In Proc. IEEE International Conference on Communications, Paris, May 2017

DSSR: Balancing Semantics and Speed Requirements in Packet Trace Replay

Scott Fortner
United States Marine Corps
scott.fortner@usmc.mil

Geoffrey G. Xie
Department of Computer Science
Naval Postgraduate School
xie@nps.edu

***Abstract*—As new network services and middleboxes proliferate, it is important to have reliable means to test these services and devices, and a common practice to generate realistic testing traffic is through replaying previously recorded packet traces. However, existing trace replay tools are highly specialized for particular protocols and scenarios. In this paper, we present a general trace replay tool called DSSR. We show that in a directional trace replay, where outbound and inbound packets are simulated by two (logically) distinct sets of hosts, the timestamps of packets handled by the host(s) simulating external destinations must be adjusted to accurately re-create the effect of network latency. Interestingly, the timestamp adjustment can boost the replay speed. Moreover, we identify the range of timestamp adjustment that will guarantee to preserve the semantic orderings of packets pertaining to client-server protocol interactions. Therefore, our solution provides an effective tuning knob for a user to balance the speed and semantics requirements in a trace replay. Equally important, it requires no clock synchronization between the replaying hosts, as the hosts leverage the arrivals of incoming packets as a clocking mechanism for generating outgoing packets.**

I. Introduction

Security and new application requirements result in a proliferation of middleboxes in enterprise networks [8]. These middleboxes such as Intrusion Prevention Systems (IPSes), voice over IP (VoIP) gateways, and wide area network (WAN) optimizers can impact the security posture of a network and the performance of user traffic in profound ways [5], and the impact is often dependent on the traffic pattern. Therefore, it is important that enterprises have the ability to test these devices with realistic traffic patterns before selecting vendors and making deployment decisions.

Often, it is infeasible to perform such testing in live networks and enterprises must set up separate testing environments. The need for emulating operational networks also arises in many training scenarios. In both cases, it is critical to create traffic patterns representative of the production traffic. One common practice to meet this goal is through replaying packet traces previously recorded using tools such as *tcpdump* or *wireshark*.

While replaying a packet trace is conceptually simple and has the advantage of replicating production traffic at the relatively fine data granularity of packets, it also faces several unique challenges. First, a trace is typically collected at a single network interface, as illustrated in the top part of Figure 1. Simply injecting these packets in one batch in order of their timestamps may not serve the intended purpose. For example, to test middleboxes requires the trace to be divided into two parts based on the direction of the packets, either *inbound* or *outbound*, with respect to the collection point. The inbound packets and outbound packets should be played back by two (logically) distinct sets of hosts and fed to a middlebox through two separate interfaces or connections, as illustrated in the bottom part of Figure 1. In a *directional* playback like this, where packets are injected by multiple independent processes, the possibility of packets to be replayed out of order arises. It is important to bound such timing errors in order to preserve the semantic ordering of packets pertaining to protocol interactions. Second, some traces are collected at links of very high speed, at 100 Mbps range or above. It is nontrivial to scale a replay tool to such link speeds while still adequately bounding the timing errors.

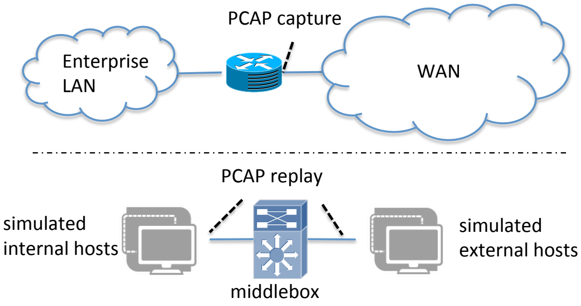


Fig. 1. Typical *directional* packet trace replay scenario

The industry has resorted to hardware assisted solutions, e.g., protocol analyzers, to overcome these challenges. It is likely that some middlebox vendors have also developed proprietary trace replay solutions to meet their testing needs. In this paper, we focus on *solutions based exclusively on software*, which are relatively in-expensive and more accessible compared to hardware assisted solutions. Specifically, we seek to understand the limits of designs that don't require stringent clock synchronization in order to scale to high link speeds. Clock synchronization at the microsecond range[1] is still unreliable with most commodity operating systems.

