Low-Storage Capture and Loss Recovery Selective Replay of Real Flows

Ying-Dar Lin, Po-Ching Lin, Tsung-Huan Cheng, I-Wei Chen, and Yuan-Cheng Lai,

ABSTRACT

Capturing and replaying real flows are important for testing network security products. However, capturing real flows demands a high storage cost and runs a risk of capture loss, which makes the replay inaccurate. Replaying real flows should be accurate and stateful to adapt to the reaction of the device under test. It should also efficiently reproduce a defect and help developers identify the flows triggering defects. Therefore, this work first presents the (N, M, P) capture scheme which begins with, for each connection, capturing at most N bytes of application payload and then at most M bytes of application payload for at most each of the subsequent P packets in the same connection. This scheme reduces 87 percent of storage cost while retaining 99.74 percent of original events. This work develops a tool named SocketReplay with the mechanisms of loss recovery, stateful replay, and selective replay. Loss recovery tracks TCP sequence numbers to identify capture loss and recovers incomplete flows with dummy data. Stateful replay maintains the states in the TCP/IP stack to replay real flows. Selective replay incrementally selects flows to replay. The results show that SocketReplay can accurately and efficiently reproduce product events and significantly decrease the volume of replayed packet traces.

INTRODUCTION

The network research community, product developers, and testers demand network traffic for testing network applications, systems, and protocols. Two main approaches can generate network traffic: model-based traffic simulation and trace-based traffic replay. The former simulates network traffic according to protocol specifications, and it is easy to configure the desired parameters such as request formats for each test case. The latter replays packet traces captured in real environments, and the realistic traffic involves complex network scenarios and behaviors such as network attacks, peer-to-peer (P2P), video streaming, online games, as well as proprietary protocols that are hard to generate by simulation. Therefore, replaying trace-based traffic can trigger many product defects not found by simulating model-based traffic.

Replaying trace-based traffic is more complicated than it may appear to be at first glance. An intuitive approach is replaying the traces in the sequence based on the timestamp of each packet, which represents the time when a packet was captured. Tcpreplay (tcpreplay.synfin.net) is an example. However, this approach may be unable to efficiently and accurately replay meaningful traffic to devices under test (DUTs), such as network address translation (NAT) devices or intrusion detection systems (IDSs). Many existing works have contributed to raise the efficiency of traffic replay [1–4] and improve its accuracy [5–8]. Some studies attempt to replay accurately at the network or transport layer [5, 6], while others attempt to do at the application layer [7, 8]. The studies all require complete traces of packets to guarantee the accuracy of traffic replay.

In large networks, packets may get lost during capturing due to limited input/output (I/O) speed of the network card, memory, or disk, making existing replay methods unsuitable since they need complete traces. We call this case *capture loss* throughout this work. Another significant problem for packet capture is managing the (potentially) huge storage space requirements. Therefore, appropriate methods are required to resolve the problems of capture loss and huge storage space requirements.

In this work, we design and implement a tool named `SocketReplay`, which provides an effective way to capture and replay large-scale network traffic. `SocketReplay` involves four primary features:

- *Low-storage capture*, which records only partial network traffic according to the traffic types to significantly reduce storage cost
- *Loss recovery*, which recovers incomplete connections due to capture loss to replay a complete TCP stream
- *Stateful replay*, which mimics the TCP/IP stack and replays payloads to maintain the TCP semantics
- *Selective replay*, which reproduces the abnormal events, such as bugs of the DUT or alerts to be analyzed, with minimal replayed traces to efficiently analyze the events

`SocketReplay` has been used for our internal testing, and may be openly available when the code is stable.

BACKGROUND

ISSUES OF TRAFFIC CAPTURE

The quality of the captured traffic affects the quality of tests or experiments based on packet traces, but capturing network traffic at high speed in a large network is nontrivial due to limited I/O speed and storage. Therefore, *storage cost* and *completeness* of packet traces are two important issues.

