Towards an estimation of the accuracy of TCP reassembly in network forensics

Gérard Wagener
University of Luxembourg
gerard.wagener@ses-astra.com

Alexandre Dulaunoy
SES ASTRA
a@foo.be

Thomas Engel
University of Luxembourg
Thomas.Engel@uni.lu

Abstract

Today, honeypot operators are strongly relying on network analysis tools to examine network traces collected in their honeynet environment. The accuracy of such analysis depends on the ability of the tools to properly reassemble streams especially TCP sessions. Network forensics analysis quality is tight to those tools and we evaluated widely used network analysis tools. We pinpoint TCP reassembly errors with their causes and propose algorithms and analytical techniques to measure them in order to improve network forensic analysis.

1 Introduction

In network forensics, network packets are captured and analyzed in a later stage [2]. Full packet capture is more granular than Netflow exports [1]. Crovella et al. claim that in a context of network trace analysis, it is mandatory to have correctly working software that is able to reconstruct network traces [5]. Network captures are sometimes done by other people than those who are doing the capture analysis. In some cases, the network capture tool was not well calibrated during the capture. Intuitively two kinds of potential errors are introduced during network capture analysis. The first class of errors is caused by the capture operator and the root cause of the second class is the presence of software implementation flaws in network analysis tools.

The contribution of this paper is to estimate the accuracy of TCP reassembly. Thus we start to pinpoint discovered TCP reassembly errors, followed by proposals of algorithms and equations to measure them, aiming to improve network forensics analysis. We validated our approaches with experiments based on popular research tools, like `Tcptrace` and `Tcpflow` [1]. We monitored a host for 47 days in order to simulate an high-interactive honeypot [1] and discovered various TCP reassembly challenges.

In section 2 is sketched a state of the art for TCP reassembly. Discovered problems and proposed solutions are enumerated. Flow definitions and relations between them are presented in section 3, followed by a TCP reassembly model. This model was designed to deal with reassembly challenges and its purpose is estimate TCP reassembly errors. Section 4 describes the experiments we did with our network traces. This article is concluded in section 5 and future work activities are enumerated.

2 Related work

An intersection of network forensics and intrusion detection is the process of TCP reassembly. This is an non trivial task [4]. Many problems can emerge during the transmission of a packet. They can be delivered out of order, replicated and corrupted.

TCP is a complex protocol and many different implementation flavors exits. Most TCP stacks behave differently in various scenarios [4]. In network forensics additional constraints are present. A first constraint is that some ambiguities can emerge during the reassembly process due to the lack of knowledge of the network topology. In other words packets might be captured that never reached their target. Network normalization techniques are proposed to remove such ambiguities [8]. A second constraint is that incomplete network packets can be captured [11] which influence the reassembly process. A third constraint emerges when attackers try to circumvent the reassembly process [7, 12]. Song et al. [12] describe attacking methods in order to test network intrusion detection systems aiming to see whether they are vulnerable or not. Sarang et al. tackled the problem of TCP reassembly when attackers are present [7]. They focus on attacks against available memory for intrusion detection systems. They established equations aiming to compute the number of resources available to the attacker for being successful. Beside available memory, it is essential to have a correct implementation of the TCP protocol. In this article we focus on implementation flaws of TCP reassembly and we elaborated a methodology to measure them.

On the one hand, in intrusion detection systems, the stream reassembly and the stream analysis are closely linked in one system. On the other hand, in network forensics, one system performs the TCP reassembly and an-

other one performs the stream analysis. It is obvious that a mandatory requirement is that the two systems use the same flow definition. M.I. Cohen et al. propose a forensics package melting network forensics, memory forensics and disk forensics [2]. Eric Cronin et al. discovered that network eavesdropping is a non-trivial task. They also discuss that the party intercepting communications can be confused. In this case the sender communicates with the recipient in such a way that the eavesdropper cannot understand or misinterprets the communications. Different and exotic hardware or operating system can cause a similar phenomenon. Thus we created a TCP reassembly model and a methodology to measure potential reassembly errors.

3 Monitoring flows

In this section we are going to define the network monitoring terms we are using in this article.