Prior attempts at software based solutions are highly specialized for particular protocols and scenarios. A detailed

[1] One might try to circumvent the clock synchronization requirement by collocating all replaying processes in one physical machine equipped with multiple network interfaces. However, CPU scheduling at microsecond granularity is equally daunting for commodity OSes.

summary of their features are provided in the next section. This paper represents a first comprehensive investigation of the trade-off between the semantics (i.e. preservation of packet orderings) and speed requirements in trace replays. The insights from the investigation have allowed us to create a new Directional, Semantics-preserving, Speed-adaptive trace Replay (DSSR) tool.

More specifically, we first show that in a directional trace replay, the timestamps for packets handled by the host(s) simulating external destinations must be adjusted to account for network latency of the testing environment. Interestingly, the timestamp adjustment may be leveraged to boost the speed of replay traffic generation. We then formally prove that there is a range of timestamp adjustment that will guarantee to preserve the orderings of packets pertaining to client-server protocol interactions. Leveraging these insights, DSSR provides an effective tuning knob for a user to balance the speed and semantics requirements in a trace replay.

The remainder of this paper is organized as follows: We first review some previously created replay tools and compare them to our tool (Section II). In Section III we describe the methods used to create DSSR, followed with some of the specific implementation details in Section IV. Section V presents an evaluation of both the timing and ordering accuracy of the tool. The paper then concludes in Section VI with a brief discussion of future work.

II. Related Work

Device vendors for testing purposes need to invest in research and tool development for network traffic replay. Unfortunately, the results from these efforts are not publicly available. In the open literature, we identify two efforts that we believe represent the state of the art.

Developed in 2003 and targeting commodity hardware and OSes (e.g., Linux), TCPivo [6] replays traffic directly from a PCAP file at one host. It relies on pre-fetching techniques to maintain timing accuracy for high speed traces, which requires nontrivial kernel modifications. For scenarios where the packet payload information can be ignored, TCPivo also provides an option to substitute actual payload data with null padding in order to increase the speed at which the packets can be replayed.

With a primary focus on intrusion detection and network security in general, TCPOpera [7] aims to re-create the sequence of TCP and session level events (e.g., TCP handshake and browsing of a web object) from a packet trace. In other words, it is essentially an event replay tool. As such, it is more concerned with replicating the distribution of events, than maintaining the timing accuracy at the packet level. Specifically, TCPOpera first develops analytics from a packet trace, then creates a statistical model of the identified events, and finally generates synthetic traffic flows from the model. It uses a different thread for managing each flow, without special consideration to inter-flow sequencing, which can be important for some replay scenarios. It may utilize multiple nodes (each with multiple threads) to mimic a large collection of clients interacting with one or more real servers. TCPopera uses a single control node and control messages passed out-of-band between the nodes to facilitate generation of concurrent flows that collectively exhibit certain statistical properties at the event level.

Table I provides a succinct comparison of our solution (DSSR) with TCPivo and TCPopera. By supporting all key functionality while requiring no special hardware or kernel modification, DSSR is a general tool that can be used in a wide range of testing and training scenarios.

TABLE I. Summary of comparison of solutions

	TCPivo	**TCPopera**	**DSSR**
Directional	No	Yes	Yes
Semantic ordering	Yes	Some events	Yes
Speed adaptation	Yes	No	Yes
OS modifications	Yes	No	No

III. Methodology

To support directional replay, DSSR runs two processes[2] at two different hosts. One host is designated as the “internal” node, emulating the vantage point (commonly a gateway router of a network) at which the original packet trace was collected, and the other as the “external” node, emulating all external destinations within the packet trace. Packets of each trace are partitioned into two groups, “outbound” or “inbound”, from the perspective of the internal node, which are played back at the internal and external nodes, respectively.

