CLACK: A Network Covert Channel Based on Partial Acknowledgment Encoding

Xiapu Luo, Edmond W. W. Chan and Rocky K. C. Chang
Department of Computing
The Hong Kong Polytechnic University
Hung Hom, Hong Kong, SAR, China
{csxluo|cswwchan|csrchang}@comp.polyu.edu.hk

***Abstract*—The ability of setting up a covert channel, which allows any two nodes with Internet connections to engage in secretive communication, clearly causes a very serious security concern. A number of recent studies have indeed shown that setting up such covert channels is possible by exploiting the protocol fields in the IP, TCP, or application layer. However, the quality of these covert channels is susceptible to unpredictable network condition and active wardens. In this paper, we propose *CLACK*, a new covert channel which encodes covert messages into the TCP acknowledgments (ACKs). Since the message encoding is performed in a TCP data channel, CLACK is reliable and resilience to adverse network conditions. Moreover, CLACK is very difficult to detect in practice, because the TCK ACKs encoded by CLACK cannot be easily distinguished from the normal ACKs. We have implemented and tested CLACK in a test-bed to validate its correctness.**

I. INTRODUCTION

Setting up and detecting network covert channels is an important security problem to consider, because it allows someone inside a fortified network to stealthily fetch information to and from someone outside [2]. A covert channel can also be used to deliver commands to launch a DoS attack [11]. There are two main approaches to sending covert information using the protocols in the network layer and above: *storage* and *timing* channels. In a covert storage channel, the messages are usually embedded into the protocol header fields—[5], [9], [1], [12] for IP storage channels and [9], [10], [12], [6] for TCP storage channels. These approaches, however, are vulnerable to active defense systems [4], [7], [3]. A covert timing channel, on the other hand, relays covert messages based on the timing relationship of the packets. Since our focus is on storage channels, we will not further discuss timing channels.

In this paper, we propose a new storage covert channel called *CLACK* which is designed to meet two main objectives. The first is to provide a reliable covert channel, similar to the reliable data service provided by TCP. That is, each covert message is guaranteed to be decoded correctly, even in the presence of packet losses, jitter, and packet reordering. The second objective is to increase the cost of detecting the covert channels, hopefully to the extent that it becomes practically infeasible to detect them.

Our attack model consists of a covert channel between a CLACK encoder and a CLACK decoder, an active warden, and a server. The encoder behind the active warden attempts to send secretive information to the decoder outside the encoder and active warden's network. To evade the warden's detection, the encoder may establish a normal application session with a server in the decoder's network and embeds covert messages into the application session. By sniffing the application traffic, the decoder can therefore decode the covert messages.

CLACK's message encoding method is more crafty than other storage channels. A CLACK encoder embeds covert information in partial acknowledgments (ACKs) of a TCP data channel and uses the TCP data sent from the server as acknowledgments to the covert message transmissions. Therefore, a CLACK encoder only needs to receive data and send pure ACKs, for example, retrieving documents from websites or FTP sites. In order to detect it, the active warden has to keep states about the send and receive states. It is also difficult for the warden to modify the ACKs without affecting the connection.

We organize the rest of the paper as follows. Section II first presents CLACK and details the design issues involved and practical considerations. Section III then presents the test-bed results to verify CLACK's correctness. Section IV concludes this paper with future direction.

II. CLACK: A NEW STORAGE COVERT CHANNEL

A. The basic approach

A CLACK encoder writes a covert message in the TCP ACK field. Therefore, a CLACK encoder is a TCP receiver, and a CLACK decoder is a TCP sender. However, unlike the ACK bounce method [9], CLACK is based on a persistent flow of TCP data. Therefore, a direct encoding method is not a viable approach for CLACK, because the ACKs have to continue to serve its acknowledging function. Instead, we have designed CLACK based on *partial ACK encoding*. To clearly explain the basic approach, we assume for the time being that all transmissions are perfect, i.e., lossless, packet order preserved, and no duplicate packets. Furthermore, the server always has data to send and its Nagle algorithm is turned off.