Storage cost: Network traffic from a large number of hosts can fill up an ordinary hard disk in a few hours. For example, the average throughput from 1374 hosts, a portion of our campus hosts, is around 600 Mb/s. The amount can fill up a 1 Tbyte hard disk in just 4 h.

Completeness: It is hard to capture every session completely in a large network. First, packets may get lost during capture due to inherent system limitations, such as I/O speed. Second, some connections may have been established before the capture starts, and the packets prior to the capture are missed. As a result, the incompleteness decreases the accuracy of replay because the replaying process may lack some critical packets.

ISSUES OF TRAFFIC REPLAY

The goal of traffic replay is to trigger potential events on a DUT, and help the testers and developers analyze the events. The *accuracy* and *efficiency* are two major concerns in the replay. The accuracy affects the validity of testing a DUT as well as event triggering. The efficiency of traffic replay affects the time spent in the replay and the difficulty in the event analysis.

To accurately replay traffic regarded as valid network traffic by DUTs, the replay must follow the states of protocols, especially those of TCP and application protocols, and send out the correct packets in the correct order and direction to test the DUTs. Furthermore, the replay has to alter the network traffic in response to some DUTs, say NAT devices, which modify the network traffic passing by.

The efficiency can be measured by the traffic volume and the time required to reproduce an event in the traffic replay. For event analysis, the lower the volume of traffic replayed to reproduce events, the easier and faster testers and developers can analyze them. Reducing the volume of replayed traffic, however, may accidentally drop critical events of interest, so there are trade-offs in the efficiency and accuracy of traffic replay.

RELATED WORK

Several tools can replay the packet traces. For example, TCPreplay simply replays the packets in a PCAP file (www.tcpdump.org) one by one at the specified rate, without maintaining the connection state. Tomahawk (tomahawk.sourceforge.net) or TCPreplay with the help of TCPPrep in the same suite can split the packets into those from the client and those from the server, and replay them between two network interfaces.

Some studies can maintain the states of the network and transport layers for traffic replay. TCPopera [5] uses four heuristics to follow the TCP/IP stack. Monkey [6] replays web traffic by using socket programming to emulate TCP stack and Dummynet (info.iet.unipi.it/_luigi/dummynet) to emulate network conditions. Avalanche (www.spirent.com/ Solutions-Directory/Avalanche), a commercial product, uses a trace file to emulate high-volume network traffic of concurrent users. Furthermore, some studies [7, 8] support the states of the application layer for traffic replay. However, these tools are still insufficient for replaying network traffic captured from a large network. Although most of them can inspect the states in the TCP/IP stack, they cannot recover from capture loss or replay traffic selectively, which are helpful to accurately and efficiently reproduce events. We develop a tool named `SocketReplay` to solve the problems in this work.

The accuracy and efficiency are two major concerns in the replay. The accuracy affects the validity of testing a DUT as well as event triggering. The efficiency of traffic replay affects the time spent in the replay and the difficulty in the event analysis.

THE CAPTURE SCHEME AND DESIGN OF SOCKETREPLAY

This section details the design of SocketReplay and capture scheme for a large network.

DESIGN GOALS

The capture scheme is intended to store valuable traffic that is just sufficient to trigger events. In other words, the scheme ignores parts of the payloads in some packets that are generally unlikely to trigger events, and decreases the total traffic volume replayed to trigger events.

The design of SocketReplay has three goals for traffic replay:

- Recovering incomplete TCP connections due to capture loss
- Statefully replaying in the network and transport layers because many DUTs, including NAT devices, proxies and security appliances, may modify TCP/IP headers
- Replaying selected packets to reproduce events

The design helps product developers easily analyze the sessions or connections that trigger events.

LOW-STORAGE CAPTURE SCHEME

In this work, the three thresholds (N, M, P) determine how much data should be captured for a connection. Let L_i be the payload length (in bytes) of the ith packet in the connection and k be the minimum value such that $\sum_{i=1}^{k} L_i \geq N$. If no such k exists, it means the connection is too short, and the scheme can capture the entire connection; otherwise, this scheme captures at least the first $\sum_{i=1}^{k} L_i$ bytes in the application payloads in a connection. Starting from the (k + 1)th packet (if any), the scheme captures at most the first M bytes in each payload of at most P subsequent packets.