Not all orderings of packets have the same importance during a replay. DSSR is designed to preserve the ordering of two packets that have a semantic relationship, which exists when the two packets *are within the same flow and have a causal relationship*. For example a TCP handshake requires that the SYN packet comes before the SYN-ACK, followed by the corresponding ACK. Additionally, a DNS request message over UDP must appear before the response message for this request. Maintaining such orderings of packets is vital for many network testing scenarios, e.g. those involving intrusion detection or WAN optimizer devices, which require fine-grain examination of protocol semantics. In contrast to these examples, packets from different flows[3], or packets transmitted back to back from one TCP send window, maintains no causal relationship up to the transport layer and thus, no semantic relationship.

For this work, the internal node is maintained as the vantage point for modeling whether DSSR preserves semantic orderings. Formally, we say that a replay preserves semantic orderings of the original trace if for any two packets P and Q that have a semantic relationship, the time ordering between P and Q at the internal node during the replay is *the same as that in the original trace*.

A. Dealing with Clock Synchronization

Since DSSR uses two separate processes to play back packets, the issue of time synchronization arises. This is especially the case if each replaying node injects packets independently

[2] It is straightforward to extend our tool to employ more processes, each of which replays a distinct set of flows.

[3] While two flows may have timing dependency, generally such a dependency is less stringent than what is considered in this paper.

entirely based on their relative timestamps. While conceptually simple, this approach requires time synchronization at microsecond resolution between the two nodes for replaying high speed traces. To not rely on special hardware or kernel modification, which is typically required of clock synchronization at microsecond resolution, we explore an alternative self-clocking mechanism as follows. Each node maintains a configuration containing the original timestamps of all the packets, and leveraging the arrivals of incoming packets as a clocking mechanism for generating outgoing packets. To illustrate, suppose one flow in the original trace contains a subsequence of these five packets, P_1 (outbound), P_2 (inbound), P_3 (inbound), P_4 (outbound), and P_5 (inbound), as shown in Figure 2. The internal node will start the sequence with transmitting P_1 and then wait until receiving P_3 before sending P_4, while the external node starts with transmitting P_2 and P_3 and then waits until receiving P_4 before sending P_5.

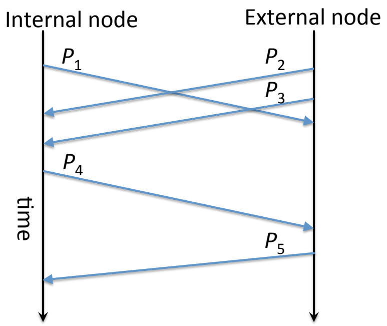


Fig. 2. A simple scenario with 2 outbound and 3 inbound packets

In general, either node can send a packet (except the first one on its side) *only after* receiving the last preceding packet from the other side. Alternatively, the internal node can inject an artifical "bootstrap" packet to initiate the packet injection process of the external node. The pseudo code for the replay logic at the internal node is shown in Algorithm 1 below. The external node uses the same logic except it expects to receive "Outbound" packets. You can see that once the conditions for sending a packet have been met, the node schedules the packet to be sent according to the configuration timestamp, so as to not get ahead of the original trace. In order to accomplish this, both nodes need to know the relative start time of the replay. For the internal node, the start time is user-defined and for the external node, it is triggered by the arrival of the first packet sent from the internal node.

Algorithm 1 Replay Logic at internal node

```
for each packet $p \in$ trace do
    if $p.direction ==$ Inbound then
        while $packetRecvQueue$ is empty do
            $wait(\,)$
        Remove one packet from $packetRecvQueue$
    else
        Send $p$ at its timestamp
```

Clearly, this algorithm may experience a deadlock or speed degradation if some packets are lost or severely delayed by some middleboxes (e.g, an IDS). If this is of concern, short control messages emulating packet counters should be exchanged out-of-band (with more predictable delays) to implement the clocking mechanism.

