol will effectively converge to the "limiting rate". This will result in a lot of unnecessary retransmissions raising suspicions at the intermediary. In addition to low bandwidth, stop-and-wait mechanism has the advantage that the size of the sequence number and acknowledgment sequence number can be restricted to just one or two bits. This minimizes protocol overhead in low-bandwidth covert channels. Though stop-and-wait is not very efficient is terms of bandwidth usage, we intentionally want to keep the bandwidth low to avoid detection.

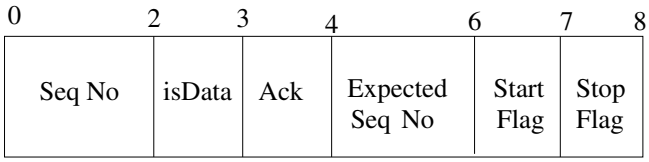

Figure 1. Protocol Header

Our protocol header is shown in Figure 1. The *sequence number* is two bits long and is used for identification purposes. Two bits of sequence number field allows for four sequence numbers. We believe that four sequence numbers are sufficient for protecting against sequence number wrapping around and being reused before an earlier packet with the same sequence number has been processed. The *IsData* field is a one-bit flag to indicate if the packet contains covert data in the data part of the Packet. The acknowledgment is comprised of three bits: the first bit *Ack* indicates acknowledgment (set to 1); the next two bits are the sequence number of the frame being acknowledged. Hence an acknowledgment value of 100 means frame 0 has been received and the sequence number of the next expected frame is 1. Similarly, a value of 101 in this field means frame 1 has been received and the sequence number of the next expected frame is 2. Finally, the *Start flag* bit is set to indicate start of the covert communication, and the *Stop flag* bit is set to indicate termination of the covert communication.

In accordance with ARQ mechanism, a sender expects an acknowledgment for a frame sent within a fixed timeout period. In case it does not receive this acknowledgment, it retransmits it. The data exchange can be bi-directional, in which case acknowledgment can be piggybacked with data, indicated by setting the *isData* flag to 1. The basic mechanism is explained in the following diagrams:

3.2 Security

Our goal is to provide two types of security support in our protocol: data confidentiality and data integrity. While techniques to provide this support are well known, the key challenge is to provide this support with minimal resources. For example, data confidentiality can be provided by using an iterated cryptosystem such as 3-DES or AES. However,