N is set based on the assumption that most events can be triggered within the initial bytes in one connection and most traffic is from a few long connections (i.e., the *heavy-tail* characteristic) [9]. M is set because in some applications such as MSN and Skype, most application headers appear in the first few bytes of individual

Packet no.	1	2	3	4	5	6
Direction	A→B	A→B	B→A	A→B	A→B	B→A
Seq., ack.	a, b	a+10, b	b, a+20	a+20, b	a+40, b	b, a+50
Data length	10	10	0	20	10	0

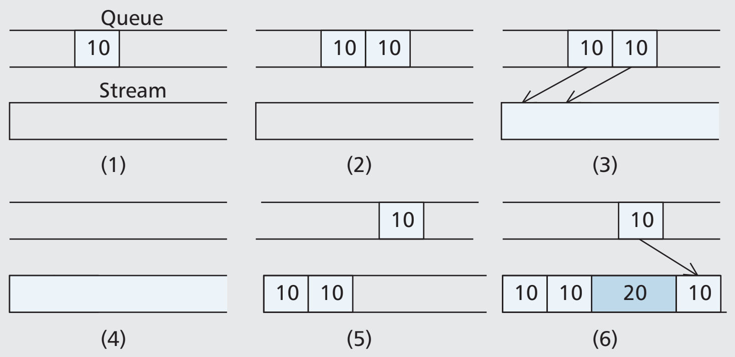

Figure 1. *An example of loss-recovery for an established TCP connection with packet capture loss.*

packet payloads, and retaining them is helpful for further analysis and identification of these applications. This work retains the first *P* packets after *N* bytes to avoid capturing unnecessary and bulk data (e.g., a large file downloaded from a web site).

SOCKETREPLAY

SocketReplay is a stateful traffic replay tool suitable for a large-scale environment. There are three stages described as follows:
- *Loss recovery* reconstructs complete traffic traces.
- *Stateful replay* mimics hosts to generate traffic without breaking protocol semantics.
- After triggering events by stateful replay, *selective replay* narrows down the scale of the replayed connections to reproduce events.

Loss Recovery and Stateful Replay — Loss recovery parses the incomplete connections and inserts dummy bytes to recover them. A captured connection could be incomplete due to the low-storage capture scheme or capture loss. The length of missing data can be derived from the sequence number and acknowledgment number in the TCP headers of the packets.

Figure 1 illustrates the loss recovery process packet by packet with a trivial example, in which six packets are transmitted in a connection between host A and host B (see the upper part of this figure). The fourth packet is lost due to capture loss during the capture process. The illustration demonstrates how SocketReplay handles the packets in the queue and the stream buffer to recover the stream bytes from host A to host B in the replay:
- The first packet is temporarily queued because we are unsure whether it can reach the destination when seeing only this packet.
- The sequence number of the second packet is checked to see whether the two segments are overlapped. Again, this packet is queued because we are unsure whether it can reach the destination.
- The ACK in the third packet from host B confirms successful transmission of the first two packets. Therefore, we put 20 bytes of data into the stream buffer, which stores the stream bytes to be recovered in the replay.
- The fourth packet is lost.
- The sequence numbers of the second and fifth packets are discontinuous due to the lost packet. The discontinuity may also happen when the fourth and fifth packets are out of order. Therefore, we are unsure which case it is.
- The ACK in the sixth packet implies that the fourth packet is not captured, and the data of the fifth packet has been transmitted successfully because the acknowledgment number is correct, but the fourth packet is unseen. Therefore, we put 20 bytes of dummy data preceding the data of the fifth packet into the stream buffer.

After the recovery process, the stream buffer contains 50 bytes of data for the replay. Note that SocketReplay just replays the packets from both hosts involved in a connection with two network interfaces on the same host, rather than running the applications or replaying the packets to the application on another host. Therefore, content loss may affect the analysis of the DUT, say an IDS, but not the application on either host of the connection. While this stage can reconstruct TCP connections, it is unable to completely recover UDP sessions from loss recovery due to lacking reliable reconstruction information in UDP. This is a fundamental limitation of UDP. In addition to determining the packet order (i.e., the sequence of packets to be replayed) and emulating TCP connections, as previous works did [5, 6], this work solves the packet order problem in this stage by inserting dummy packets with the correct sequence numbers.

