Additional Technical and Experimental Report for Rosetta

Renjie Xie[1,3,*], Jiahao Cao[1,2,3,*], Enhuan Dong[1,3,5,*], Mingwei Xu[1,2,5], Kun Sun[4]
Qi Li[1], Licheng Shen[2] and Menghao Zhang[2,6]

[1]*Institute for Network Sciences and Cyberspace, Tsinghua University*
[2]*Department of Computer Science and Technology, Tsinghua University*
[3]*Beijing National Research Center for Information Science and Technology, Tsinghua University*
[4]*Department of Information Sciences and Technology, George Mason University*
[5]*Quan Cheng Laboratory,* [6]*Kuaishou Technology*

A Experiments with Different TCP Implementations

We conduct the experiments to demonstrate that Rosetta can generalize across different TCP implementations in different Operating Systems (OSes). We train the models with about 1.8 million website TLS flows generated and collected by the Linux clients in τ_1. We then test the models with about 1.2 million website TLS flows generated and collected by the Windows clients in τ_1 to τ_6.

Results on Classifying Windows Website TLS Flows in Different Network Environments. As shown in Table 1, the six deep learning models achieve low accuracy on classifying Windows website flows in diverse network environments. However, Rosetta effectively improves the classification performance of the models in diverse network environments, which is shown in Table 2. Particularly, there is 13.87% accuracy improvement on average in τ_5. Moreover, Rosetta improves about 27% accuracy at most for a deep learning model on classifying Windows Website TLS flows. It is obvious that Rosetta enables the models to robustly classify Windows TLS flows in diverse network environments though the models are trained with Linux TLS flows.

B Comparisons with the Baseline Website Fingerprinting Classifier

We conduct experiments to compare our approach with the baseline website fingerprinting classifier [10], i.e., the VNG++ classifier. We train the VNG++ classifier to classify TLS flows from CIRA-CIC-DoHBrw-2020 in different networks, respectively. Table 3 shows that the baseline website fingerprinting classifier fails to achieve robust traffic classification. However, as we have shown in Table 6 of Section 5.2, our approach can achieve robust and acceptable classification performance in diverse network environments.

*These authors contributed equally to this work.

C Root Cause Analysis

We explain the root causes of the three phenomenons, i.e., packet subsequence shift, packet subsequence duplication, and packet size variation, from the perspective of the TCP protocol in detail.

Packet Subsequence Shift Analysis. We discover that packet loss and TCP retransmission mechanisms are the two root causes of packet subsequence shift. Packet loss often occurs when packets travel across congested wired or wireless networks. When packets arrive at an outgoing interface of a forwarding device at a higher rate than the capacity of the corresponding link, the forwarding device first builds up a queue at the interface and then begins to drop packets if the queue length is higher than the maximum queue length [2,17]. The packet loss rate in wireless networks is higher than that in wired networks [4,8,27] since the packets can be dropped due to other congestion irrelevant reasons [4,9,13,14,23–25]. As the network only provides best-effort delivery service, TCP is designed to detect packet loss and performs retransmission to ensure reliable transmission [22].

TCP develops retransmission timeout (RTO) and fast retransmit mechanisms to handle packet loss and avoid congestion [6]. RTO mechanism detects packet loss with a timer. After sending a packet, a sender starts a retransmission timer. If it does not receive the corresponding acknowledgment before the timer expires, the sender considers the packet as lost and retransmits the lost packet. Besides using the retransmission timer, packet loss can be detected by the duplicate acknowledgments following the fast retransmit mechanism. For a receiver, after receiving an in-order packet, the receiver updates the acknowledgment number (ACK number) of the next acknowledgment packet (ACK packet). Once the receiver receives an out-of-order packet, it sends an ACK packet immediately to the sender with an unchanged ACK number (the same as the last ACK packet). If multiple out-of-order packets arrive at the receiver after a packet is lost, the receiver will generate the same number of ACK packets with the same ACK number (i.e., duplicate acknowledgments). When a sender

Table 1: Results on Classifying Windows Website TLS Flows in Different Network Environments (Without Rosetta). Here, all the models are trained with Linux Website TLS Flows.