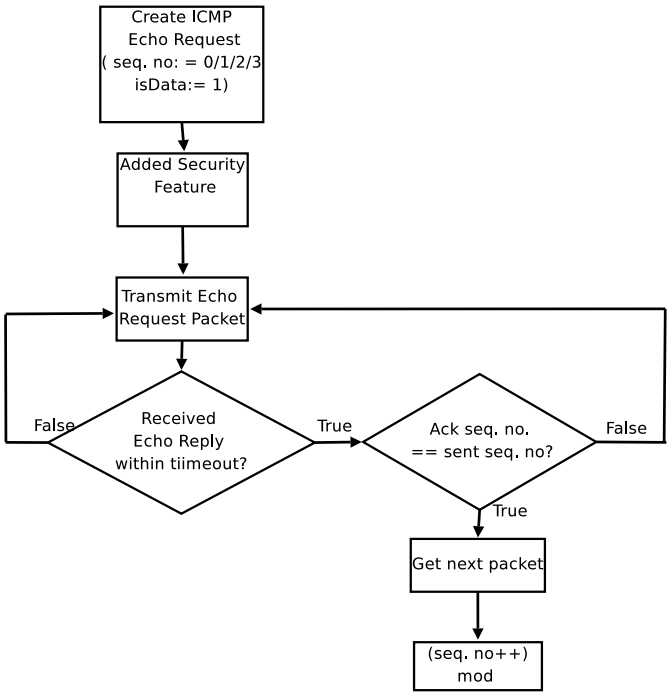

Figure 2. State Machine for Sender

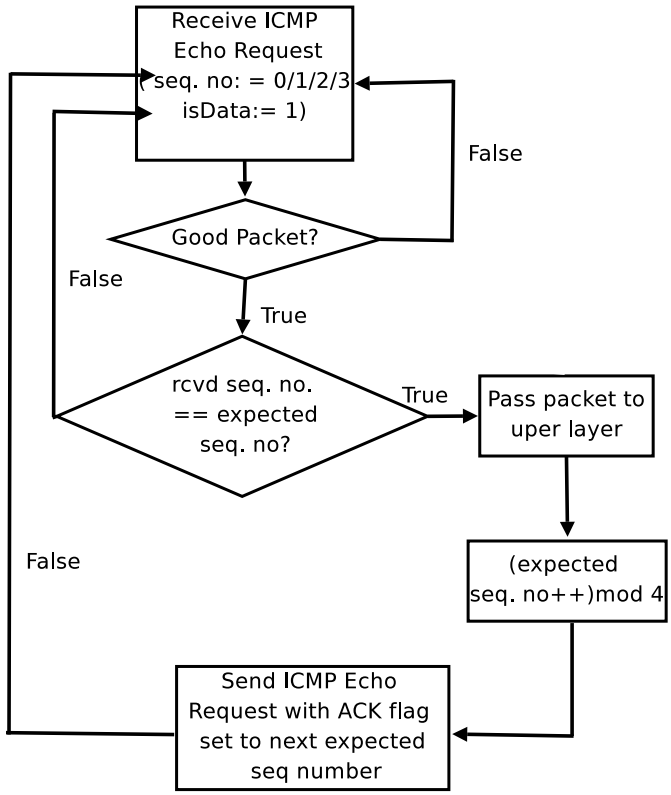

Figure 3. State Machine for Receiver

iterated cryptosystems are hugely complex and computationally intensive so the chances of being exposed increases significantly. As we discussed earlier, in most of the cases a compromised host acts as one end point of the covert channel. It is very important for the covert channel daemon to use minimum resources, as more resource usage will raise suspicion. Hence iterated cryptosystems do not satisfy our requirements. Instead, we have chosen a secure, lightweight encryption-decryption algorithm based on chaotic dynamics.

This algorithm uses one-time pad (OTP) as the basic encryption mechanism. In this algorithm, plain text M is combined (XOR-ed) with an equal length string of random bits K called key or pad. This key is usually generated by a cryptographically strong pseudo-random number generator (CSPRNG). As this key (pad) is used only once and never used anywhere else by any mechanism, this is called one time pad. Encryption mechanism is defined as

$$C = E(M, K) = M \oplus K \tag{1}$$

Decryption mechanism is similarly defined as

$$M = D(C, K) = C \oplus K \tag{2}$$

One-time pad is the only cryptosystem that can is claimed to be fully secure. But the basic requirement is that no portion of the key would be reused for another encryption. This requires random key distribution at both ends of the covert channel, which is non-trivial. If the key is transmitted by another covert channel and the intermediary can somehow decipher it, the whole method will be in vein. Hence, using OTP in its basic form is not possible in most of the general-purpose applications, including our protocol.

We use chaotic dynamics based pseudo random number generator (PRNG) as the source of our key material to eliminate this drawback. We need secure distribution of the random string K, and K must get initialized to a new random value for each encryption. The first requirement can be achieved if we can keep the amount of information to be exchanged to a minimum and that is exchanged once per encryption. For solving the second problem, we need a cryptographically strong pseudo random number generator (CSPRING) that produces sequences of values that have the following properties:

1. The values have minimal internal correlation.
2. The values convey minimum possible information about their origin.
3. The values are absolutely dependent upon unique and sensitive initial condition for reproduction.

Hence, we propose a new cryptographically strong pseudo-random number generator based on a well-known

chaotic map called the logistic map [6]. The logistic map is defined as follows

$x_n = rx_{n-1}(1 - x_{n-1})$, where $0 < x_i < 1$ and $0 \leq r \leq 4$

When $3.57 < r < 4$, the iteration values $(x_0, x_1, , x_n, ...)$ are random. Now let the pseudo random number generator function be defined as (here R_k is the k^{th} bit generated by PRNG)

$$R_k = \begin{cases} 0, & \text{if } x_{nk} \geq 0.5 \\ 1, & \text{if } x_{nk} < 0.5 \end{cases}$$

The values of x_0, r and n need to be secretly pre-shared between the sender and the receiver. As this logistic map has even distribution between (0,1) on both sides of 0.5, it serves as a good random number generator.