B. Dealing with Network Latency

Additionally, we find that using two replay nodes introduces timing inaccuracies due to network latency between the nodes. Consider again the above example packet sequence $P1 - P5$. Suppose the average one-way network latency between the nodes (d) is 200 μs. Then the configuration at the external node should be adjusted as illustrated by Figure 3: The timestamps of P_2, P_3, and P_5 should be subtracted by 200 μs, while the timestamps of P_1 and P_4 should be added by 200 μs. Otherwise, the ordering of packets at the external node will be incorrect, e.g., based on the original PCAP timestamps (t_1-t_3), sending of P_2 and P_3 will be incorrectly dependent on receipt of P_1. Consequently, Algorithm 1 will experience undesirable slowdowns, e.g., sending of P_4 will be delayed by 200 μs or more compared to the original trace because P_3 will be delayed by extra 200 μs or more. And we have observed that such errors can have a "snowball" effect due to the self-clocking mechanism.

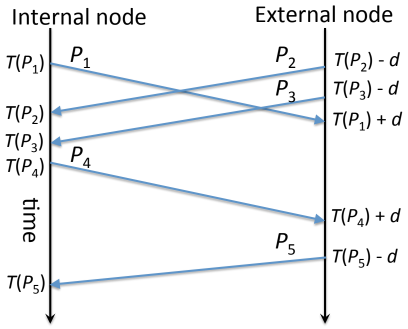


Fig. 3. Illustration of need for timestamp adjustment at the external node. Timestamps of inbound and outbound packets need be decreased or increased by d, respectively.

Therefore, DSSR requires the external node to adjust the timestamp of each packet in its configuration by the expected one-way delay (d). This one-way delay is added to timestamps of outbound packets (to be received) and subtracted from timestamps of inbound packets (to be sent). Once all timestamps have been adjusted, the list of packets are reordered according to the new times. The pseudo code for these adjustments is shown in Algorithm 2.

Algorithm 2 Timestamp Adjustment at external node

```
for each packet $p \in$ trace do
    if $p.direction ==$ Outbound then
        $p.timestamp \leftarrow p.timestamp + d$
    else
        $p.timestamp \leftarrow p.timestamp - d$
Reorder packets based on new timestamps
```

Clearly, the network latency will vary from scenario to scenario and should be estimated *a priori* by measuring the testing network. More importantly, we observe that there is some flexibility to the timestamp adjustment without adversely changing the ordering of packets in a replay. For example, while ignoring network delays (i.e. setting $d = 0$) negatively impacts the replay speed, it does not change packet orderings. Ordering can be maintained trivially by placing a strict ordering constraint for transmitting outbound packets at the internal node. We formally prove in Section V-A that there exists a range of d values to preserve semantic ordering of packets.

IV. Implementation

We have created a prototype implementation of DSSR using C++, leveraging several BOOST [1] libraries, and the libcrafter packet generation and sniffing library [3]. A MySQL database structure is used to initially store and organize all of the packet timestamps and header information for use in constructing the necessary configurations at the replay nodes. The use of a database allows for expansion of the tool through manipulation of the original trace data, such as creation of a statistical replay model, or removal of specific types of packets (e.g., retransmissions).

V. Evaluation

We have two goals in evaluating the performance of DSSR. First, we show a surprisingly strong result that DSSR preserves the orderings of any two packets at the internal node as long as the network latency adjustment (d) meets certain conditions. Second, we explore how the choice of d impacts the speed performance. Combining the two results, we can conclude that DSSR provides an effective tuning knob for a user to balance the speed and semantics requirements in a trace replay. Furthermore, we show how the use of a real-time OS may increase the performance of DSSR.

Unfortunately, we were not able to obtain an implementation of TCPopera [7], the only directional replay tool presented in the open literature, to conduct a detailed performance comparison beyond the qualitative analysis in Section II.

A. Preservation of Semantic Orderings

The following theorem establishes that there exists a range of timestamp adjustment d that will guarantee to preserve the original orderings of packets, *irrespective if they are semantically related*, in a trace replay with DSSR.