During the TCP handshaking, the TCP sender can determine its effective maximum segment size (MSS), denoted by `EFF.SND.MSS`, from the exchange of the MSS values. The CLACK encoder usually selects a smaller MSS to advertise (e.g., 1460 bytes or even 536 bytes) in order to dictate the value of `EFF.SND.MSS`. Furthermore, the encoder advertises a fixed receive window (`RCV.WND`) size of `EFF.SND.MSS` bytes to the sender to constrain the number of data segments sent from the sender each time to one. A data segment of size equal to `EFF.SND.MSS` bytes is referred to as a full-sized segment.

As a result of the settings induced by the encoder, the data segments and ACKs are sent in a stop-and-wait manner, as depicted in Fig. 1. Moreover, all the ACKs are partial. Let the sequence number of S_i be s_i and the value of A_i be a_i. Note that the segment size of S_i is given by $s_{i+1} - s_i$. A *partial ACK* A_i is one for which $a_i < s_{i+1}$, whereas a full ACK A_i is one for which $a_i = s_{i+1}$.

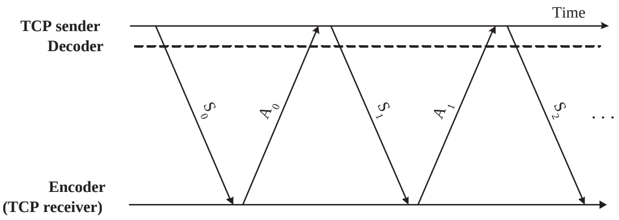


Fig. 1. Data segments and ACKs are sent alternately during the transmission of covert messages in CLACK.

The main novelty of CLACK's design is a clever way of crafting covert messages into the ACKs. Figs. 2(b)-(c) depict the sequence number (SN) space when covert messages $M_i, i \geq 1$, are sent out through the partial ACKs. Let the value of M_i be m_i which is a nonnegative integer. The numbers inside ( ) indicate the event sequence. In Fig. 2(b), for example, A_0 fully acknowledges S_0. After that, the encoder starts sending the covert messages. The first two covert messages, for example, are embedded into the ACKs as

$$a_1 = s_2 - m_1 \quad \textit{and} \quad a_2 = s_3 - m_2.$$

That is, m_i is represented by the amount of S_i that is left *unacknowledged* by A_i. Note that the covert message encoding method in Fig. 2(c) is exactly the same, although A_2 is also acknowledging the data in S_1. It is also instructive to compare the scenario of no covert messages in Fig. 2(a) with the other two.

B. The CLACK encoder

Now we turn to a more detail description of the CLACK encoder. As mentioned earlier, a CLACK encoder sets its receive window size to $\mathtt{RCV.WND} = \mathtt{EFF.SND.MSS}$ which limits the number of data segments sent each time to one. However, our measurement results show that some TCP senders (e.g., web servers) would return more than one packet at the beginning of the slow start which will disrupt the stop-and-wait transmission pattern. Therefore, the encoder must first receive a *single* full-sized data segment before it can starting encoding covert messages. For example, S_1 in Figs. 2(b)-(c) is a full-sized segment.

Let the first full-sized segment be S_1, and the encoder starts sending covert messages $M_i, i \geq 1$. Therefore, $A_i, i \geq 1$, are all partial ACKs, i.e., $a_i < s_i, i \geq 1$. From the examples in the last section, it is clear that the partial ACKs are given by $a_i = s_{i+1} - m_i$. Thus, the encoder is able to craft a partial ACK corresponding to m_i after receiving S_i. To use the typical state variable for a TCP receiver, we let $\mathtt{RCV.NXT}_{i+1}$ be the expected SN to receive in S_{i+1} or $\mathtt{RCV.NXT}_{i+1} = s_{i+1}$. Therefore, the encoding rule for M_i is given by