Model	Different Wired Network Environments						Different Wireless Access Network Environments					
	τ_1		τ_2		τ_3		τ_4		τ_5		τ_6	
	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1
CNN	88.25%	87.53%	77.36%	72.64%	77.03%	73.13%	83.58%	82.67%	92.03%	91.85%	78.33%	78.28%
SDAE	74.10%	70.92%	71.39%	68.65%	70.31%	70.31%	82.59%	82.01%	64.14%	63.94%	70.33%	69.82%
LSTM	70.65%	69.06%	60.57%	59.76%	60.69%	61.27%	71.64%	67.47%	62.15%	61.53%	61.10%	58.16%
DF	88.61%	87.77%	78.86%	73.46%	77.87%	76.03%	89.55%	89.28%	92.83%	92.49%	78.74%	78.72%
FS-Net	70.77%	68.79%	71.14%	64.61%	57.70%	53.49%	71.29%	69.22%	65.76%	59.79%	63.51%	59.17%
Transformer	92.54%	91.87%	76.47%	75.46%	58.56%	56.44%	95.02%	93.49%	90.44%	89.54%	86.21%	83.64%
On Average	**80.82%**	**79.32%**	**72.63%**	**69.10%**	**67.03%**	**65.11%**	**82.28%**	**80.69%**	**77.89%**	**76.52%**	**73.04%**	**71.30%**

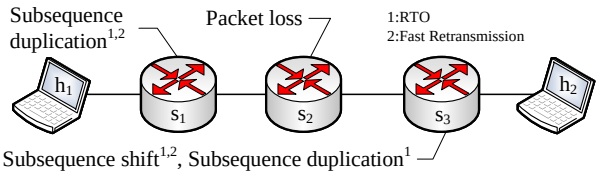
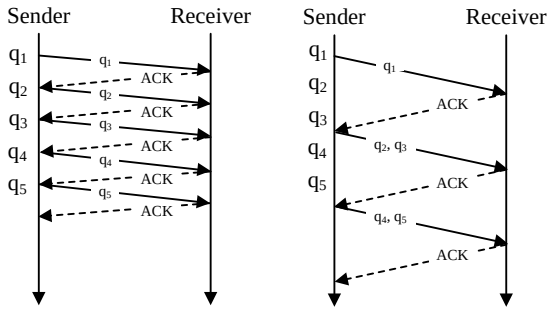


Figure 1: An example of retransmission mechanisms. When packets are lost at s_2, RTO and fast retransmit both lead to packet subsequence duplication of the flows captured at s_1 while both of them may also incur packet subsequence shift of the flows captured at s_3. Moreover, RTO can also lead to packet subsequence duplication of the flows captured at s_3.



(a) Low delay. (b) High delay.

Figure 2: An example of how the Nagle algorithm performs under different delays. Each data block is sent separately by a packet under low delays. However, some data blocks are combined into a packet under high delays.

receives N (e.g., $N = 3$) duplicate ACK packets, the sender is aware of packet loss and retransmits the packet. It is worth to note that the RTO mechanism is usually triggered by severe congestion or a continuous large number of packet loss events, making the sender unable to receive enough duplicate ACK packets to start fast retransmission.

As an example shown in Figure 1, h_1 establishes a TCP connection with h_2 and sends packets to h_2. There exists three switches s_1, s_2, and s_3 between them. When s_2 becomes congested, some packets are dropped. Then TCP RTO and fast retransmit mechanisms lead to packet sequence shift at s_3. Now considering a flow with packet length sequence of $[q_1, q_2, q_3, q_4, q_5]$ with q_2 dropped by s_2. With the help of the fast retransmit mechanism, h_1 could detect q_2 is missing and resend it. Then at s_3, the captured packet length sequence would be $[q_1, q_3, q_4, q_5, q_2]$ and q_2 is shifted to be the last position of the sequence. If continuous packets are dropped at s_2, the RTO mechanism may be triggered. After the RTO timer expires, the sender will retransmit all the packets that have not been cumulatively acknowledged. If q_2, q_3 and q_4 are dropped, the captured packet length sequence at s_3 could be $[q_1, q_5, q_2, q_3, q_4]$. The subsequence of q_2, q_3, q_4 is shifted.