Theorem 1 *Assuming no packet losses and negligible processing delays at the replay nodes, DSSR will maintain the ordering of any two packets P and Q in the original capture if $d < |T(P) - T(Q)| + D$, where D is the actual one-way network latency between the replay nodes.*

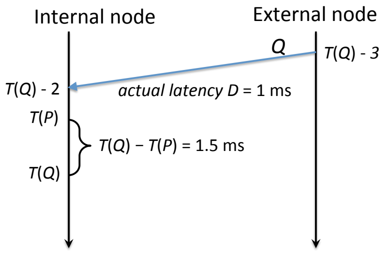


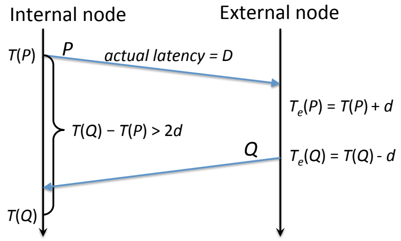
Fig. 4. Example scenario where setting d too much larger than the actual one-way network latency (3 ms vs. 1 ms) results in a packet misordering.

TABLE II. Notation used in proof

Expression	Definition
$T()$	Original packet timestamp
t_i	= $T(Q) - T(P)$
$T_e()$	Adjusted timestamp at external node
t_e	= $T_e(Q) - T_e(P)$
$P.dir$	P's direction (Inbound or Outbound)
D	Actual one-way network latency
$T'_i()$	Actual departure or arrival time at internal node
$T'_e()$	Actual departure or arrival time at external node

1) Proof of Theorem 1: Before presenting the detailed proof steps, we would like to consider Figure 4 and provide some intuition why setting d too large may lead to semantic ordering violations. Suppose packets P and Q are semantically related (i.e., P causing Q) in the original trace and $T(P) - T(Q) = 1.5$ ms. Suppose d is set to 3 ms while the actual network one-way latency between the replay nodes D is 1 ms. In this case, since the external node will adjust P's timestamp to $(T(Q)-3)$ ms, Q will arrive 0.5 ms ahead of P in the replay, resulting in a packet misordering. Now if d is reduced to 2 ms, d would be less than $|T(P) - T(Q)| + D = 1.5 + 1 = 2.5$ ms, and it is straightforward to verify that Q no longer will arrive ahead of P.

The proof is as follows. Consider any two packets P and Q in the original trace. Without loss of generality we assume $T(Q) > T(P)$. Our proof considers five different cases. They collectively cover all combinations of packet directions and the timing difference between P and Q in the original trace. Table II defines the variables used throughout the proof.



$T_e(P) = T(P) + d$

$T_e(Q) = T(Q) - d$

Fig. 5. Case 1 illustration

a) Case 1: $t_i > 2d \wedge P.dir = Outbound \wedge Q.dir = Inbound$ *(See Figure 5):*

1) $T_e(Q) = T(Q) - d$ **(Algorithm 2)**
2) $T_e(P) = T(P) + d$ **(Algorithm 2)**
3) $T(Q) - T(P) > 2d$ **(Case 1 assumption)**
4) $(T_e(Q) + d) - (T_e(P) - d) > 2d$ **(from steps 1-3)**
5) $T_e(Q) > T_e(P)$ **(Simplification)**
6) $T'_e(P) = T'_i(P) + D$ **(network latency)**
7) $T'_i(Q) = T'_e(Q) + D$ **(network latency)**
8) $T'_e(Q) > T'_e(P)$ **(Algorithm 1 with condition of step 5)**
9) $(T'_i(Q) - d) > (T'_i(P) + D)$ **(from steps 7, 8, and 9)**
10) $T'_i(Q) > T'_i(P) + 2d$ **(simplification)**
11) So, $T'_i(Q) > T'_i(P)$ **(simplification)**
12) Therefore, $\forall D, T'_i(Q) > T'_i(P)$

b) Case 2: $ti \leq 2d \wedge P.dir = Outbound \wedge Q.dir = Inbound$ *(See Figure 6):*