Based on this, the random number generator function is defined as $R_k(x_0, r, n)$ that takes x_0 , r and n as input and outputs a bit 0 or 1 for each iteration k. The sender breaks the message in chunks of P_1, P_2, ..., of size i each and computes the cipher text blocks C_1, C_2, ... The pads (secret keys) are defined as B_1, B_2, ... from which each cipher text block is computed.

$$B_1 = R_1..R_i \qquad C_1 = P_1 \oplus B_1$$

$$B_2 = R_{i+1}..R_{2i} \oplus C_1 \qquad C_2 = P_2 \oplus B_2$$

$$B_n = R_{(n-1)i+1}..R_{2ni} \oplus C_{n-1} \qquad C_n = P_n \oplus B_n$$

Since the receiver has knowledge about x_0, r and n, it can easily generate $B_1 = R_1...R_i$. On receiving C_1 the receiver generates the original plain text by simply XOR-ing it with B_1.

$$P_1 = C_1 \oplus B_1$$

The receiver generates the next random key (pad) by

$$B_2 = R_{i+1}..R_{2i} \oplus C_1$$

The received cipher text C_2 is then deciphered as

$$P_2 = C_2 \oplus B_2$$

Similarly,

$$P_n = C_n \oplus B_n$$

This method provides data confidentiality with minimal computational requirements. In addition, a checksum provides support for data integrity. This feature is optional, because the legitimate channel within which covert data is being transmitted may itself provide some support for data integrity. For example, ICMP provides a checksum field that can be used to detect data tampering.

3.3 Covert Channel over ICMP

Based on these reliability and security mechanisms, we now present a design to transmit covert data via ICMP Echo request packets. One Approach is to put the protocol header and covert data together, and send it across after proper encryption. But this is only feasible when the underlying covert channel has enough bandwidth. For Example, if we use ICMP echo request data as our covert channel medium, we can do this. We assume here that the maximum size of the data part is 56 bytes (normally ICMP data part sent by Ping command has 56 bytes of data). With 56 bytes of payload, it is easy to squeeze the data, the protocol header and cryptographic checksum (optional) together in ICMP payload. But if the covert channel media is something like TCP timestamps where the available bandwidth is 1 bit per packet, this method is impractical.

In addition, generating echo request packets using this method will cause changing the data part with every packet. In general, every operating system puts it finger print in the data part. Hence, ICMP echo request payload carries a fixed bit pattern. If the data part of ICMP echo request keeps on changing with every packet the covert channel will be easily detectable.

In our design we implement two covert channels in the same packet. One is used for transmitting the protocol header and the other one is used for transmitting data. For example, in an ICMP echo packet, the last byte of ICMP identifier field can be exploited to carry the protocol header and data can be put in ICMP payload. Necessary care must be taken so that the data transmitted through covert channels have the characteristics of the operating system fingerprint. One of the main advantages of this approach is that it can use two different low-bandwidth covert channels together to transmit data reliably across, while neither of these channels alone has enough bandwidth to transmit both the data and the protocol header.

Now, we will look into how the protocol header can be hidden in the ICMP identifier field to make detection difficult. When we say that it is difficult for an intermediary to detect the covert transfer of data, we mean that by simply inspecting all packets it will not be possible for the intermediary to decide with certainty that a particular packet is carrying some covert data and not normal traffic generated by the end hosts.

As most of the ICMP echo request packets are generated by ping program, we look into how the ping program sets the ICMP identifier field. The current implementation of ping in Linux fills the ICMP identifier field with lower 16 bits of the Process ID of the ”ping” process. So each time the ping program is run, it uses a different value in the ICMP identifier field. But all packets sent by the same instance of ”ping” contain the same value in the ICMP identifier field.

In Linux, process IDs are initialized to 300 and incremented by 1 every time a new process is created. When the ID overflows, it is initialized back to 300. All IDs below 300 are reserved for the kernel threads. So, the ICMP identifier field can safely contain random values as the monotonicity of the process IDs won't always be reflected in the lower 16 bits, i.e. it can increase and decrease depending on the particular Process ID in use at that time.

Though monotonicity in the identifier is not absolutely necessary, the last two bytes of process ID wrap around after $2^(16)$ processes. Hence to minimize the anomaly caused by insertion of data, we chose to increase the 1st byte of the ICMP identifier field monotonically, while inserting the protocol header in the last byte of the identifier field.

In Linux if time measuring option is enabled (which is enabled by default) the "ping" program fills the first eight bytes of the payload with current time. First four bytes contain the "seconds" portion in the network byte order and the next four bytes contain the microsecond portion in the network byte order. The rest of the data is filled with values from 8 to 64. So, in the payload, only the first 8 bytes are variable, rest always remains the same.

Hence, we suggest using the lower 4 bytes (where the microsecond portion is stored) and the lowest byte of second portion together to hide our data. As this causes minimal statistical changes in the timestamp, it'll be very difficult to differentiate from the clock skew.