Packet Subsequence Duplication Analysis. The packet loss and TCP retransmission mechanisms are also the main reasons of packet subsequence duplication. We consider the example in Figure 1 again. Due to TCP RTO and fast retransmit mechanisms, the packets lost at s_2 would be resent from h_1. At s_1, the captured packet length sequence would include duplicate packet subsequences. For example, consider a flow with packet length sequence of $[q_1, q_2, q_3, q_4, q_5]$ and assume that the q_2 is dropped. The fast retransmit mechanism can detect q_2 is missing and resend it. The captured packet length sequence from s_1 could be $[q_1, q_2, q_3, q_4, q_5, q_2]$. If severe congestion occurs and continuous packets are dropped at s_2, the RTO mechanism may be triggered. After the RTO timer expires, the sender will retransmit all the unacknowledged packets. If q_2, q_3 and q_4 are dropped, the captured packet length sequence at s_1 could be $[q_1, q_2, q_3, q_4, q_5, q_2, q_3, q_4]$. Moreover, we can see packet subsequence duplication at s_3 in Figure 1. Consider a flow with the packet length sequence of $[q_1, q_2, q_3, q_4, q_5]$ and assume that severe congestion leads to continuous lost packets, i.e., q_2, q_3 and q_4 are dropped. In such a case, the captured packet length sequence at s_3 would be $[q_1, q_5, q_2, q_3, q_4, q_5]$, where the second q_5 is a duplicate subsequence.

Packet Size Variation Analysis. We find packet size variation can be observed in the situation where the delay between TCP endpoints becomes higher or the access mode changes. High end-to-end delays may increase the packet size, since TCP

Table 2: Results on Classifying Windows Website TLS Flows in Different Network Environments (With Rosetta). Here, all the models are trained with Linux Website TLS Flows.

Model	Different Wired Network Environments					
	τ_1		τ_2		τ_3	
	AC	F1	AC	F1	AC	F1
CNN + Rosetta	89.57%(↑1.32%)	88.90%(↑1.37%)	77.61%(↑0.25%)	72.88%(↑0.24%)	81.51%(↑4.48%)	79.53%(↑6.40%)
SDAE + Rosetta	94.24%(↑20.14%)	94.02%(↑23.11%)	79.73%(↑8.34%)	74.35%(↑5.70%)	72.15%(↑1.84%)	71.55%(↑1.24%)
LSTM + Rosetta	87.53%(↑16.88%)	83.49%(↑14.43%)	78.48%(↑17.91%)	73.46%(↑13.70%)	81.93%(↑21.24%)	80.48%(↑19.21%)
DF + Rosetta	90.20%(↑1.59%)	90.00%(↑2.24%)	80.85%(↑1.99%)	75.13%(↑1.67%)	83.47%(↑5.60%)	82.00%(↑5.98%)
FS-Net + Rosetta	88.13%(↑17.36%)	87.22%(↑18.43%)	79.98%(↑8.84%)	74.66%(↑10.05%)	79.27%(↑21.57%)	77.27%(↑23.78%)
Transformer + Rosetta	90.65%(↓1.89%)	90.04%(↓1.83%)	78.23%(↑1.76%)	74.16%(↓1.30%)	82.49%(↑23.93%)	80.78%(↑24.34%)
On Average	**90.05%(↑9.23%)**	**88.95%(↑9.62%)**	**79.15%(↑6.51%)**	**74.11%(↑5.01%)**	**80.14%(↑13.11%)**	**78.60%(↑13.49%)**
Model	**Different Wireless Access Network Environments**					
	τ_4		τ_5		τ_6	
	AC	F1	AC	F1	AC	F1
CNN + Rosetta	91.04%(↑7.46%)	91.25%(↑8.58%)	95.22%(↑3.19%)	95.07%(↑3.22%)	78.62%(↑0.29%)	78.60%(↑0.32%)
SDAE + Rosetta	92.54%(↑9.95%)	92.48%(↑10.470%)	92.03%(↑27.89%)	91.69%(↑27.75%)	78.56%(↑8.23%)	78.56%(↑8.74%)
LSTM + Rosetta	74.63%(↑2.99%)	71.76%(↑4.30%)	81.67%(↑19.52%)	80.18%(↑18.65%)	75.00%(↑13.90%)	74.95%(↑16.79%)
DF + Rosetta	92.01%(↑2.46%)	90.85%(↑1.57%)	94.82%(↑1.99%)	93.56%(↑1.07%)	80.14%(↑1.40%)	80.14%(↑1.42%)
FS-Net + Rosetta	73.13%(↑1.84%)	69.78%(↑0.56%)	92.03%(↑26.27%)	91.79%(↑32.00%)	78.10%(↑14.59%)	77.99%(↑18.82%)
Transformer + Rosetta	90.05%(↓4.97%)	89.81%(↓3.68%)	94.82%(↑4.38%)	94.61%(↑5.07%)	79.50%(↓6.71%)	79.48%(↓4.16%)
On Average	**85.57%(↑3.29%)**	**84.32%(↑3.63%)**	**91.77%(↑13.87%)**	**91.15%(↑14.63%)**	**78.32%(↑5.28%)**	**78.29%(↑6.99%)**