$$a_i = \mathtt{RCV.NXT}_{i+1} - m_i. \tag{1}$$

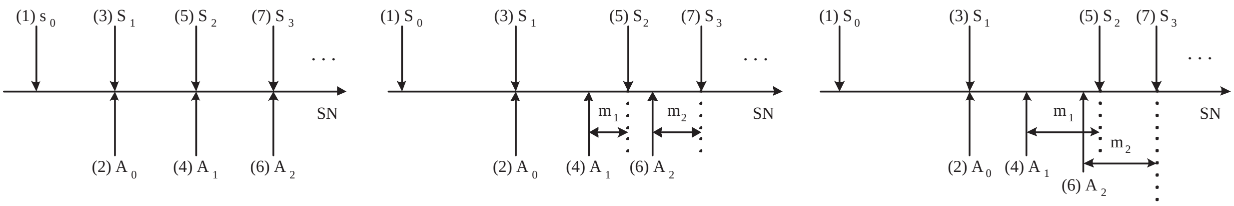
Whenever receiving a partial ACK, the sender will send out a new data segment to fill up the send window again. Therefore, the amount of outstanding data segments is always equal to `RCV.WND` = `EFF.SND.MSS` before the encoder crafts a new partial ACK. However, the encoder will not use the two values—0 and `RCV.WND`. The use of 0 would be confused with the case of no covert message for which a full ACK is sent, i.e., $a_i = \mathtt{RCV.NXT}_{i+1}$. The use of `RCV.WND`, on the other hand, would produce duplicate ACKs, because $a_i = \mathtt{RCV.NXT}_{i+1} - \mathtt{RCV.WND} = a_{i-1}$. Therefore, the acceptable range for m_i is

$$0 < m_i < \mathtt{EFF.SND.MSS}. \tag{2}$$

C. Achieving covert channel reliability

An important feature of CLACK is its provision of reliable covert channel communication. The stop-and-wait communication pattern, first of all, reduces the reliability problem complexity. For instance, packet reordering would not affect the decoding correctness. Moreover, since the covert messages are encoded into the partial ACKs, CLACK uses the data segment as an acknowledgment to the covert messages. Therefore, the roles of the TCP ACKs and TCP data segments are exactly reversed for the covert messages in CLACK. Consider again the examples in Figs. 2(b)-(c). The recipient of nonretransmitted S_2 ensures to the encoder that A_1 (and therefore M_1) has been received correctly by the sender (and therefore the decoder). In general, a nonretransmitted S_{i+1} serves as an "acknowledgment" for A_i.

The encoder is able to distinguish nonretransmitted data segment from a retransmitted one. Consider that the encoder sends out a partial ACK A_i. If the next data segment's SN is equal to $\mathtt{RCV.NXT}_{i+1}$ or a_i, the encoder confirms that M_i has been received correctly. In other words, the data segment is new to the encoder. Otherwise, the encoder will retransmit M_i. We use the two cases in Fig. 3, which correspond to the scenario in Fig. 2(b), to illustrate CLACK's reliability mechanism. In Fig. 3(a), the partial ACK A_2 is lost. As a result, the sender timeouts and retransmits S_2. However, due to the partial ACKs, the retransmitted S_2 is not identical to the originally transmitted S_2. The latter's SN is s_2, but the former's is a_1 (recall that $a_1 < s_2$). Therefore, the retransmitted S_2's SN is considered old, i.e., it is neither $\mathtt{RCV.NXT}_3$ nor a_2. As a result, the encoder is required to retransmit M_2.



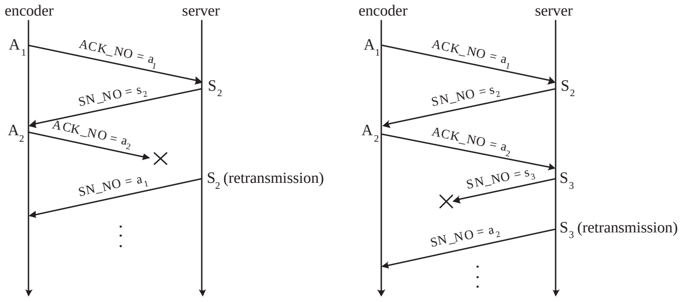
(a) No covert messages.