Selective Replay — Some events will be triggered after the stage of replaying traces *statefully* to a DUT. An event can be an alert of detecting an attack, a report of finding a virus, and so on, depending on the DUT. To identify the causes of events and reproduce the events efficiently, it is better to find out the critical connections from the huge packet traces that trigger the events. This stage can *selectively* replay critical connections according to the event information and the replay log from the stage of stateful replay. The event information includes event time, connection information, messages of errors or alerts, and so on. The replay log includes the time of connection establishment and termination. This stage selects only the potential connections to test whether the event can be reproduced. If not, more connections are included in the replay. The algorithm is described below:

```
$S = \phi$; {the set of selected connections.}
 {Step 1}
 {5-tuple information: source/destination IP addresses, source/destination ports, protocol}
**if** event comes with 5-tuple information **then**
   replay specified connection and call checking();
**end if**
```

```
{Step 2}
if event includes source/destination IP then
  replay connections between two hosts and call
  checking();
end if
{Step 3}
$E$ = the time at which an event occurs;
$T = 0$;
if some connections exist over the time $E$ then
  replay the connections and call checking();
end if
while $E – T \geq 0$ and there are connections closed
between $E – T$ and $E$ do
  replay connections during the time slot and
  call checking();
  increase T by a second;
end while
report failure of reproducing the events ;

function checking()
$S = S \cup$ {the replayed connections};
if triggered then
  Select connections in $S$ for replay and exit;
end if
```

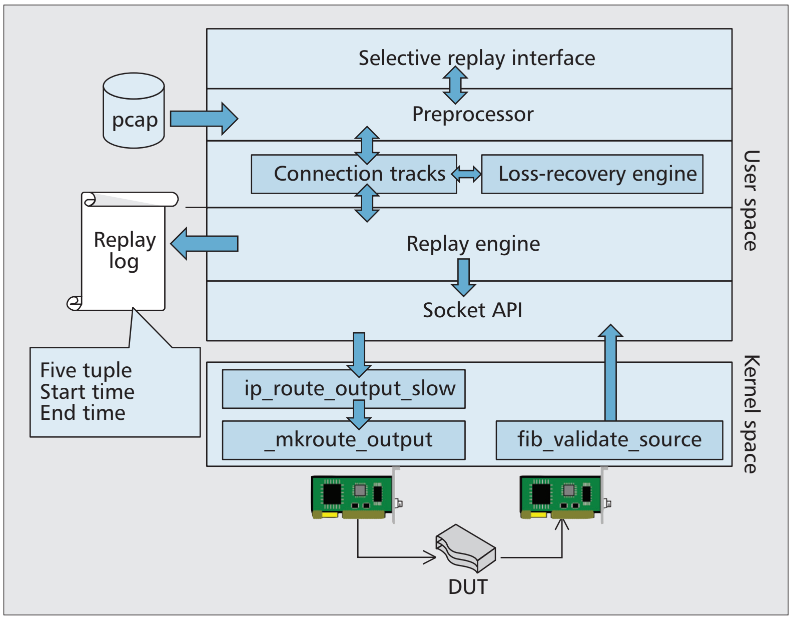

Figure 2. `SocketReplay` *components.*

SocketReplay Components and Implementations

SocketReplay emulates interactive hosts as a client or server with two network interfaces. Figure 2 illustrates the components of SocketReplay. Because the volume of real flows is too large to load all packets from hard disks into the memory in advance, SocketReplay has to read and replay packets simultaneously. The details of the components and implementations are described as follows.