Table 3: Results on TLS Traffic Classification with the Baseline Website Fingerprinting Classifier in Different Network Environments.

Method	Different Wired Network Environments								Different Wireless Access Network Environments						On Average	
	θ_0		θ_1		θ_2		θ_3		θ_4		θ_5		θ_6			
	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1	AC	F1
VNG++ [10]	89.47%	89.44%	88.53%	88.44%	52.73%	34.53%	57.80%	36.63%	86.73%	86.59%	74.00%	71.73%	52.84%	34.57%	71.73%	63.13%

applies the Nagle algorithm [20] to improve transmission efficiency. The Nagle algorithm buffers a number of small outgoing packets within an RTT and sends them into one big packet together. Besides delays, the changed access modes of the Internet also have an impact on MTU and may cause IP fragmentation for packets, as MTUs limit the maximum size of packets.

The Nagle algorithm is designed to handle the small-packet problem, where an application repeatedly sends packets with small payloads. The Nagle algorithm is enabled by default in major TCP implementations [26]. It can effectively reduce the number of packets when applications or web servers generate massive small data segments. Therefore, it effectively saves CPU resources and network bandwidth. Moreover, the Nagle algorithm can be a useful firewall when sloppy applications or complex bugs send too many tiny data packets and cause network congestion [16, 19].

As TCP packets have headers with a length l of at least $40B$, a small d bytes of user data results in considerable overheads. The Nagle algorithm improves transmission efficiency by combing a number of outgoing packets and sending them in one packet, reducing the number of packets that need to be sent and increasing packet sizes. After data segments are generated by applications, the Nagle algorithm buffers unsent segments in the TCP stack. It then merges them into a new packet for delivery until receiving the corresponding acknowledgment. This process buffers the packet within an RTT. The Nagle algorithm has been included in state-of-the-art TCP implementations for many years [19]. Figure 2 demonstrates how the Nagle algorithm works under different delays. The sender delivers five data blocks into the TCP stack in a fixed interval. In Figure 2a, each data block is transmitted by one packet. However, in Figure 2b, some data blocks are combined into one packet due to the Nagle algorithm. Until receiving the ACK packet for q_1, the sender keeps buffering q_2 and q_3 in the TCP stack. Then, q_2 and q_3 are combined and sent together. q_4 and q_5 are also transmitted in one packet. Thus, the packet length sequence of a flow under high delays is different from that under low delays due to the Nagle algorithm.

Besides delays, the maximum transmission unit (MTU) impacts packet length by limiting the maximum packet length and causing IP fragmentation. MTU can be different in vari-

Algorithm 1 Packet Subsequence Duplication Augmentation via RTO

Input: O : *packet length sequence of a flow*;
p : *packet loss rate*;
$[L_{min}, L_{max}]$: *Range of Lost Packets*
Output: M;

1: $T \leftarrow EmptyList$
2: **while** $i <= |O|$ **do**
3: **if** $Random(0,1) < p$ **then** ▷ If packets are lost.
4: $L \leftarrow Random(L_{min}, L_{max})$
5: $T.add(O.Subseq(i, i+L))$
6: $M.add(O.Subseq(i, i+L))$
7: $i \leftarrow i+L$
8: **else**
9: $M.add(O[i])$
10: $M.add(T)$
11: $T \leftarrow EmptyList$
12: $i \leftarrow i+1$
13: **end if**
14: **end while**
15: $M.add(T)$
16: $output(M)$

ous networks, e.g., 1500 bytes in Ethernet v2 [12], 1492 bytes in PPPoE v2 [15], and 2304 bytes in Wi-Fi [1]. Hence, to avoid fragmentation in the IP layer, TCP usually specifies the maximum amount of user data (payload size or packet length without headers) in a packet as MSS and makes it equal to the PMTU [18] minus the IP and TCP header sizes. Therefore, changing access modes of the Internet may make MTU (or MSS) different, leading to packet size variation.

