Covert Messaging Through TCP Timestamps

John Giffin[1], Rachel Greenstadt[1], Peter Litwack[1], Richard Tibbetts[1]
({gif,greenie,plitwack,tibbetts}@mit.edu)

Massachusetts Institute of Technology

Abstract. We present a protocol for sending data over a common class of low-bandwidth covert channels. Covert channels exist in most communications systems and allow individuals to communicate truly undetectably. However, covert channels are seldom used due to their complexity. Our protocol is both practical and secure against attack by powerful adversaries. We implement our protocol on a standard platform (Linux) exploiting a channel in a common communications system (TCP timestamps).

1 Introduction

A covert channel is a communications channel which allows information to be transferred in a way that violates a security policy. As a result, covert channels are important methods of censorship resistance. An effective covert channel is undetectable by the adversary and can provide a strong degree of privacy. Often the fact that secret communication is taking place between parties is extremely revealing.

Consider the prisoners' problem, first formulated by Simmons[17]. Alice and Bob are in prison attempting to plan an escape. They are allowed to communicate, but a Warden watches all of their communications. If the Warden notices that they are planning to escape or even suspects them of trying to communicate secretly, they will be placed in solitary confinement.

The prisoners' problem is theoretically interesting and provides a good explanation of the problem that covert channels solve, this problem is increasingly relevant in real world situations. Many governments provide restrictions on the use of cryptography on their systems. The situation is particularly extreme in China, where all ISPs are subject to government control, although electronic systems are increasingly subject to surveillance in all parts of the world as attempts to integrate Carnivore monitoring systems[7] in U.S. ISPs has shown. Private companies increasingly monitor and censor communications with firewalls.

An effective covert channel requires several apparently contradictory properties.

- Plausibility - the adversary must believe that it is not only possible, but likely, that the user of the covert channel is using the medium in which the covert channel is found without sending covert data.

- Undetectability - In order for the channel to be undetectable, the distribution of bits sent must conform to the distribution of the channel itself, otherwise the channel with be able to be automatically detectable via statistical tests.
- Indispensability - The channel must be something which an adversary cannot or will not close off, usually due to the fact that it serves some useful function to the users which are not sending covert data.

TCP timestamps are useful for this purpose because they satisfy these properties to a high degree.

- TCP is ubiquitous protocol used all over the world for the majority of all Internet traffic. It is almost always employed using the timestamp option. As a result, it is extremely plausible that the majority of users using TCP will not be sending covert data and have a good reason for using it.
- On a slow connection, the low order bits of timestamps appear random. As a result, statistical analysis of the headers of the packet will be ineffective in detecting the channel.
- Since TCP is so widely employed, it is likely that users who are not sending covert data would react unfavorably to attempts to modify their TCP packets on a large scale or to limit the timestamp option which is used to ensure high performance.

The channel is broken if it is detected by the adversary. This situation can be modeled as follows: For any two TCP connections t_1 and t_2 on the same network, an adversary cannot distinguish which of the connections is using with probability greater than $\frac{1}{2}$. The channel is also broken if the adversary can deny service to the users of the covert channel while still allowing service to the users of unmodified TCP connections.

The adversary has the power to monitor all the traffic on the network and to modify some traffic. The adversary only views network traffic and cannot see the processing of packets on sender and receiver machines. We consider an eavesdropper who has the power to modify any number of packets, but cannot constantly modify packets sent by a single host.

In the prisoners' analogy, the warden would not be constantly changing every message which passes between prisoners, but might occasionally modify some messages in the hopes of detecting covert communication. The warden lacks the resources to modify every message sent between every prisoner all of the time. If the warden was sufficiently suspicious of one prisoner to modify all of their messages, he would just put that prisoner in solitary and be done with it.

It is notable to realize that if a more powerful adversary than this is willing and capable of either preventing users from using the timestamp option with TCP or overwriting the low order bits of TCP timestamps of every packet, then the adversary will have closed the channel. We assume that the adversary is either unwilling to do this, unable to do this, or will be annoyed by being forced to do this. In addition, we believe that even if this channel is closed, the techniques presented in this paper will be useful in providing reliable communication over other low bandwidth covert channels. It is also useful to realize that even if the adversary denies service to the channel, he still cannot detect whether covert data was being sent regardless of how much data he modifies or snipes.

Most of the interesting work which we have done deals with the problem of sending a message at a rate of one bit per packet over an unreliable channel, and we believe that even if this particular channel is closed the work we have done will be relevant to other similar channels that may be identified.