4 Evaluation

We have implemented a prototype of a covert channel using our protocol in Linux ICMP. In this implementation, we embedded our 1-byte protocol header in the lower (second) byte of the ID field in the ICMP protocol header. As discussed earlier, ID field in Linux ICMP header includes the lower two bytes of the process id of the process sending the ICMP packet. Since process ids in Linux can take any value staring from 300 until the largest value that can be accommodated (4 bytes or 8 bytes), all values in the range $0 \ldots (2^{16} - 1)$ can appear in the lower two bytes of legal process ids. Thus, incorporating our one-byte protocol header in the lower byte of the ID field of an ICMP header and monotonically increasing the upper byte of the identifier field, does not make the corresponding ICMP packet unusual, and hence does not raise any suspicion of unusual activity.

An added benefit of updating the ID field in this manner is that every packet that the end application transmits has a different value in the ID field. This prevents an intermediary from inferring that a series of Ping packets are in fact originating from the same process and thus causing suspicion.

Up to five bytes of data are included from fourth through eighth bytes of ICMP payload. As discussed earlier, the first eight bytes in Linux ICMP payload contain the value of current time (seconds and microseconds). By incorporating our covert data, we update the value of the microseconds field (fifth through eighth bytes) and the one byte of the seconds field (fourth byte). It is essential to ensure that this update does not make the corresponding ICMP packet stand out as unusual.

We have run a number of experiments using our prototype implementation to see if incorporating our protocol headers and data in the above-mentioned way causes any statistical difference in the bit patterns of the corresponding ICMP packet from that of a normal ICMP packet. Our experimental set up consists of two Pentium PCs running Linux connected to each other via a 10 Mbps Ethernet hub. In each run, we send random data bytes that are embedded in 5 bytes of ICMP payload. Each data set contains 60 modified timestamp values. It is then compared with the corresponding unmodified timestamps.

Our results show that there is no statistical difference between the bit patterns generated by our covert channel laden ICMP packets and normal ICMP packets. Figure 4 shows the value of the timestamp field in ICMP packet under normal conditions, i.e. when no covert data has been embedded, over 70 different runs. Figure 5 shows the value of the time stamp field in ICMP packet when our protocol has been deployed, again over 70 different runs. Notice that both of these graphs show similar statistical behavior.

Our protocol is extremely light weight in the sense that its resource consumption is similar to that of the underlying communication channel. To demonstrate this, we have measured memory consumption and CPU utilization of our protocol as well as that of Linux Ping program (See Table 1). Here, values of 00.00.00 for CPU utilization indicates that both programs require extremely low CPU time. It is clear that there is no significant difference in memory consumption or CPU utilization between the two programs.

4.1 Telnet Application

To measure the performance of the proposed covert channel, we have implemented a telnet like application over the covert medium. We have captured a shell of the remote machine and execute some well known commands over the covert channel. The final goal is to measure the performance of our application over normal Telnet.

Our application over covert channel has satisfactorily performed execution of the commands without any significant delay. In the current application, we have not followed the Telnet specification thoroughly, hence comparing its performance with telnet will not provide a proper estimation. This is a plan for our future work. Though we could not take a performance measurement during the time of submission of this paper, it gives a fair idea about the potential

Table 1. Resource utilization

Resource	Ping	Covert Channel
Resident Memory	592 kb	584 kb
Shared Memory	500 kb	476 kb
Time	00.00.00	00.00.00