D Details on TCP-Aware Traffic Augmentation Algorithms

Although TCP has different implementations, their core design mostly follows the specification defined in RFC-9293 [11]. Therefore, we provide five TCP-aware traffic augmentation algorithms to cover the major features shared by different TCP implementations. Here, our algorithms do not simulate the expected behavior on the target network where we classify flows. Instead, they generate massive augmented TLS flows by configuring different parameters without the prior knowledge of the target network. These augmented TLS flows are used to train the traffic invariant extractor of Rosetta for robust feature extraction.

Packet Subsequence Duplication Augmentation. Algorithm 1 shows packet subsequence duplication augmentation via the RTO mechanism. For each original TCP flow's packet length sequence (O), the output is a modified TCP flow's packet length sequence (M) that may appear in a network environment with a packet loss rate of p. We introduce a list T to record the length subsequence of lost packets. The elements of T need to be appended to M after the successfully delivered packet (line 10) or at the end of the transport (line 15). Line 1 initializes the list T to an empty list. Lines 2 to 15 simulate the impact of the TCP RTO mechanism. Since the TCP RTO mechanism is triggered by severe congestion (i.e., a continuous large number of lost packets), we consider continuous packet loss (lines 3 to 7). The number of lost packets is randomly chosen (line 4) and the lengths of lost packets are recorded in T (line 5). The lost packets can be presented in M multiple times. Line 6 appends them into M to simulate the lost packets. When these packets are retransmitted, their lengths would be appended into M again (line 10).

Algorithm 2 Packet Subsequence Duplication Augmentation via Fast Retransmit

Input: O : *a flow's packet length sequence*;
p : *packet loss rate*
Output: M;

1: **for** $i = 1 \rightarrow |O|$ **do**
2: $O[i].flag \leftarrow unsent$
3: **end for**
4: **while** $|O| > 0$ **do**
5: **for** $i = 1 \rightarrow |O|$ **do**
6: $M.add(O[i])$
7: **if** $Random(0,1) > p$ *or* $O[i].flag = lost$ **then**
8: $O.pop(i)$ ▷ The packet is successfully delivered.
9: *break*
10: **else**
11: $O[i].flag \leftarrow lost$
12: **end if**
13: **end for**
14: **end while**
15: $output(M)$

Algorithm 2 shows packet subsequence duplication augmentation via the fast retransmit mechanism. For each original TCP flow's packet length sequence (O), the output is a modified TCP flow's packet length sequence (M) that may appear in a network environment with a packet loss rate of p. For each element of O, the algorithm maintains a flag to indicate whether the corresponding packet is unsent or lost. Lines 1 to 3 initialize the flag of each element of O to "unsent". Lines 4 to 14 consider the impact of the fast retransmit mechanism. For each delivered packet, it will be removed from O (Line 8), so at the end of the algorithm, the element number of O will be 0. Since the number of duplicate ACK packets that triggers the fast retransmit mechanism is different for different implementations of TCP [3, 5, 7], we assume that number is 1. It means after an out-of-order or in-order packet is successfully delivered to the receiver (Lines 7 to 9), the receiver will send back a duplicate or normal ACK packet, then the sender will retransmit the lost packet or send a new packet by breaking

Algorithm 3 Packet Subsequence Shift Augmentation via RTO

Input: O : *a flow's packet length sequence*;
p : *packet loss rate*;
$[L_{min}, L_{max}]$: *Range of Lost Packets*
Output: M;

```
1:  T ← EmptyList
2:  while i <= |O| do
3:      if Random(0,1) < p then                ▷ If packets are lost.
4:          L ← Random(L_min, L_max)
5:          T.add(O.Subseq(i, i+L))
6:          i ← i+L
7:      else
8:          M.add(O[i])
9:          M.add(T)
10:         T ← EmptyList
11:         i ← i+1
12:     end if
13: end while
14: M.add(T)
15: output(M)
```