2 Related Work

Many other channels have been identified in TCP. These include initial sequence numbers, acknowledged sequence numbers, windowing bits and protocol identification.[13][19] These papers focus on finding places where covert data could potentially be sent but do not work out the details of how to send it. Those implementations which exist[13] generally place into header fields values that are incorrect, unreasonable or even outside the specification. As long as the adversary is not looking, this may be effective, but it will stand up to concerted attack, being effectively security through obscurity. These systems do cannot withstand statistical analysis.

The TCP protocol is described in RFC 793.[12] A security analysis TCP/IP can be found in [3] We are certainly not the first group of people to identify the possibility of using the TCP/IP Protocol Suite for the purposes of transmitting covert data. In "Covert Channels in the TCP/IP Protocol Suite,"[13] Craig Rowland describes the possibility of passing covert data in the IP identification field, the initial sequence number field, and the TCP acknowledge Sequence Number Field. He wrote a simple proof-of-concept, raw-socket implementation `covert_tcp.c`. The possibility of hiding data in timestamps is not discussed. We feel that embedding data in the channels identified here would not be sufficient to hide data from an adversary who suspected that data might be hidden in the TCP stream.

In "IP Checksum Covert Channels and Selected Hash Collision,"[1] the idea of using internet protocol checksums for covert communication is discussed. Techniques for detecting covert channels, as well as possible places to hide data in the TCP stream, are discussed (the sequence numbers, duplicate packets, TCP window size and the urgent pointer) in the meeting notes of the UC Davis Denial of Service(DOS)Project[19]

The idea of using timing information for covert channels (in hardware) is described in "Countermeasures and Tradeoffs for a Class of Covert Timing Channels."[8] More generalized use of timing channels for sending covert information is described in "Simple Timing Channels."[10]

Covert channels are discussed more generally in a variety of papers. A generalized survey of information-hiding techniques is described in "Information Hiding – A Survey."[6] Theoretical issues in information hiding are considered in [4] and [2]. John McHugh provides a wealth of information on analyzing a system for covert channels in "Covert Channel Analysis."[9]. The subject is addressed mainly in terms of classified systems. These sorts of channels are also analyzed in “Covert Channels – Here to Stay?”[11]. These papers focus on the prevention of covert channels in system design and detecting those that already exist, rather than exploiting them. G.J. Simmons has done a great deal of research into subliminal channels[17][15][14][16]. He was the first to formulate the problem of covert communication in terms of the prisoners' problem, did substantial work on the history of subliminal communication – in particular in relation to compliance with the SALT treaty and identified a covert channel in the DSA.

3 Design

3.1 Goals

The goal of this system is to covertly send data from one host to another host. There are two important parts to this goal. First, we must send data. Second, we must be covert (i.e. only do things that our adversary could not detect).

It is important to note that these two goals are at odds with each other. In order to send data, we must do things that the receiving host can detect. However, in order to be covert, we must not do anything that an eavesdropper can detect.

We approach this problem by presuming the existence of a covert channel that meets as few requirements as possible. We then describe a protocol to use such a channel to send data. Finally, we identify a covert channel that meets the requirements that we have proposed.

3.2 Characteristics of the Channel

In designing our covert channel protocol, we seek to identify the minimum requirements for a channel which would allow us to send useful data.

In the worst case scenario, the channel would be bitwise lossy, unacknowledged, and the bits sent would be required to pass certain statistical tests. By bitwise lossy, we mean the channel can drop and reorder individual bits. By unacknowledged, we mean that the sender does not know what bits, if any, were dropped and does not know what order the bits arrived in.

Using this channel to send data is extremely difficult. However, if we relax these restrictions in reasonable ways, the problem becomes clearly tractable.

For simplicity, we will assume that the only statistical test that the bits must pass is one of randomness, since this will be convenient for embedding encrypted data. This is reasonable since it is not prohibitively difficult to identify covert channels that normally (i.e. when they are not being used to send covert data) contain an equal distribution of ones and zeros.

We will also assume that each bit has a nonce attached to it and that if the bit is delivered, it arrives with its nonce intact. This condition is both sufficient to make the channel usable to send data and likely to be met by many covert channels in network protocols. The reason why it is an easy condition to meet is that most covert channels in network protocols involve embedding one or more bits of covert data in a packet of innocuous data. Thus, the innocuous data (or some portion thereof) can serve as the nonce.

3.3 Assumptions

We presume that we have a channel with the above characteristics. We further presume that the adversary cannot detect our use of that channel. Lastly, we presume a shared secret exists between the sender and receiver.