1) $T_e(Q) = T(Q) - d$ **(Algorithm 2)**

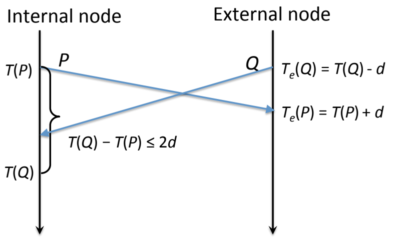


Fig. 6. Case 2 illustration

2) $T_e'(Q) \geq T_e(Q)$ **(Algorithm 1)**
3) $T_i'(Q) = T_e'(Q) + D$ **(by definition)**
4) $T_i'(Q) \geq Te(Q) + D$ **(substitution: 2, 3)**
5) $T_i'(Q) \geq T_i(Q) - d + D$ **(substitution: 1, 4)**
6) $T(Q) = T(P) + t_i$ **(by definition)**
7) $T_i'(P) \geq T(P)$ **(Algorithm 1)**
8) $T_i'(Q) \geq T_i'(P) + t_i + D - d$ **(substitution: 5, 6, 7)**
9) $T_i'(Q) - T_i'(P) \geq t_i + D - d$ **(simplification)**
10) Therefore, when $d < t_i + D$, we have
11) $T_i'(Q) - T_i'(P) > 0 \Rightarrow T_i'(Q) > T_i'(P)$

For brevity, we simply list the rest of the cases and omit the proof steps. Q.E.D.

- Case 3: $P.dir = Inbound \land Q.dir = Outbound$
- Case 4: $P.dir = Inbound \land Q.dir = Inbound$
- Case 5: $P.dir = Outbound \land Q.dir = Outbound$

Corollary 1 *Assuming no packet losses and negligible processing delays at the replay nodes, there exists a range of d values such that DSSR will preserve semantic orderings of packets in a trace replay.*

The proof is straightforward as $d = 0$ meets the condition prescribed in Theorem 1. The following section will show how adjustments to d upwards from 0 can positively effect the speed performance of the replay.

B. Speed Adaptation

While retaining the intra-node ordering constraints, DSSR supports artificially inflating the estimated delay d to boost the replay speed. Inflating this delay essentially causes the configuration at the external node to front-load the send packets. This, in turn, reduces the amount of time the internal node has to wait on packets to be received.

Another solution for improving speed performance involved disregarding packet sequencing almost entirely. By removing the ordering constraints between nodes and sending packets based entirely on their configuration timestamp (ignoring packets received), we were able to much more closely match the speed of the original trace. Upon examination, though, this solution revealed so many ordering discrepancies that we chose not to investigate it further.

1) **Performance Metrics:** We measure the time duration of each replay (in seconds) and derive the *replay speed* in packets per second (pps) from this period and the total number of packets replayed. We model packet ordering at two granularities: (i) the *semantic match rate* identifies the percentage of packets with a semantic relationship that retain their relative orderings during replay. We primarily detect semantic ordering violations in the replay trace using Wireshark and additionally, check if the original timestamp gap is sufficiently large to warrant a causal relationship; (ii) the *direct match rate* identifies the percentage of *all* packets in the replay trace that match the ordering of the original trace and is included for completeness.

2) **Data Sets:** Five diverse test cases are used to validate DSSR: (1) “Home”: consisting of HTTP-centric traffic over a cable modem connection; (2) “Satellite”: provided by a satellite network operator; (3) “Enterprise”: captured on an interface of the Naval Postgraduate School backbone network; (4) “Malware”: a malware traffic trace recorded by a honeypot named HONEYBOT during the Capture the Hacker 2013 competition [2]; and (5) “Ultra High Speed”: an ultra-high speed trace from a busy private network’s access point to the Internet, provided by the author of the *tcpreplay* tool [4]. (Details of these traces are provided in the “Original” column of Tables III-VI.) The first four data sets are used to show the speed and ordering performance of DSSR while the ultra high-speed trace is used to show performance differences from adjusting scheduling priority.