Algorithm 4 Packet Subsequence Shift Augmentation via Fast Retransmit

Input: O : *a flow's packet length sequence*;
p : *packet loss rate*
Output: M;

```
1:  for i = 1 → |O| do
2:      O[i].flag ← unsent
3:  end for
4:  while |O| > 0 do
5:      for i = 1 → |O| do
6:          if Random(0,1) > p or O[i].flag = lost then
7:              M.add(O[i])      ▷ The packet is successfully delivered.
8:              O.pop(i)
9:              break
10:         else
11:             O[i].flag ← lost
12:         end if
13:     end for
14: end while
15: output(M)
```

the for-loop at Line 9. For standard TCP, if a retransmitted packet is lost again, the fast retransmit mechanism cannot detect such loss again, and only the RTO mechanism can detect retransmitted packet loss. Therefore, in Algorithm 2, we assume the retransmitted packets will not lose again, which means if $O[i].flag = lost$ (Line 7), then the i-th packet will be successfully delivered (Line 8). These assumptions reduce the complexity and runtime of Algorithm 2.

We should that our augmentation algorithms simplify the RTO and the fast retransmit mechanisms. As there are various specific implementations for the two mechanisms in practice, designing augmentation algorithms to cover all the implementation details of the mechanisms is challenging. We thus focus on the core design of the two mechanisms, and simulate the main effects of the RTO and fast retransmit mechanisms on packets to augment flows. For simplicity, we set the number of duplicate ACK packets as 1 and we randomly drop packets in our algorithms. Besides, we control consecutive packets that are lost in our algorithms to simulate the RTO mechanism.

Packet Subsequence Shift Augmentation. Algorithm 3 shows packet subsequence shift augmentation via the RTO mechanism. Algorithm 1 is similar to Algorithm 3 except that Algorithm 3 considers the capture operation after the packet loss and Algorithm 1 considers the capture operation before the packet loss. Therefore, we remove the code of $M.add(O.Subseq(i, i+L))$. Now, the length sequence of the first sent packets would not be counted in M.

Algorithm 4 shows packet subsequence shift augmentation via the fast retransmit mechanism. Since Algorithm 4 is similar to Algorithm 2 except that Algorithm 4 considers the capture operation after the packet loss and Algorithm 2 considers the capture operation before the packet loss. Therefore, we modify the place of the code of $M.add(O[i])$. Now, the packet length $O[i]$ is added to the output after we know the corresponding packet can be delivered.

Packet Size Variation Augmentation. Algorithm 5 shows packet size variation augmentation with an MSS and an RTT distribution. Here, MSS is the MSS of the simulated network, RTT_{min} is the minimum RTT and RTT_{max} is the maximum RTT. MSS, RTT_{min} and RTT_{max} are configurable parameters. By changing the values of the two parameters, the algorithm can generate massive augmented TLS flows in various simulated network environments to train the traffic invariant extractor of Rosetta. Line 1 extracts Δ, the distribution of packet intervals that can be extracted from offline traffic traces. Note that the distribution of packet intervals of flows can be extracted from any offline traffic dataset. The algorithm simply uses the packet interval distribution to generate different augmented flows with varying packet sizes, which further enables Rosetta to learn TCP semantics for robust traffic classification with the augmented flows. Line 2 initializes i as the index of O. Line 4 generates a RTT based on RTT_{min} and RTT_{max}. As RTTs are random in real network environments [21], we use random RTTs for each packet transmission to simulate different network environments. Lines 5 to 11 simulate the behavior of the Nagle algorithm, i.e., combining all the packets within an RTT into one packet. If the packet is larger than MSS, it will be divided into multiple packets with a maximum size of MSS, as shown in Lines 12 to 16.

Note that our algorithms take MSS, packet loss, burst-loss length and the range of RTT as input parameters. By changing the input parameters as the continuous values each time we

Algorithm 5 Packet Size Variation Augmentation

Input: O : *a flow's packet length sequence*;
$[RTT_{min}, RTT_{max}]$: *RTT Range*;
MSS : *Maximum Segment Size*;
Output: M;

$\Delta \leftarrow from(Traces)$
$i \leftarrow 1$
while $|O| \geq i$ **do**
  $RTT \leftarrow random(RTT_{min}, RTT_{max})$
  $buf \leftarrow 0$
  while $|O| \geq i$ *and* $RTT > 0$ **do**
    $Interval \leftarrow \Delta.get()$
    $RTT \leftarrow RTT - Interval$
    $buf \leftarrow buf + (O[i] - header\_len)$
    $i \leftarrow i + 1$
  end while
  while $buf > MSS$ **do**
    $buf \leftarrow buf - MSS$
    $M.add(MSS + header\_len)$
  end while
  $M.add(buf + header\_len)$
end while
$output(M)$

run the algorithm, we actually vary these parameters over plausible ranges. Such a design also allows us to flexibly give arbitrary values of the parameters when we repeatedly run our algorithms to augment flows. By changing the values of the inputs each time we augment one flow, we can generate massive and diverse augmented flows to train TIE in Rosetta for robust feature extraction.