The first two presumptions will be justified in sections 3.5 and 5.1 respectively. The third presumption is justified on the grounds that it is impossible to solve the problem without it. This is the case because if the sender and receiver did not have a shared secret, there would be nothing to distinguish the receiver from the adversary. Any message that the sender could produce that was detectable by the receiver could be detected by the adversary in the same manner. Note that public key cryptography is no help here, because any key negotiation protocol would still require sending a message to the receiver that anyone could detect.

We also, assume that it is sufficient to implement a best effort datagram service, such as that provided (non-covertly) by the Internet Protocol. In such a service, packets of data are delivered with high probability. The packets may still be dropped or reordered but, if a packet reaches its destination, all the bits in the packet reach the destination and the order of the bits within the packet is preserved. This level of service is sufficient because the techniques to implement reliability over unreliable datagrams are well understood, and in some applications reliability may not be required.

We now present a method to implement best effort datagrams over a channel with the above characteristics.

3.4 Protocol

In order to send messages over this channel, we send one bit of our message block M per bit of the channel, rather than sending some function of multiple bits. This way, each bit of the data is independent and if one bit is lost or reordered it will not affect the sending of any of the other bits. We choose which bit of the message block to send based on a keyed hash of the nonce. That is, for a message block of size l and a key K, on the packet with nonce t we send bit number n where

$$n = H\left<t, K\right> \pmod{l} \tag{1}$$

The hash function H should be a cryptographic hash function which is collision-free and one-way. Because the nonce T will vary with time, which bit we send will be a random distribution over the l bits in the block. We can keep track of which bits have been sent in the past, in order to know when we have sent all the bits. The expected number of channel bits x it takes to send the l bits of the block will be

$$x = \sum_{i=0}^{l-1} \frac{l}{l-i}. \tag{2}$$

Of course, because our channel loses bits, this is not sufficient. We thus send each bit more than once, calling the number of times we send each bit the occupation number of that bit, o. The probability of our message getting through, p, will be based on the probability that a bit is dropped d and the occupation number o. The probability will be bounded below by $(1-d^o)^l$. Thus for any drop rate, we can choose a sufficiently high occupation number to assure that our messages will get through. And for small drop rates the occupation number does not need to be large to for the probability of successful transmit to be high.

When sending each bit, it must have the same statistical properties as the covert channel has when not being used or else an adversary could use statistical analysis to detect the use of the channel. As we mentioned above, we assume that the channel is normally random. Thus, our bits must appear random. Since much research has been done in finding cryptographic means to make ciphertexts indistinguishable from random distributions, this will be easy. We accomplish this as follows. We derive a key bit k from the same keyed hash of the nonce t in Equation 1, making sure to not correlate n and k.

$$k = \begin{cases} 1 & \left\lfloor \frac{H\langle t,K \rangle}{l} \right\rfloor = 0 \pmod 2, \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The transmitted bit b is the exclusive or of the key bit k and the plaintext message bit M_n. Because k seems random, M_n will seem random, and thus the random characteristic of our channel is preserved.

There are several techniques that the sender can use to determine when a bit has been transmitted.

The sender assumes that a block has been transmitted after it has achieved the occupation number o for every bit in the message. In order for the receiver to know when they have received a block, the last l_c bits of the message are a checksum C of the first $l - l_c$ bits.

3.5 Finding a Covert Channel

In attempting to locate a covert channel we restrict our considerations to covert channels over the network. This is because most of the time the network is the only mechanism through which a pair of hosts can reasonably communicate.

There are two ways that we could transmit information. We could send new packets and try to make them look innocuous, or we could modify existing packets. Obviously, it will be easier to maintain covertness if we modify existing packets. If we were to send new packets, we would need to come up with a mechanism to generate innocuous looking data. If an adversary knew what this mechanism was, they could likely detect our fake innocuous data and our communication would no longer be covert. In contrast, if we modify packets, all packets that get sent are legitimate packets and an adversary will have a more difficult time detecting that anything is amiss. Thus, we choose to modify existing packets.

We can modify existing packets in two ways. We can modify the application data or we can modify the protocol headers. Modifying the application data requires a detailed understanding of the type of data sent by a wide variety of applications. Care must be taken to ensure that the modified data could have been generated by a legitimate application, and we must guess what sort of applications the adversary considers innocuous. It is easier and more general to modify the protocol headers because there are fewer network protocols in existence than application protocols. Most applications use one of a handful of network protocols. Furthermore, the interpretation of protocol header fields is well defined, so we can determine if a change to a field will disrupt the protocol.