(b) Transmission of covert messages start in A $_1$: Scenario 1.

(c) Transmission of covert messages start in A $_1$: Scenario 2.

Fig. 2. Encoding of covert messages M_1 and M_2 in CLACK.

On the other hand, data segment S_3 is lost in Fig. 3(b). As a result, the sender times out and retransmits S_3. Similar to the first case, the original S_3 is different from the retransmitted S_3. However, unlike the first case, the retransmitted S_3 is considered new, because its SN is equal to a_2. Therefore, the encoder can continue to send the next covert message. In other words, the data segment loss does not affect the covert channel's reliability.



(a) An ACK loss.

(b) A data segment loss.

Fig. 3. Recovering covert messages in CLACK due to packet losses.

D. Extension to Nagle-enabled senders

So far we have considered Nagle-disabled sender who is expected to send a new data segment immediately after receiving a partial ACK. However, a Nagle-enabled sender may be prevented from sending a nonfull-sized data segment if there are still outstanding segments, thus disrupting the data-ACK sequence in the CLACK channel. The solution to this problem is, first of all, to double the value of `RCV.WND`, i.e., $\texttt{RCV.WND} = 2 \times \texttt{EFF.SND.MSS}$. Second, the encoder is required to send a partial ACK that acknowledges at least `EFF.SND.MSS` bytes of data. In this way, the sender could always return a full-sized data segment. Moreover, the encoder uses the ACK-every-other-segment strategy which sends one ACK for every two full-sized segments.

Consider Fig. 4 for an example. We again first assume that the TCP transmissions are perfect. The sender first sends two full-sized data segments S_0 and S_1, whose SNs are s_0 and s_1, respectively. Upon receiving them, the encoder sets $\texttt{RCV.NXT}_2 = s_0 + 2 \times \texttt{EFF.SND.MSS}$ and sends the first covert message M_1 in the partial ACK A_1. Similar as before, the ACK value is encoded as $a_1 = \texttt{RCV.NXT}_2 - m_1$. For $m_1 \leq \texttt{EFF.SND.MSS}$, $a_1 \geq \texttt{EFF.SND.MSS}$. Thus, this partial ACK will clock a full-sized data segment from the sender, which is S_2. The encoder, upon receiving S_2, sends M_2 in A_2 which is again a partial ACK with $a_2 \geq \texttt{EFF.SND.MSS}$. As a result, we again have the stop-and-wait transmission pattern for the data segments and ACKs.

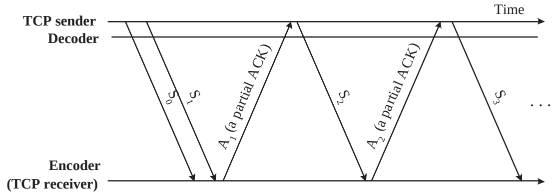


Fig. 4. Data-ACK sequence in CLACK channel with a Nagle-enabled sender.

Similar to the case of Nagle-disabled senders, the encoder here can start sending covert messages only after receiving two consecutive full-sized data segments from the sender. Recall that the receive window size is equal to $2 \times \texttt{EFF.SND.MSS}$ and the partial ACK's value has to be at least equal to `EFF.SND.MSS`. As a result, we have $m_i \leq \texttt{EFF.SND.MSS}$. We again do not use $m_i = 0$ which would be confused with the case of no covert messages. Therefore, the acceptable range of m_i for the case of Nagle-enabled senders is

$$0 < m_i \leq \texttt{EFF.SND.MSS}. \qquad (3)$$

Note that Eq. (3) and Eq. (2) are almost identical. Finally, the encoder must apply the same mechanisms to handle packet loss events as discussed in Section II-C. After the encoder has transmitted an ACK to fully acknowledge the retransmitted data segment, the encoder has to wait for two consecutive, full-sized data segments before it can continue encoding covert messages. Once again this does not affect the covert message encoding and decoding.