3.1 Flow identification and sessions

An unidirectional IP flow is a set of IP packets and can be characterized by a 5-tuple [3, 5, 6] (Source IP address, Source port, Destination IP address, Destination port, Protocol) $\in I$. The protocol parameter identifies which protocol is used. Frequent used protocols are *TCP* and *UDP*. There is a mathematical relationship $\mathcal{F}$ between captured packets and flow identifier shown in relation 1 where $\hat{P}$ is the set of captured packets and I is the set of flow identifier.

$$(I, \mathcal{F}, \hat{P}) \subset I \times \hat{P} \tag{1}$$

The 5-tuple identifier is extended in Netflow [3] by the addition of N other parameters like ingress interface and type of service. Assuming that all parameters can be represented by numbers, the relationship $\mathcal{F}'$ between a general flow identifier and a set of captured packets is shown in relation 2.

$$(\mathbb{N}^N, \mathcal{F}', \hat{P}) \subset \mathbb{N}^N \times \hat{P} \tag{2}$$

In this article we focus on TCP flows which are defined as a set of TCP packets identified by a 4-tuple (Source IP address, Source port, Destination IP address, Destination Port).

3.2 TCP reassembly model

The main purpose of TCP is to serve as transport layer and to guarantee that a data stream is correctly transfered to a given destination, using an unreliable network. Due to the diversity of TCP reassembly designs let R be the set of reassembly functions. A reassembly function maps TCP packets with streams. The purpose of such a function is to recover the initial stream, emitted by the sender, from captured TCP packets. Let $\hat{P}$ be the set of captured TCP packets. $\hat{P} = \{p_1, \ldots, p_t, \ldots, p_T\}$. A packet that was captured at time t is noted p_t. A packet contains checksums in order to detect transmission errors. These checksums are verified with the function ω which returns the value 1 if the packet has a correct checksum and 0 otherwise. The set of TCP packets with a correct checksum is defined in equation 3. In a network capture we only have the checksum information to see whether the packet is correct or broken, although it was shown that checksums are not always reliable [13].

$$P = \{p_i \in \hat{P} \mid \omega(p_i) = 1\} \tag{3}$$

For a given set of captured packets, P and a reassembly function $\rho \in R$ there is a set S that includes the data streams, recovered from TCP packets.

A reassembly function $\rho \in R$ reassembles TCP packets and is defined in equation 4 such that $k, j \in \{1, 2, 3, \ldots, N\}$. The number j identifies the flow and the index k identifies the offset in the stream.

$$\begin{aligned} &\rho : P \to S \\ &\rho \mapsto f_k^j \end{aligned} \tag{4}$$

Furthermore we define a function σ, shown in equation 5, that maps TCP packets to TCP sessions. A TCP session starts with a connection establishment and finishes with a connection close, like it is described in the RFC 793. The variable s_k holds TCP packets belonging to a TCP session. $s_k = \{p_i | \{(p_i, k)\} \subset \mathcal{F}\}$. Thus the set E contains subsets of TCP packets that belong to a TCP session. An algorithm, to extract these subsets is proposed in section 4.1.1.

$$\begin{aligned} &\sigma : P \to E \\ &\sigma \mapsto s_k \end{aligned} \tag{5}$$

Each session is mapped to a data stream, defined in function 6. Ideally the data stream, should be identical with the data stream emitted by the sender.

$$\begin{aligned} &\eta : E \to S \\ &x \mapsto \eta(x) \end{aligned} \tag{6}$$

A reassembly function ρ is composed of the session function and the session mapping function, shown in proposition 7.

$$\begin{aligned} &P \to S \\ &\rho = \sigma \circ \eta \end{aligned} \tag{7}$$

3.3 TCP reassembly challenges

TCP reassembly is a difficult task. Although a standard specification of the protocol, described in RFC 793, there are different implementations. Each reassembly tool has its

own specification of stream. The tool `Tcptrace` matches a session with a stream, the tool `Tcpflow` links one tuple with one stream. The tools `Tcptrace` and `Tcpflow` put data sent from the sender to the receiver in one stream and data from the receiver to the sender in another one. A stream generated by the tool `Wireshark`, the successor of Ethereal [1], puts data sent from the sender and from the receiver in one stream.