The problem remains, however, that we must only produce modified protocol headers that would normally have been produced by the operating system. For example, we could attempt to modify the least significant bit of the window size field of TCP packets. However, most 32 bit operating systems tend to have window sizes that are a multiple of four. Since our modification would produce many window sizes that were not multiples of four, an adversary could detect that we were modifying the window size fields. Similarly, we could attempt to hide data in the identification field of IP packets. However, many operating systems normally generate sequential identification field values, so an adversary could detect the presence of covert data based upon this discrepancy.

For these reasons, we wish to avoid directly modifying packet headers. Instead we observe that more subtle modifications to the operating system's handling of packets can result in a legitimate (and, thus, presumably harder to detect) change in headers. In particular, if we delay the processing of a packet in a protocol with timestamps, we can cause the timestamp to change.

Detecting these delays will likely be very difficult because operating system timing is very complex and depends on many factors that an adversary may not be able to measure – other processes running on the machine, when keys are pressed on the keyboard, etc. Thus, this technique for sending information is very difficult to detect.

We now look at applying this technique to TCP to create a channel with the properties described above.

3.6 TCP Timestamps as a Covert Channel

By imposing slight delays on the processing of selected TCP packets, we can modify the low order bits of their timestamps.

The low bit of the TCP timestamp, when modified in this way, provides a covert channel as described above. The low bit is effectively random on most connections. The rest of the packet, or some subset, can be our nonce. When examined individually, packets (and thus bits) are not delivered reliably.

Because TCP timestamps are based purely on internal timings of the host, on a slow connection their low bits are randomly distributed. By rewriting the timestamp and varying the timing within the kernel, we can choose the value of the low bit. As long as we choose values with a statistically random distribution, they will be indistinguishable from the unaltered values.

The rest of the TCP headers provides a nonce that is nearly free from repetition. The sequence number sent with a TCP packet is chosen more or less randomly from a 2^{32} number space. Thus, it is unlikely to repeat except on retransmission of a packet. Even if it does repeat, the acknowledgment number and window size fields will likely have changed. Even if those fields are the same, the high order bits of the timestamp will likely have changed. It is extremely unlikely that all of the headers, including the high order bits of the timestamp, will ever be the same on two packets.

While TCP is a reliable stream protocol, it provides a stream of bytes that are reliably delivered, rather than guaranteeing reliable delivery of individual packets. For example, if two small packets go unacknowledged they may be coalesced into a single larger packet for the purpose of retransmission. As a result, bits associated with the packets can be dropped, when their packets are not resent. Also, because bytes are acknowledged rather than packets, it is often not clear whether a given packet got through, further complicating the question of whether a bit was delivered.

3.7 TCP Specific Challenges

Rewriting TCP timestamps presents some additional challenges over and above a standard implementation of the protocol from Section 3.4. Timestamps must be monotonically increasing. Timestamps must reflect a reasonable progression of time. And when timestamps are rewritten, it can cause the nonce in the rest of the packet to change.

Timestamps must be monotonically increasing. Because timestamps are to reflect the actual passing of time, no legitimate system would produce earlier timestamps for later packets. Were this done, it could be observed by checking the invariant that a packet with a larger sequence number in a stream also has a timestamp greater than or equal to other packets in that stream. When rewriting timestamps, we must honor this invariant. As a result, if presented with the timestamp 13 and needing to send the bit 0, we must rewrite to 14 rather than 12. Additionally, we must make sure than any following packet has a timestamp of not less than 14, even if the correct timestamp might still be 12.

Timestamps must reflect a reasonable progression of time. Though timestamps are implementation dependent and their low order bits random, the progression of the higher order bits must reflect wall clock time in most implementations. Because an adversary can be presumed to know the implementation of the unmodified TCP stack, they are aware of what the correct values of timestamps are. In order to send out packets with modified timestamps, and keep timestamps monotonically increasing, streams must be slowed so that the timestamps on packets are valid when they are sent. Thus, we can be thought of as not rewriting timestamps but as delaying packets.

As an additional challenge, because we must only increase timestamps, we will sometimes cause the high order bits of the timestamp to change. To decrease the chance of nonce repetition, we include the higher-order bits of the timestamp in the nonce. When incrementing timestamps, these bits may change, and the nonce will change. When the nonce changes, we will have to recompute n and k, and thus may have to further increment the timestamp. However, at this point the low bit of the timestamp will be 0, and so incrementing will not change the nonce. This algorithm can be seen in Figure 1.