E. The CLACK decoder

A TCP sender keeps variables `SND.NXT` and `SND.UNA` for every connection. `SND.NXT` is the SN of the next data segment to be sent, and `SND.UNA` is the oldest unacknowledged SN. In order to validate the ACK sent by the encoder, a CLACK decoder keeps track of the sender's `SND.NXT` and `SND.UNA`. We use $\texttt{SND.NXT}_D$ and $\texttt{SND.UNA}_D$ to denote the two respective variables recorded by the decoder.

The decoder updates $\texttt{SND.NXT}_D$ by examining the SN and packet length in every data segment from the sender. In the other direction, the decoder, upon receiving a copy of ACK A_i from the encoder, first validates the ACK by confirming that $\texttt{SND.UNA}_D < a_i \le \texttt{SND.NXT}_D$. After passing the test, the decoder determines whether A_i is a partial ACK by comparing a_i with $\texttt{SND.NXT}_D$. In the case of a partial ACK, the decoder retrieves the covert message from $\texttt{SND.NXT}_D - a_i$. Lastly, it sets $\texttt{SND.UNA}_D = a_i$.

III. EXPERIMENTAL EVALUATION

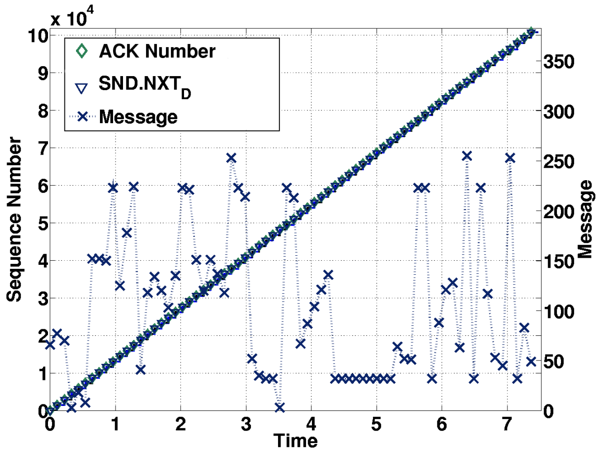
In this section, we evaluate CLACK's decoding accuracy and performance by conducting extensive experiments on our test-bed which hosts of an IP router. An encoder is connected to the one side of the router; a web server and a decoder are connected to the other side. Dummynet [8] is installed in the router to generate various network conditions, including packet loss, delay jitters, and packet reordering. The RTT between the encoder and the server is 32 milliseconds. The bandwidth between the router and the encoder is 100 Mbps, whereas that between the router and the web server/decoder is 10 Mbps. The web server has enabled the Nagle algorithm. The router is configured with a droptail queue size of 30 packets.

We have implemented CLACK encoder as a web client using raw sockets. A CLACK encoder starts the transmission after sending an HTTP `GET` command to request a large HTTP document from the server. The HTTP documents are large enough for the encoders to complete the covert communication with the decoders.

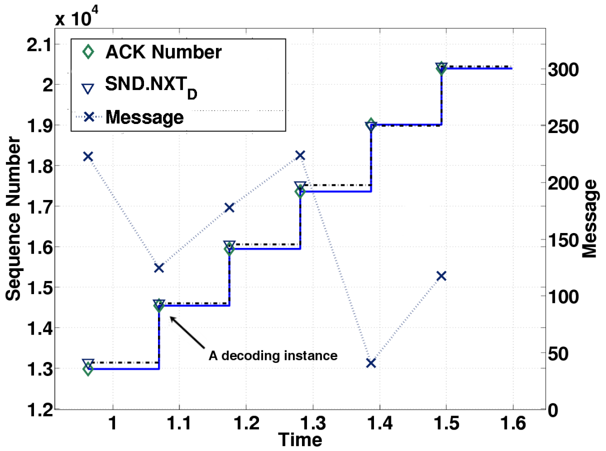
In evaluating CLACK, we have chosen a 256-value CLACK channel to transmit a bitmap file of 70 bytes, and each covert message is of at most 8 bits. Therefore, the CLACK encoder only needs to successfully transmit 70 partial ACKs for the bitmap file. Moreover, it sends a full ACK to acknowledge all the outstanding data segments whenever it receives a retransmitted segment.