References

[1] Maximum transmission unit. https://en.wikipedia.org/wiki/Maximum_transmission_unit, March 2022.

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[3] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. RFC 5827: Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP) , 2010.

[4] Sanjit Biswas, John Bicket, Edmund Wong, Raluca Musaloiu-e, Apurv Bhartia, and Dan Aguayo. Large-scale Measurements of Wireless Network Behavior. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 153–165, 2015.

[5] E. Blanton, M. Allman, K. Fall, and L. Wang. RFC 3517: A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP , 2003.

[6] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. RFC 5681: TCP Congestion Control, 2009.

[7] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Larry Luoand Yongqiang Xiong, Xiaoliang Wang, et al. Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers. In *USENIX ATC*, pages 29–42, 2016.

[8] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. A Measurement-based Study of Multipath TCP Performance over Wireless Networks. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 455–468, 2013.

[9] D. N. Cottingham, I. J. Wassell, and R. K. Harle. Performance of IEEE 802.11a in Vehicular Contexts. In *IEEE VTC*, pages 854–858, 2007.

[10] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE symposium on security and privacy*, pages 332–346. IEEE, 2012.

[11] W Eddy. Rfc 9293 transmission control protocol (tcp). 2022.

[12] Charles Hornig. A standard for the transmission of ip datagrams over ethernet networks. Technical report, 1984.

[13] L. Li, K. Xu, D. Wang, C. Peng, Q. Xiao, and R. Mijumbi. A Measurement Study on the TCP Behaviors in HSPA+ Networks on High-speed Rails. In *IEEE Infocom*, pages 2731–2739, 2015.

[14] T. Ma, Y. Lee, S. Winkler, and M. Ma. QoS Provisioning by Power Control for Video Communication via Satellite Links. *International Journal of Satellite Communications and Networking*, 33(3):259–275, 2014.

[15] Louis Mamakos, Kurt Lidl, Jeff Evarts, David Carrel, Dan Simone, and Ross Wheeler. A method for transmitting ppp over ethernet (pppoe). *RFC2516, Feb*, 1999.

[16] Greg Minshall, Yasushi Saito, Jeffrey C Mogul, and Ben Verghese. Application performance pitfalls and tcp's nagle algorithm. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):36–44, 2000.

[17] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin

Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM*, 2015.

[18] Jeffrey Mogul, Steve Deering, et al. Path mtu discovery, 1990.

[19] Jeffrey C Mogul and Greg Minshall. Rethinking the tcp nagle algorithm. *ACM SIGCOMM Computer Communication Review*, 31(1):6–20, 2001.

[20] John Nagle. Rfc0896: Congestion control in ip/tcp internetworks, 1984.

[21] Zhong-Hua Pang, Guo-Ping Liu, Donghua Zhou, and Dehui Sun. Data-based predictive control for networked nonlinear systems with network-induced delay and packet dropout. *IEEE Transactions on Industrial Electronics*, 63(2):1249–1257, 2015.

[22] Jon Postel. RFC 793: TCP: Transmission Control Protocol, 1981.

[23] David C Salyers, Aaron D Striegel, and Christian Poellabauer. Wireless reliability: Rethinking 802.11 packet loss. In *2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–4. IEEE, 2008.

[24] C. Steger, P. Radosavljevic, and J. P. Frantz. Performance of IEEE 802.11b Wireless LAN in an Emulated Mobile Channel. In *Semiannual Vehicular Technology Conference*, volume 2, pages 1479–1483, 2003.

[25] F. Tso, J. Teng, W. Jia, and D. Xuan. Mobility: a Double-edged Sword for HSPA Networks: a Large-scale Test on Hong Kong Mobile HSPA Networks. In *ACM Mobihoc*, pages 81–90, 2010.

[26] Wikipedia. Nagle's algorithm. https://en.wikipedia.org/wiki/Nagle's_algorithm, Octorber 2022.

[27] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 509–522, 2015.