3.8 Choosing Parameters for TCP

For a checksum of size n bits, a collision can be expected one time in 2^n. Assuming a sustained packet rate of ten packets per second (an upper bound), we will see a collision every $2^{\frac{n}{10}}$ seconds. We selected our checksum to be a multiple of eight and a power of two to keep the checksum byte aligned and to make it consistent with standard hash functions. A checksum size of 16 bits is clearly too small, as it results in collisions every two hours. A 32 bit checksum raises this time to 13.5 years, which we deem to be an acceptable without making the amount of data per block too small.

4 Implementation

4.1 Sending Messages

Our sender is implemented on top of the Linux kernel. The current implementation of a sender is a minor source modification to provide a hook to rewrite timestamps, and a kernel module to implement the rewrite process, to track the current transmission, and to provide access to the covert channel messaging to applications. The current system only provides one channel to one host at a time, but generalizing to multiple channels should not be difficult.

We selected SHA1 as the hash. It is a standard hash function, believed to be collision resistant and one-way. Source is freely available[5], which made it even more attractive. We needed to put our own interface on SHA1 and modify the code so that it could be used in both the kernel code and in the receiving application.

The basic algorithm is for each packet compute the cipher text bit to be included in that packet according to Figure 2. Then the timestamp is rewritten according to the method described in Figure 1. This is a simple function implementing the rewriting algorithm described in Section 3.7. This algorithm can be seen in the pseudocode of Figure 3, particularly in the recursive call to EncodePacket.

To encode a packet, the timestamp is incremented until it has the proper value to be sent. When a packet is ready to be sent, the occupation number for the bit in the packet is increased. Occupation numbers are tracked in the array TransmitCount. If the minimum occupation number of every bit in the block is ever higher than the required occupation number, the block is presumed received and the next block begins transmission.

4.2 Receiving Messages

The receiving process is designed to be portable and entirely located in userspace. It is much simpler than the sender side and the primary interesting part is