Furthermore some implementations might be defectively coded. Applying a code checker, like Valgrind [10] on reassembly tools, software errors could be detected, like memory leaks, the use of invalid file descriptors and the use of uninitialized memory. By observing such phenomena we are searching for strategies to estimate TCP reassembly errors and verify reassembled streams.

3.3.1 Multiple sessions per flow

High-interactive honeypots, resources which purpose is to attacked, are frequently monitored by capturing the traffic towards them. When we apply the traditional flow relation, defined in equation 1, we notice that the source/destination IP address and the destination port is constant for the given monitored resource. We are interested in the case where the same source port is reused. The set of packets that belong to multiple sessions using the same source port is described in equation 8.

$$\begin{aligned} &p_a, p_b \in P \\ &M_p = \{(\sigma(p_a) \neq \sigma(p_b)) \wedge (\rho(p_a) = \rho(p_b))\} \end{aligned} \tag{8}$$

The phenomenon of having multiple sessions per flow might be reduced to the birthday problem [9], assuming that the source port distribution is uniform. It consists of the probability P_b of finding at least two streams, in a set of n streams belonging to the same flow. The applied birthday problem is formulated in equation 9. $P_r = 2^{16} - 1024 - 1$ represents the TCP port range an operating system can choose as source port.

$$P_b = 1 - \frac{P_r!}{P_r^n (P_r - n)!} \tag{9}$$

3.3.2 Corrupted streams

As we have defined in equation 4 the payload of TCP packets are put in a stream at an offset defined in the TCP header. Moreover header information indicates corrupted or duplicated packets and out-of-order received packets.

We are looking for a verification process for reassembly functions in order to detect wrongly reassembled streams. On the one hand for estimating the accuracy of the reassembly of streams we can compare reassembled streams with different independent tools. On the other hand we want to be able to understand why streams were reassembled differently at the end of the reassembly process. Therefore we aim to have methods to check streams. The input of these methods is composed of a reassembled stream and the raw captured packets. We put the payload of TCP packets in a vector $\overrightarrow{b}$ shown in figure 1. This vector represents one TCP session. $p_i \in P$, $\overrightarrow{b} = (\sigma(p_i))$. The packets are put at arrival order in the vector. A TCP packet p_i is a tuple (TCP header, and TCP payload).

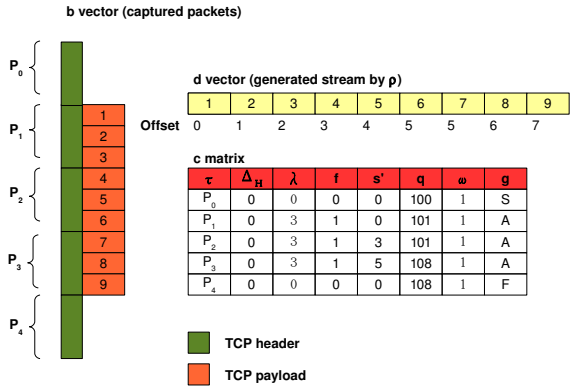

Figure 1. Detecting corrupted streams

In a next step we consider a reassembled stream $\overrightarrow{d}$ as a vector of bytes shown in figure 1. From the vectors $\overrightarrow{b}$ and $\overrightarrow{d}$ we generate a matrix c shown in figure 1 serving to check the reassembly process. The column τ contains the time stamp attached during the capture. The variable Δ_H is the difference of the real packet length and the effective captured packet length. For packets that were completely captured Δ_H is 0. The variable λ quantifies the TCP payload length. The number of occurrences of the packet's payload in the stream is specified by the variable f. The offset in the stream of the payload is described with the s' variable, followed by the q variable which is the used sequence number in the TCP packet. The column $\omega(p_i)$ indicates correct or incorrect checksums. Finally the column g contains the TCP flags present in the TCP header. This matrix is sorted according sequence numbers and a matrix c' matrix is the result. Here it is already worth to mention that in the best case for a reassembly function $\rho(p_i) = f_k^j$, $j = s'$. Two choices are possible if this statement does not hold: (i) the stream was not correctly reassembled or (ii) the payload of the TCP packet p_i was found more than once, i.e. $f > 1$. The computational complexity for establishing the matrix c is $O(n^2)$ because the frequencies of the TCP payloads in the stream is required.