Fig. 5(a) presents a data trace observed by the decoder at each decoding instance (i.e., when an ACK arrives). There are three types of data: the ACK value ($\Diamond$), $\texttt{SND.NXT}_D$'s value ($\bigtriangledown$), and the message decoded from the ACK ($\times$). The first two data values come from the TCP sender's SN, and their labels are given on the left y-axis, whereas the labels for the message values are given on the right y-axis. Note that each covert message is decoded by the difference between an ACK value and $\texttt{SND.NXT}_D$, as discussed in the last section.

Due to the scale of the figure, the ACK values and $\texttt{SND.NXT}_D$ values are overlapped with each other Fig. 5(a). We have therefore plotted a small segment in Fig. 5(b) to observe the actual trends. As shown, both increase in steps, because each ACK acknowledges consecutive SNs at the same time. However, the two do not totally overlap, because their differences are used to embed covert messages.



(a) Transmitting M_1 to M_{70}

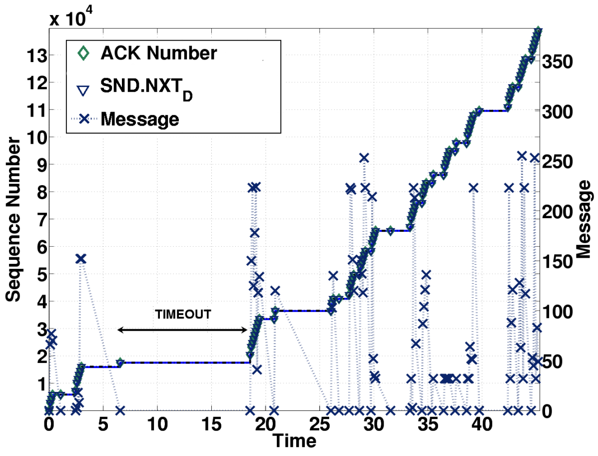


(b) Transmitting M_{10} to M_{15}

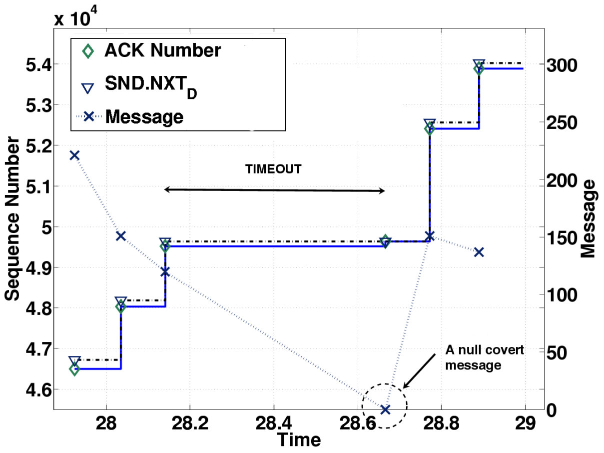
Fig. 5. Experiment results for a CLACK channel under lossless condition.

Fig. 6 presents a data trace for another set of experiments with a packet loss ratio (PLR) of 0.04. As expected, the TCP connection suffers from severe packet losses and frequent retransmission timeouts during the experiment period. Therefore, the decoder records a number of null covert messages which have the message value of 0. To delve into one such case, Fig. 6(b) shows that the decoder receives covert message M_{23} at around 28.1 seconds. After that, the server timeouts and retransmits the lost segment. Since the encoder crafts a full ACK, instead of a partial ACK, in response to each incoming retransmitted segment, the decoder receives a null covert message at around 28.7 seconds. After that, the encoder transmits the next covert message which is received by the decoder at around 28.8 seconds.