3) **Testbed:** Two PCs, one with an Intel i7-2.4GHz CPU and 8 GB of RAM, the other with an Intel i5-2GHz CPU and 8 GB of RAM, play the role of the internal and external node, respectively. They are wired directly via an Ethernet cable with a verified Gigabit connection. Both machines are natively running Ubuntu 14.04 LTS with most non-essential processes stopped. Before each replay run, the original trace is preprocessed for batch packet pre-construction and external-side timestamp adjustments.

4) **Results:** The results reported are averages over 10 runs. Again, we use D to denote the measured average one-way network latency in our replay testbed.

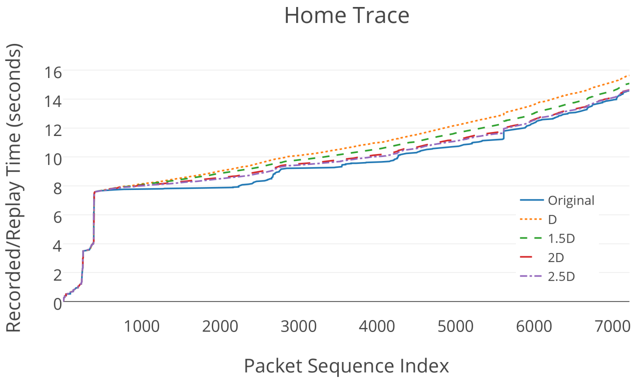


Fig. 7. Effect of d on replaying “Home” trace

Speed vs ordering trade-off: To demonstrate the speed/ordering trade-off, we first plot the average injection time of each packet of the “Home Trace” at various d values against the original trace (Figure 7). These results show the performance limitations of DSSR as well as the effects of adjusting d. As d is increased, the average speed performance also increases. This increase is not linear, nor can it eliminate speed issues all together. Observe that once d is increased beyond 2.5×D, the performance improvements taper off drastically. Additionally, while in some periods the speed of DSSR is insufficient and the replay falls behind the original, DSSR is able to catch back up as the traffic rate in the trace subsides.

TABLE III. RESULTS FOR "HOME" TRACE

	Original	D	1.5×D	2×D	2.5×D
# Packets	7218	7218	7218	7218	7218
# TCP Flows	265	265	265	265	265
# UDP Flows	351	351	351	351	351
Link Speed	105Mbps	1 Gb/s	1 Gb/s	1 Gb/s	1 Gb/s
Duration (s)	14.58	15.64	15.08	14.65	14.62
Speed (pps)	495.1	461.5	478.7	492.7	493.7
% S. Match	-	100	100	100	100
% D. Match	-	84.16	83.49	79.06	79.00

TABLE IV. RESULTS FOR "SATELLITE" TRACE

	Original	D	1.5×D	2×D	2.5×D
# Packets	8882	8882	8882	8882	8882
# TCP Flows	45	45	45	45	45
# UDP Flows	5	5	5	5	5
Link Speed	Unknown	1 Gb/s	1 Gb/s	1 Gb/s	1 Gb/s
Duration (s)	86.97	86.97	86.97	86.97	86.97
Speed (pps)	102.1	102.1	102.1	102.1	102.1
% S. Match	-	100	100	100	100
% D. Match	-	98.23	98.23	98.23	98.23

TABLE V. RESULTS FOR "MALWARE" TRACE

	Original	D	1.5×D	2×D	2.5×D
# Packets	1103	1103	1103	1103	1103
# TCP Flows	92	92	92	92	92
# UDP Flows	0	0	0	0	0
Link Speed	Unknown	1 Gb/s	1 Gb/s	1 Gb/s	1 Gb/s
Duration (s)	100.75	100.75	100.75	100.75	100.75
Speed (pps)	11.0	11.0	11.0	11.0	11.0
% S. Match	-	100	100	100	100
% D. Match	-	98.01	97.55	97.28	97.10