3.4 Reassembly error estimation

After the presentation of the TCP reassembly model and TCP reassembly challenges we are interested in establishing probabilities for quantifying TCP reassembly errors. Two kind of potential errors are presented (i) potential errors at packet level and (ii) potential errors at stream level. Of course for a given reassembly tool it should first be checked if these probabilities should be computed or if they could be neglected due to a correct software design.

3.4.1 Errors at packet level

A potential error at packet level is a probability that a defective packet caused a wrongly reassembled stream. The probability that faulty checksums influence stream reassembly is defined in equation 10.

$$P_c = \frac{|\{p_i \in \hat{\mathcal{P}} \mid \omega(p_i) = 0\}|}{|\hat{\mathcal{P}}|} \tag{10}$$

Neglected IP fragmentation in a reassembly process induces the probability of observing fragmented IP packets during the reassembly process defined in equation 11 where ϕ is a function that returns 1 if a packet is fragmented and 0 otherwise.

$$P_f = \frac{|\{p_i \in \mathcal{P} \mid \phi(p_i) = 1\}|}{|\mathcal{P}|} \tag{11}$$

In order to avoid errors caused by fragmented IP packets or by faulty checksums it must be ensured that the used reassembly function correctly handles such packets.

3.4.2 Errors at stream level

A potential error at stream level is the probability that quantifies how many streams seem to be corrupted.

On the one hand the set E contains TCP sessions and on the other hand we have the set of recovered streams. Ideally the number of reassembled TCP sessions should be the number of recovered streams. The number δ represents the difference of reassembled sessions and recovered streams $\delta = \mid E \mid - \mid S \mid$. If δ is zero, no mismatch was detected.

One the one hand, we call invisible streams, TCP sessions present in the set of captured packets and the probability of having such streams is defined in equation 12. On the other hand we name an additional stream, a stream that was not generated from a session shown in equation 13.

$$\delta > 0 : I_p = 1 - \frac{\mid S \mid}{\mid E \mid} \tag{12}$$

$$\delta < 0 : A_p = 1 - \frac{\mid E \mid}{\mid S \mid} \tag{13}$$

Equation 13 and 12 give an idea about the number of streams that does not match with the number of sessions. However these equations might say that there is no error even if there are errors that compensate each other. An example is an invisible stream that compensates a additional stream. Therefore it is essential to check if the streams are consistent. The probability where multiple sessions are present in a TCP flow is defined in equation 14.

$$P_{spec} = \frac{|\{\sigma(p_i) \mid p_i \in \mathcal{M}_p\}|}{|I|} \tag{14}$$

The methodology proposed in figure 1 establishes a sorted matrix c' which can be used to detect reassembly in some cases. The cases where ambiguities are present can also be detected. Streams can be wrongly reassembled due to incomplete collected payloads or mixed packet payloads which induce a faulty offset computation.

At first and foremost the packet capturing process must capture the complete packets in order to reassemble correctly a stream. The libpcap[1] library, used by the tool `Tcpdump` provides two packet lengths. The $caplen$ is the packet length effectively captured, and len is the length of the initial sent packet length. The parameter $\Delta_H = len - caplen$ can be different than zero which means that the packet was not completely captured. Incomplete packets that are used by reassembly functions cause non-corrective errors due to the lack of captured information, show in equation 15.

$$P_{\Delta_H} = \frac{\mid \{p \in \sigma(p_i) | \Delta_H > 0\} \mid}{\mid P \mid} \tag{15}$$

Incorrect offsets can also be caused by software defects. In a correct reassembled stream no holes should be present which can be formulated if relation 16 holds where $s'_0 = 0$.

$$i > 0 : s'_i = \lambda_{i-1} + s'_{i-1} \tag{16}$$

4 Experimental evaluation

Section 3 shows a TCP reassembly model and TCP reassembly challenges. In this section we describe our practical experiments we did aiming to measure the accuracy of TCP reassembly.