Preprocessor

The preprocessor uses the *libpcap* library to read packets from a hard disk and reassemble the IP fragments into IP datagrams if necessary. The preprocessor then outputs TCP and UDP packets to *connection tracks*, the buffers in which packets of the same connection are stored.
Connection Tracks and Loss-Recovery Engine

A great number of connections will be established in a large environment. To quickly find out to which connection track a packet belongs, SockeReplay uses a hashing algorithm described as follows. The current implementation supports only IPv4, and the extension to IPv6 is left to the future work.

- `SocketReplay` sorts the source and the destination IP addresses so that packets in both directions of a connection are mapped to the same hash.
- For either of the two IP addresses, `SocketReplay` splits the 32-bit address into two 16-bit values and performs a bitwise XOR on them.
- `SocketReplay` performs 8-bit left-shift operation on the 16-bit value from the larger IP address (appending an 8-bit 0's), and performs a bitwise XOR on the 24-bit value with the 16-bit value from the lower IP address.
- `SocketReplay` derives a 24-bit hash value in the hash table as an index to a certain connection track. Because some of the connections between two hosts may belong to the same session, `SocketReplay` tracks these connections in a linked list.

If the captured data is found incomplete during the tracking, the loss-recovery engine will inspect the TCP states and recover the missing data to make every connection complete. SocketReplay can be configured to start replaying when reading a certain number of packets into the connection tracks. It does not have to read all the captured packets in the packet trace.

Replay Engine

The replay engine is implemented using socket programming to establish connections with reverse engineering, so `SocketReplay` needs to bind many IP addresses and port numbers to the interfaces connected to the DUT. The diversity of IP addresses in the captured traces may influence product testing. For example, the IP addresses of the DUT interface and the firewall rules may be not configurable to accommodate such diversity. Therefore, SocketReplay maps each IP address appearing in the packet traces to an IP address in a "/24" subnet, and it assigns the subnet of IP addresses to the network interface in advance. The mapping and assignment are not performed during replay to avoid degrading the performance of traffic replay. If the "/24" subnet is unable to accommodate the total number of IP addresses in the packet traces, we will enlarge the subnet by lowering the "/N" value for the mapping.

The replay log has the information such as time and the five tuples of each replayed connection. After a packet is sent through an interface, `SocketReplay` checks whether it is received on the other network interface. If `SocketReplay` fails to receive the sent packet, it also logs the event on the replay log.