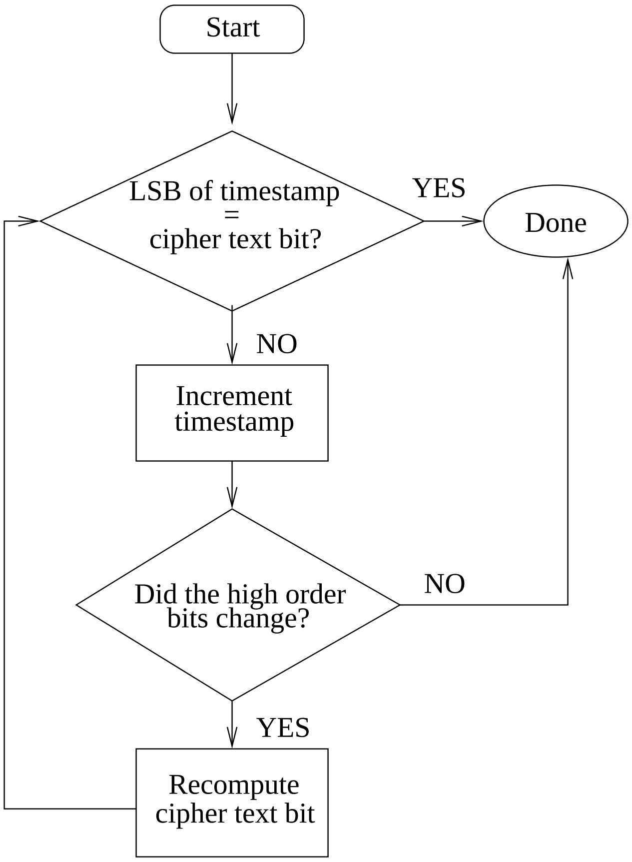

Fig. 1. Rewriting Timestamps

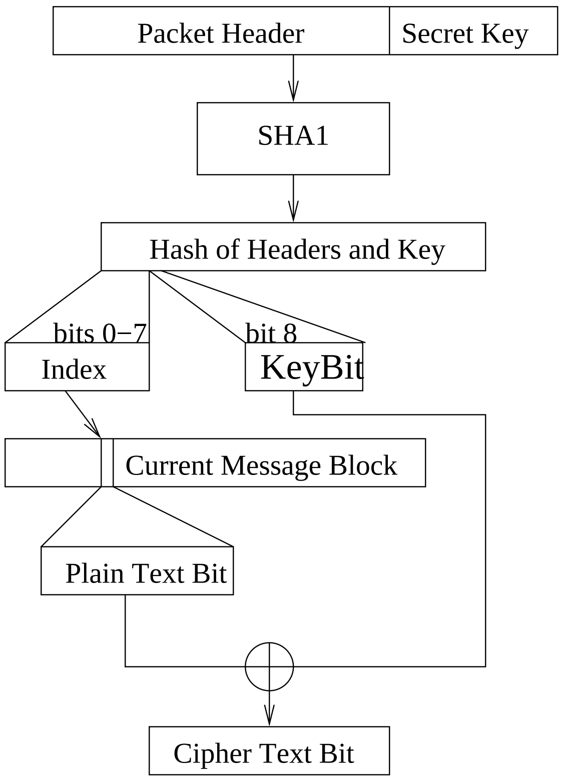

Fig. 2. Sender

ENCODEPACKET(Packet P, TimeStamp T)
  GetHeaders(P) $\rightarrow$ PacketHeader
  SHA1(PacketHeaders)[0-7] $\rightarrow$ Index
  CurrentBlock[Index] $\rightarrow$ PlainTextBit
  SHA1(PacketHeaders)[8] $\rightarrow$ KeyBit
  PlainTextBit $\oplus$ KeyBit $\rightarrow$ CipherTextBit
  if T[0] $\neq$ CipherTextBit **then**
    $T + 1 \rightarrow$ T
    if T[0] = 0 **then**
      return EncodePacket(P,T)
    end if
    TransmitCount[Index]++
    if Minimum(TransmitCount) > MinimumTransmitCount **then**
      NextBlock $\rightarrow$ CurrentBlock
    end if
  end if
  SendPacket(P,T)

Fig. 3. Pseudocode for ENCODEPACKET

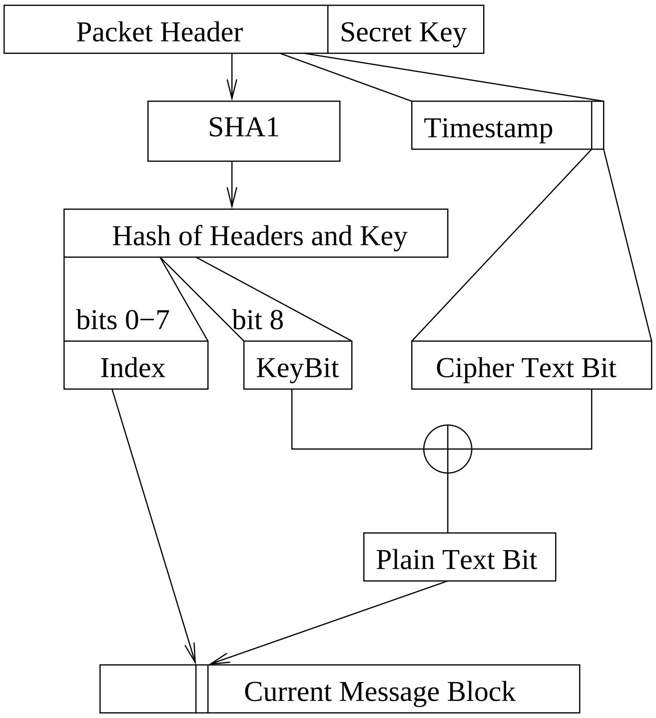

Fig. 4. Receiver

ReceivePacket(Packet P, TimeStamp T)
  GetHeaders(P) → PacketHeader
  SHA1(PacketHeaders)[0-7] → Index
  T[0] → CipherTextBit
  SHA1(PacketHeaders)[8] → KeyBit
  CipherTextBit ⊕ KeyBit → PlainTextBit
  PlainTextBit → CurrentBlock[Index]
  if ValidateChecksum(CurrentBlock) **then**
    OutputBlock
  end if

Fig. 5. Pseudocode for ReceivePacket

determining when we are done with a block and the boundaries between different data blocks.

Packets are collected by the receiver using the `libpcap` interface to the Berkeley Packet Filter[18]. This library is part of the standard utility *tcpdump* and has been ported to a wide variety of platforms. Our receiver is simple C, using only `libpcap` and our SHA1 library. Unlike the sender, it is not tied to the Linux platform, and will probably run anywhere that `libpcap` will run.

The receiver maintains a buffer initialized to all zeroes which represents the current data block to be decoded. As packets are received, the receiver computes the hash of the packet headers concatenated with the shared secret. He then XORs bit 8 of the hash with the low order bit of the timestamp of the packet, he places the result in the buffer at the place indicated by the index.