4.1 Methodology

Based on two captures created with the tool `Tcpdump` we evaluated the popular research tools `Tcpflow` and `Tcptrace`. Our purpose is to estimate reassembly errors. The library libcap provides an API[2] to manipulate captured

[1] http://www.tcpdump.org

[2] Application Programming Interface

	hotspot	$proxy_1$
Start date	2006-05-15	2008-03-21
End date	2006-05-17	2008-05-06
Number of packets	3833988	41609423
Number of IP packets	3781761	41609423
Number of TCP packets	1227920	41609423
Number of UDP packets	2534954	0
Number of Non-IP packets	52227	0
pcap file size	1.3GB	38GB
Wrong checksums	5	0

Table 1. Used data sets

network packets. In the following sections we use a custom program that uses the pcap library and reads stored packet capture already in pcap format.

Table 1 shows the data sets we used in our experiments. We used two network captures: With the $proxy_1$ capture we simulated the situation where a given destination IP address and source address is monitored over a large period of time, like it is the case for high-interactive honeypots. We captured the traffic from a web proxy. The $hotspot$ capture contains traffic from an hotspot which was active last year during two days and contains more versatile TCP packets. Inside the $proxy_1$ capture an host running Windows XP SP2 was monitored and the $hotspot$ capture contains packets initiated from different operating systems. On our captures we executed the tool `Tcpflow` and `Tcptrace`. Moreover our custom pcap program extracts all the tuples from a pcap file, according the 5-tuple relation presented in equation 1.

4.1.1 Counting the number of TCP sessions

We also counted the number of sessions inside a pcap file in three steps. At first we generated with a script a list that summarises the TCP packets in the pcap file. From each captured TCP packet we extracted the time stamp, followed by the tuple and TCP flag. In a second step we processed this list and created a list for each tuple. In the algorithm proposed in figure 2 we inspected the TCP flag. If a TCP connection is established with a SYN flag and terminated with a FIN or RST flag we incremented a counter per list. This counter represents the number of sessions.

```
1: end_con ← 0
2: sessions ← 0
3: while (pkt = read_packet()) do
4:     if (IS_FIN_SET(pkt.flag)) then
5:         end_con ← 1
6:     end if
7:     if (IS_RST_SET(pkt.flag)) then
8:         end_con ← 1
9:     end if
10:    if (IS_SYN_SET(pkt.flag)) then
11:        if (sessions = 1) then
12:            sessions ← 1
13:        else if (end_con = 1) then
14:            sessions ← sessions + 1
15:        end if
16:    end if
17: end while
```

Figure 2. Counting session algorithm

4.2 Results

4.2.1 $Proxy_1$ capture analysis

In the proxy capture, 0 IP packets had a bad checksum. Thus $P_c = 0$. Next 0 IP packets were fragmented and $P_f = 0$. Moreover 7942 tuples were extracted directly from the pcap file with a script. The computation of $P_{\Delta H} = 0.57$ provides the proportion of incomplete captured packets, which prevent a correct stream reassembly. The tool `Tcpflow` has processed 7930 flows. We see that 12 flows are missing. This can be explained due to the fact that the tool `Tcpflow` wanted to generate too large files which were refused to be created. In case only a connection establishment is present in a tuple and when no data is exchanged, the tool `Tcpflow` does not create the corresponding stream. The tool `Tcpflow` maps sessions and streams by generating stream files that are identified with a tuple. Thus the tool `Tcpflow` generated 7930 streams which exactly matches with the number of processed tuples. The mismatch between the number of sessions and streams can be detected by computing $I_p = 0.88$. In case multiple sessions per flows are present, which can be measured by calculating P_{spec}, the tool `Tcpflow` mixed the sessions.

The tool `Tcptrace` refused to process a pcap file of 38GB. Therefore we split the 38GB pcap file in small pcap files using `Tcpdump`. Each generated file has a size of 2GB which were processed by the tool `Tcptrace`. The tool `Tcptrace` processed 73937 5-tuples. We see that 5-tuple are missing due to the fact that sessions were interrupted by cutting the file. A second reason for this difference is that some flows only contain TCP packets to establish a TCP connection but no data is exchanged. The computation of $I_p = 0.016$ indicates mismatches of streams and sessions. By comparing I_p for the tools `Tcptrace` and `Tcpflow`, we see that the tool `Tcptrace` extracts more streams than the tool `Tcpflow`. The small deviation of the tool `Tcptrace` has two origins. At first a session that only contains packets of a connection establishment are counted as a session with our algorithm and the tool `Tcptrace`

does not create a session for them. The second reason is that some sessions could not be stored due to the use of invalid file descriptors. For a capture of 38GB recorded during 47 days, we computed $P_{spec} = 0.98$ which means that the flows contain more than one session. The phenomenon of having more than one session per flow emerged after three days and can be explained by the birthday problem, discussed in section 3.3.1. If, in equation 9, n increases, $p(n)$ also increases. In case of a continuous captured traffic n increases.