Fig. 7 depicts the data rates of the CLACK channel for $\texttt{EFF.SND.MSS} = \{365, 730, 1460\}$ and PLR $= \{0, 0.02, 0.04, 0.06, 0.08\}$. The figure plots an average of the results obtained from three independent experiments. The results show that when the network is lossless, the maximum data rate is about 229.3 bps. Note that the CLACK encoder



(a) Transmitting M_1 to M_{70}



(b) Transmitting M_{21} to M_{25}

Fig. 6. Experiment results for a CLACK channel under a PLR of 4%.

conveys an 8-bit covert message in each RTT during this experiment. Thus, the theoretical data rate is $8\times(1/0.032) = 250$ bps, which is quite close to the experiment result. Moreover, the channel data rate can be further increased by embedding a larger covert message in each partial ACK. For instance, when the maximum value of `EFF.SND.MSS` is 1460 under the same network environment, the theoretical maximum data rate can be increased to 44.6 Kbps.

Besides, CLACK provides lossless covert communication at the expense of a lower data rate. Fig. 7 shows that the data rate drops with the PLR. With PLR $= 0.08$, for example, the data rate drops to at most 29.3 bps. Moreover, using a smaller `EFF.SND.MSS` generally yields a higher data rate, except for the case of `EFF.SND.MSS` $= 0.08$. A smaller data segment requires a shorter time to transmit; therefore, the next partial ACK can be clocked in a faster manner. The exception is due to the adverse effects of the severe packet losses and retransmission timeouts which outweigh the benefit gained from using a smaller `EFF.SND.MSS`.

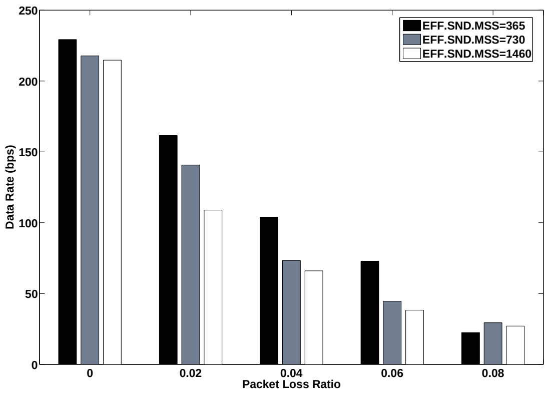


Fig. 7. CLACK's data rate versus packet loss ratio.

IV. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed CLACK, a new covert channel via TCP data channel. Previous approaches based on direct encoding in IP and TCP header fields are susceptible to various unpredictable network events. In contrast, CLACK's partial encoding method provides reliable covert channels and is resilient to packet reordering and variable network delay. We have implemented and validated CLACK on a test bed. One of the important benefits of basing covert channels on TCP is the difficulty of detecting them without keeping states about the connection. Therefore, an important future work is designing algorithms to detect CLACK with minimal state information.

ACKNOWLEDGMENTS

This work was partially supported by a grant from The Hong Kong Polytechnic University (project code U386).

REFERENCES

[1] K. Ahsan and D. Kundur. Practical data hiding in TCP/IP. In *Proc. Workshop on Multimedia Security*, 2002.
[2] K. Borders and A. Prakash. Web Tap: Detecting covert Web traffic. In *Proc. ACM CCS*, 2004.
[3] D. Watson, M. Smart, G. Malan, and F. Jahanian. Protocol scrubbing: Network security through transparent flow modification. *IEEE/ACM Trans. Networking*, 12(2):261–273, April 2004.
[4] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil. Eliminating steganography in Internet traffic with active wardens. In *Proc. IH*, 2002.
[5] T. Handel and M. Stanford. Hiding data in the OSI network model. In *Proc. IH*, 1996.
[6] J. Giffen, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging in TCP. In *Proc. PET*, 2002.
[7] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, 2001.
[8] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), Jan. 1997.
[9] C. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday: Peer-reviewed Journal on the Internet*, 2(5), 1997.
[10] J. Rutkowska. The implementation of passive covert channels in the Linux kernel. In *Proc. Chaos Communication Congress*, 2004.
[11] S. Cabuk, C. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proc. ACM CCS*, 2004.
[12] S. Murdoch and S. Lewis. Embedding covert channels into TCP/IP. In *Proc. IH*, 2005.