In actuality, the data block contains less than BLOCKSIZE bits of data. Appended to it is a checksum of the data. The purpose is the checksum is to inform the receiver when he has received the entire valid block and should output plaintext and allocate a new block buffer. The receiver calculates this hash every time he receives a bit and adds it to the buffer. This checksum needs to be collision resistant such that the probability that the receiver will believe he has prematurely found a valid output without actually having done so (either by chance or design by the adversary) is sufficiently low.

5 Evaluation

5.1 Security

The security of this protocol is violated when an adversary can determine what data we are sending or that we are sending data at all.

The system is designed to send out packets in intervals such that the timestamps monotonically increase over time and such that they do not increase quicker than the 10 ms granularity of linux timestamps. We use our ability to delay packets to make certain that the mean time between packets is relatively constant. In a normal link of this speed, it would be expected that the low order bits of the timestamp would be unpredictable. If this is the case, we will have achieved our goal of undetectability.

Two things contribute to the low order bit of the timestamp: the plain text bit and the key bit. Our goal is to xor these two bits together and obtain an unpredictable bit that we can use in our timestamp. Recall that the key bit is the 9th bit of the hash of the packet headers and the secret key. Given a random oracle model for the hash function used by the sender, the key bit will be a random number provided that packet headers do not collide.

Packet headers collide only when all TCP header fields are the same, including sequence number, window, flags, options, source port, destination port, and the high-order 31 bits of the timestamp; the odds of such a collision happening are remarkably small. As long as no such collisions occur, the XOR of the plain text bit with the key bit is essentially a one-time pad. The low order 9 bits of the hash will collide approximately once every 512 packets, but the adversary has no way to detect these collisions without the key, he is left with a $Pr[\frac{1}{2}]$ chance that there is a collision on the 9th bit, in which case he can determine if the two timestamp bits are the same or different. However, he has a $Pr[\frac{1}{2}]$ chance of guessing this anyway.

Should headers collide, one bit of information is revealed about the two bits of plain text encoded in those two packets. Even so, no information is gained about the sender's secret key[1]. Of course if the headers collide often enough the timestamps will display the entropy of the message, which is lower than expected and the process with be detectable. It is meaningful to note that if the headers collide often enough it will take a great deal of packets and a large amount of time to get the message through at all. This might tip off the sender that something odd is up. Of course, too many collisions will make the low order bits of the timestamp have low entropy and the system will be detectable.

We believe the probability of collisions, using a good hash function with the properties specified in our design section, is sufficiently low to avoid the problem of a two time pad. However, if the users were in a situation where the adversary could somehow cause the packet headers to collide, or simply in a system where the packet headers had little entropy, the security of the system could be increased by encrypting blocks of data before sending them (using an encryption mode that does not cause interdependency between blocks). In this way, the entropy of the data being sent would be high. This bit would then be sent as the low order bit of the timestamp and the system would otherwise proceed as before. It is notable to realize that this contributes to the inefficiency of the system because the hash also must still be computed in order to generate the 8 bit index.

5.2 Robustness

The system is also broken if the adversary can deny service to the users of the channel. A powerful adversary could defeat this system by overwriting the tcp timestamps of all packets in some unpredictable way. This is a feasible attack for some powerful adversaries. However, it is not currently being done in any situations of which we are aware. If this paper caused this system to be used so widely that corporations and governments who wished to censor those using networks they controlled were forced to modify all packets, that would be a favorable outcome. The techniques applied in this paper could be used to design another covert channel which would have to be closed, raising the cost of censorship.

Apart from this attack our system is fairly robust. Since all bits are sent independently multiple times, modifications and packet sniping are unlikely to prevent the message from getting through unless performed in large numbers.

[1] This assumes that the hash function used is one-way.

5.3 Performance

After sending 3000 packets, there is a 99.6% chance that we have sent every bit at least once. After sending 5000 packets the probability that we have not sent every bit has dropped to around 1 in a million.

3000 packets may seem like a lot but a single hit on an elaborate website can generate 100 packets or more, especially if the site has many images which must be fetched with individual HTTP GET requests. Furthermore, transfer of a 3 megabyte file will likely generate that many packets. Thus, it is fairly easy to generate enough packets to assure a fairly high probability of successful transmission of a data block.

To send a total of n bits, the message will take approximately $n * 3.75$ms if the sender is not limited by network constraints. 3.75ms is the expected time to wait to send a packet.