4.2.2 Hotspot capture analysis

This capture contains more heterogenous TCP sessions due to the fact that we are monitoring a transit network and not a specific resource like in the previous proxy capture. In this capture some packets contain a bad checksum. Thus $P_c = 13 \times 10^{-6}$. In this capture no fragmented IP packets were present and $P_f = 0$. The computation of $P_{\Delta H} = 0.43$ shows that packets were not completely captured. The tools `Tcptrace` and `Tcpflow` have the same number of invisible streams, $I_p = 0.23$. On this capture we also applied the tool Valgrind on the tools `Tcptrace` and `Tcpflow`. For the tool `Tcptrace` we observed that 5 times invalid addresses were read and 36196 invalid file descriptors were used. For the tool `Tcpflow` we noticed that 11 times invalid addresses were read and 4 times uninitialized buffers were used. These numbers motivate further analysis and building a methodology to check if streams were correctly reassembled.

5 Conclusion

In order to use the output of a network forensic tool against some one we believe that it is mandatory to firstly check the captured input data and the capabilities of the used tools. Secondly an output should validated with other independent tools. Two families of errors can emerge. Firstly it might be that the capture tool was not correctly calibrated and some packets are truncated. Next streams might be defectively reassembled due to caused ambiguities that end in implementation flaws. An additional need is that the analysis tools have the same interpretation of flows, sessions and streams. We proposed a TCP reassembly model and a stream verification methodology that can be used to derive and compute reassembly errors. For the future work activities, we firstly planed to continue the evaluation of our stream verification methodology and we are trying to establish a precise classification of TCP reassembly error causes. Based on such classification we can create probabilistic models for estimating TCP reassembly errors. Finally we plan to contribute to existing free and open source TCP reassembly software to correct or improve them.

Acknowledgments

This work is partially funded by U-2010, an integrated research project of the 5th Call of the 6th European Research Frame Program. We also want to thank Dr. Radu State for his useful comments and corrections and the members, of the SECAN LAB for their fruitful discussions.

References

[1] R. Bejtlich. *The Tao Of Network Security Monitoring: Beyond Intrusion Detection*. Addison-Wesley Professional, 2004.

[2] M. Choen. Pyflag - an advanced network forensic framework. *Digital Investigation*, 5(1), August 2008.

[3] B. Claise. Cisco Systems Netflow Services Export System, Oct 2004. RFC 3954.

[4] E. Cronin, M. Sherr, and M. Blaze. On the reliability of current generation network eavesdropping tools. *International Federation for Information Processings*, 222(2):103–113, 2008.

[5] M. Crovella and B. Krishnamurthy. *Internet Measurement*, chapter Issues in capturing data, pages 101–102. John Wiley & Sons Ltd, 2006.

[6] A. Das, N. David, Z. Jospeh, M. Gokhan, and C. Alok. An FPGA-based network intrusion detection architecture. *Information Forensics and Security*, 3(1):118–132, Mar 2008.

[7] S. Dharmapurikar and V. Paxson. Robust tcp stream reassembly in the presence of adversaries. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[8] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[9] E. H. McKinney. Generalized birthday problem. *American Mathematical Monthly*, 73:385–387, 1966.

[10] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[11] L. Schaelicke and J. Freeland. Characterizing sources and remedies for packet loss in network intrusion detection systems. In *Workload Characterization Symposium*, pages 188–196. IEEE, Oct 2005.

[12] D. Song, G. Shaffer, and M. Undy. Nidsbench - a network intrusion detection test suite. In *Recent Advances in Intrusion Detection*, 1999.

[13] J. Stone and C. Partridge. When the crc and tcp checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, 2000.