TABLE VI. RESULTS FOR "ENTERPRISE" TRACE

	Original	D	1.5×D	2×D	2.5×D
# Packets	54014	54014	54014	54014	54014
# TCP Flows	1	1	1	1	1
# UDP Flows	0	0	0	0	0
Link Speed	Unknown	1 Gb/s	1 Gb/s	1 Gb/s	1 Gb/s
Duration (s)	60.01	61.45	60.04	60.01	60.01
Speed (pps)	900.1	879.0	899.6	900.1	900.1
% S. Match	-	100	100	100	100
% D. Match	-	60.15	51.98	46.98	42.34

A detailed summary of performance results for data sets 1-4 is provided in Tables III-VI. We generated both TCP and UDP statistics to demonstrate that DSSR is viable for more than simply TCP. The first point to notice is that DSSR accurately recreates the exact number of packets, TCP flows, and UDP flows from the original trace. As can be seen from Tables III and VI where the original trace speed exceeded the capabilities of DSSR, increasing d resulted in a decreased percent of direct match packets but an increased performance in terms of average speed and duration. The more encouraging result, though, is that all tests cases produced very strong ordering results (100% in fact), no matter the value of d being used, when only semantically related packets are considered.

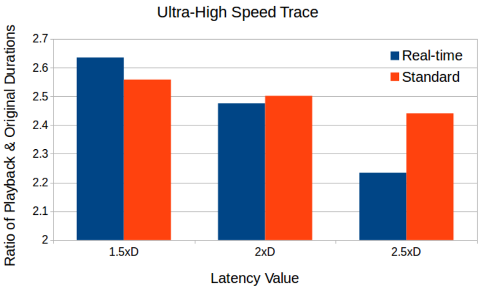


Fig. 8. Effect of real-time scheduling. The time performance improves slightly when $d = 2 \times D$ or $2.5 \times D$, but the average replay duration is still more than twice as slow as the original "Ultra High Speed" trace.

Effect of real-time scheduling: There is a performance limit with a software replay tool like DSSR. We have found that DSSR would take more than twice the original trace duration to replay the "Ultra High Speed" trace. This is expected given the laptops' limited hardware capability. We tried to increase its performance by assigning it the highest priority possible (i.e. "real time") for a non-kernel process. As can be seen in Figure 8, a slight but not consistent speed increase can be obtained from the real-time priority.

VI. CONCLUSION

We have shown DSSR to be a general-purpose network traffic replay tool that is directional and capable of maintaining full protocol semantics. By design, it provides the flexibility of relaxing nonessential packet ordering constraints in order to boost the replay speed, by simply adjusting the timestamps of packets replayed by the host(s) simulating external destinations.

DSSR is designed to be extensible. Other future work involves adding the following functionalities: (i) ability to use multiple VMs to increase speed performance, (ii) replaying to a routed network with middleboxes in different subnets, (iii) generation of packet arrival statistical models from a given collection of traces, (iv) support for IPv6, (v) detection and handling of packet errors or losses during replay, and (vi) support for encryption below the application layer.

REFERENCES

[1] Boost c++ libraries. http://www.boost.org.

[2] Capture the hacker 2013 competition: HONEYBOT. http://www.snaketrap.co.uk/pcaps/hbot.pcap.

[3] libcrafter: A high level library for C++ to generate and sniff network packets.

[4] TCPreplay sample captures: Bigflows.pcap. http://tcpreplay.appneta.com/wiki/captures.html.

[5] R. Craven, R. Beverly, and M. Allman. A middlebox-cooperative TCP for a non end-to-end Internet. In *ACM SIGCOMM*, Aug. 2014.

[6] W.-C. Feng, A. Goel, A. Bezzaz, W.-C. Feng, and J. Walpole. TCPivo: A high-performance packet replay engine. In *Proc. ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, MoMeTools '03, pages 57–64, 2003.

[7] S.-S. Hong and S. Wu. On interactive Internet traffic replay. In A. Valdes and D. Zamboni, editors, *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 247–264. Springer Berlin Heidelberg, 2006.

[8] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 27–38.