6 Conclusion and Future Directions

6.1 Conclusions

We have designed a protocol which is applicable to a variety of low bandwidth, lossy covert channels. The protocol provides for the probabilistic transmission of data blocks.

Identifying potential covert channels is easier than working through the details of sending data covertly and practically through them. The protocol gives a method for sending data over newly identified cover channels with minimal design investment. The implementation of this protocol with TCP timestamps is not yet complete, but we are confident that there are no major obstacles remaining.

6.2 Future Directions

Future directions of our research involve improvements to our implementation and work on channel design that deals with more powerful adversaries and more diverse situations.

It would be useful if the sender in the implementation were able to track, possible via `ack` messages, which data had actually been received by the receiver. If this were the case, the sender would not have to rely on probability to decide when a message had gotten through and when he should begin sending more data.

It would also be useful to develop a bidirectional protocol that provided reliable data transfer. Although it would theoretically be possible to implement something like TCP on top of our covert channel, this would likely be inefficient. Thus, it would be useful to develop a reliability protocol specifically for this type of channel.

We would also like to identify channels which a resource rich active adversary would not be able to close. It would also be useful to deal with key exchange, as our sender and receiver may not have the opportunity to obtain a shared secret.

Our system is currently only practical for short messages; it would be desirable to be able to send more data. Lastly, our protocol is designed to work between two parties. It would be interesting to design a broadcast channel such that messages could be published covertly.

References

1. Christopher Abad. Ip checksum covert channels and selected hash collision. <http://www.gravitino.net/~aempirei/papers/pccc.pdf>, 2001.
2. Ross Anderson and Fabien A.P. Petitcolas. On the limits of steganography. *IEEE Journal on Selected Areas in Communications*, 16:474–481, May 1998.
3. S.M. Bellovin. Security problems in the tcp/ip protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.
4. Christian Cachin. An information-theoretic model for steganography. In David Aucsmith, editor, *Information Hiding, 2nd International Workshop, volume 1525 of Lecture Notes in Computer Science*, pages 306–318. Springer, 1998. Revised version, March 2001, available as Cryptology ePrint Archive, Report 2000/028, url http://eprint.iacr.org/.
5. 3rd D.Eastlake and P. Jones. Us secure hash algorithm 1 (sha1). Rfc, Network Working Group, 2001. <http://www.ietf.org/rfc/rfc3174.txt>.
6. Markus G. Kuhn Fabian A.P. Petitcolas, Ross J. Anderson. Information hiding – a survey. In *Proceedings of the IEEE*, 1999.
7. Federal bureau of investigation - programs and initiatives - carnivore. <http://www.fbi.gov/hq/lab/carnivore/carnlrgrmap.htm>.
8. James W. Gray III. Countermeasures and tradeoffs for a class of covert timing channels.
9. John McHugh. *Covert Channel Analysis*. Portland State University, 1995.
10. Ira S. Moskowitz and Allen R. Miller. Simple timing channels. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 56–61. IEEE Press, May 16-18 1994.
11. I.S. Moskowitz and M.H. Kang. Covert channels – here to stay? In *COMPASS '94*, pages 235–243, 1994.
12. Jon Postel. Transmission control protocol. RFC 793, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, Sep 1981. <http://www.ietf.org/rfc/rfc0793.txt>.
13. Craig H. Rowland. Covert channels in the tcp/ip protocol suite. *First Monday, <http://www.firstmonday.dk/issues/issue2_5/rowland/>*, 1996.
14. G. J. Simmons. The subliminal channels in the u.s. digital signature algorithm (dsa). In W. Wolfowicz, editor, *3rd Symposium on: State and Progress of Research in Cryptography*, pages 35–54, Rome, Italy, February 15–16 1993.
15. G. J. Simmons. Subliminal channels : Past and present. In *European Trans. on Telecommunications, 4(4)*, pages 459–473, Jul/Aug 1994.
16. G. J. Simmons. Results concerning the bandwidth of subliminal channels. *IEEE J. on Selected Areas in Communications, 16(4)*, pages 463–473, May 1998.
17. G.J. Simmons. The prisoners' problem and the subliminal channel. In *CRYPTO '83*, pages 51–67. Plenum Press, 1984.

18. et al Steve McCanne. libpcap, the packet capture library. <http://www.tcpdump.org>.
19. Uc davis denial of service (dos) project, meeting notes. <http://seclab.cs.ucdavis.edu/projects/denial-service/meetings/01-27-99m.html>, January 27 1999.