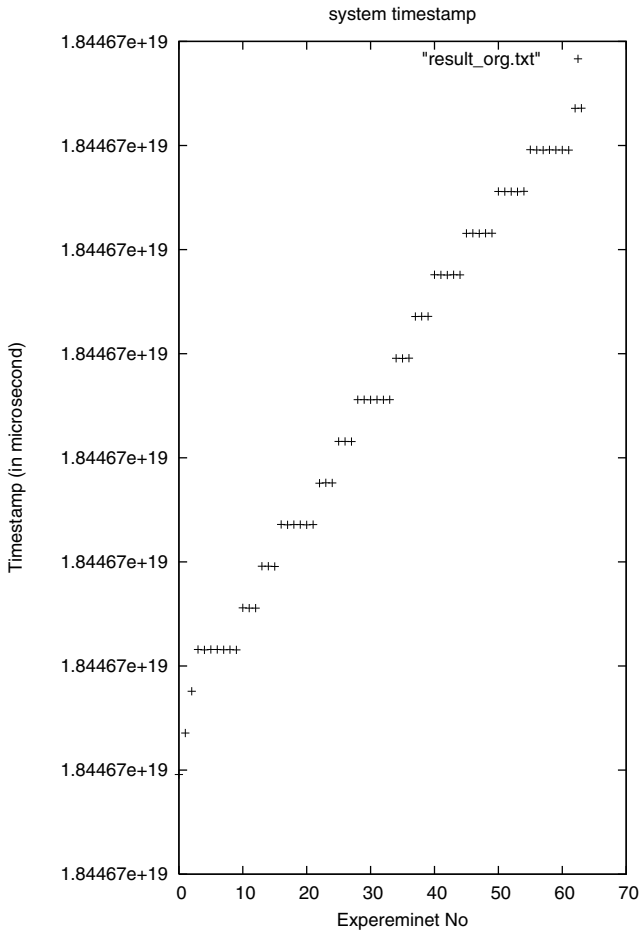

Figure 4. system Timestamp value

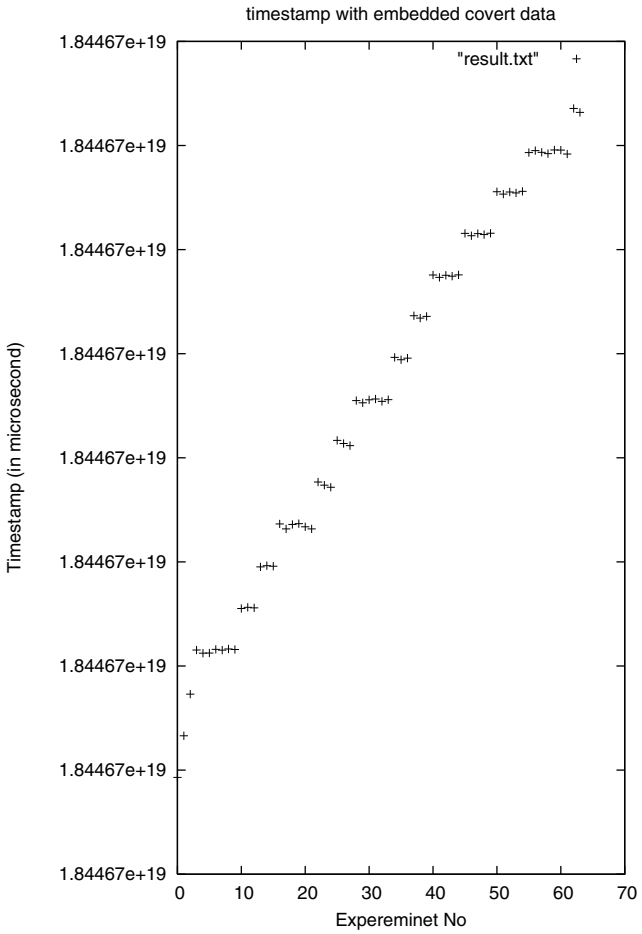

Figure 5. Modified timestamps with embedded covert data

of the proposed covert channel protocol for intelligent data transfer.

5 Discussion

Building trustworthy systems is a huge challenge not only because of the complexities in dealing with individual component failures and determined adversaries launching serious attacks, but also because of the possibilities of covert communication that are extremely hard to detect. In fact, there are so many different and subtle ways in which covert channels can be built that it is practically infeasible to design a system that can reliably detect the presence of covert channels. The only redeeming fact in this respect so far has been that covert communication via covert channels is limited to very low bandwidth, e.g. from one bit to 1 byte per packet. Furthermore, it was not possible to transmit a large number of packets via covert channel with in a short

period of time.

This paper describes the design and implementation of a protocol that not only supports construction of moderate bandwidth covert channels, i.e. five bytes of covert data per packet, but also provides support for reliability and security in covert communication. Our experimental analysis shows that it is not possible to detect this covert channel based on an statistical analysis of the transmitted packets.

Future work includes a detailed analysis of the protocol over both a short-range, local communication network and long-range communication network. In addition, we plan to measure the computation and memory resources consumed by this protocol. Finally, we plan to explore approaches to detect covert channels, and mitigate the effects of a covert channel in highly dependable systems.

References

[1] *Man Ping: Linux manual page of PING*.

[2] K. Ahsan and D. Kundur. Practical data hiding in tcp/ip, 2002.

[3] daemon9 AKA route. Project loki. *Phrack Magazine*, 7(49), August 1996.

[4] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Internet Engineering Task Force: RFC 2460, December 1998.

[5] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through tcp timestamps.

[6] L. Kocarev and G. Jakimoski. Logistic map as a block encryption algorithm. *Physics Letters A*, 289:199–206, Oct. 2001.

[7] C. H. Rowland. Covert channels in the tcp/ip protocol suite. *First Monday*, 2(5), May 1997.