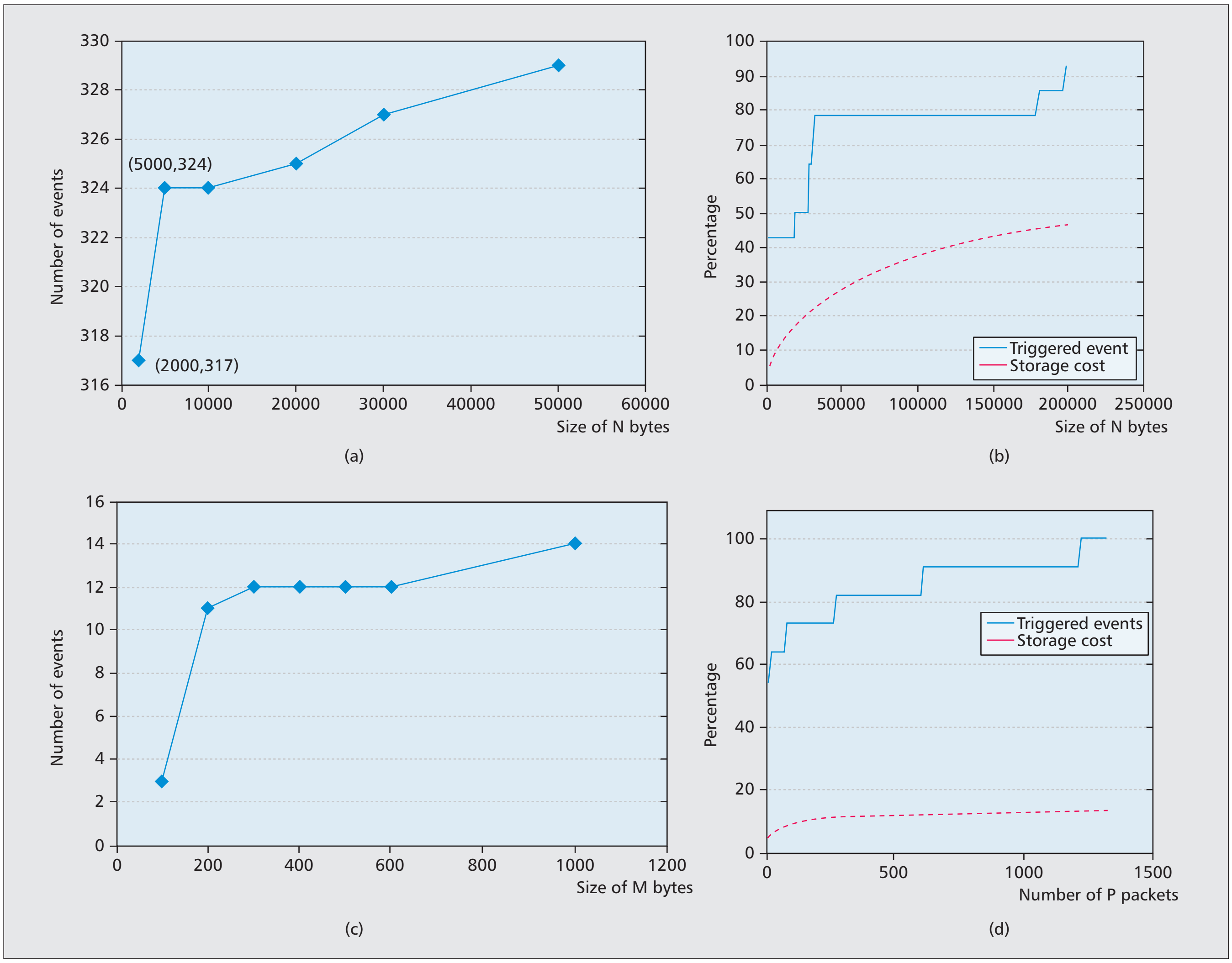

Figure 3. *a, c) Number of triggered events after setting the thresholds (N, 0, 0) and (N, M, ∞); (b, d) Percentage of storage cost and triggered events after setting the thresholds (N, 0, 0) and (N, M, P).*

Kernel Modification

SocketReplay uses the socket API to emulate various clients and servers appearing in the packet trace on the same host, so it is necessary to modify the routing policy in the kernel. This work modified three functions in the source of Linux kernel version 2.6.20.3. First, the function `ip_route_output_slow` in `net/ipv4/route.c` is modified to overwrite the outgoing interface, so that packets with the mapped IP addresses mentioned earlier can be sent from the desired outgoing interface. Second, the function `__mkroute_output` in `net/ipv4/route.c` is modified to overwrite the default gateway. SocketReplay therefore can forward the packets to gateway devices such as NAT in the replay when it is necessary. Third, the function `fib_validate_source` in `net/ipv4/fib_frontend.c`, which verifies the validity of source addresses, is modified to accept packets coming from the same host.

Selective Replay Interface

The selective replay interface implements the mechanism described earlier. `SocketReplay` knows how to replay selectively according to the event message from the DUT and the replay log from the replay engine. An event message contains at least a timestamp. If it contains five-tuple information, `SocketReplay` starts from step 1 in the 3-step procedure. If it contains a source and a destination address, `SocketReplay` starts from step 2. If it contains only the timestamp, SocketReplay starts from step 3.

Evaluation

In this section, we evaluated the capability of lowstorage capture, loss-recovery and selective replay of `SocketReplay` in a large-scale environment. The evaluations of low-storage capture focus on storing two types of events in network traffic: attack and virus. The evaluation of loss recovery and selective replay focuses on the ability of recovering capture loss and reducing the volume of selected traffic.

In the evaluation, we mirrored the network traffic of 1374 hosts from a core router to a capture device. SocketReplay then replayed real flows into an all-in-one commercial security appliance that can detect events such as attacks and viruses. The logs of these events from the

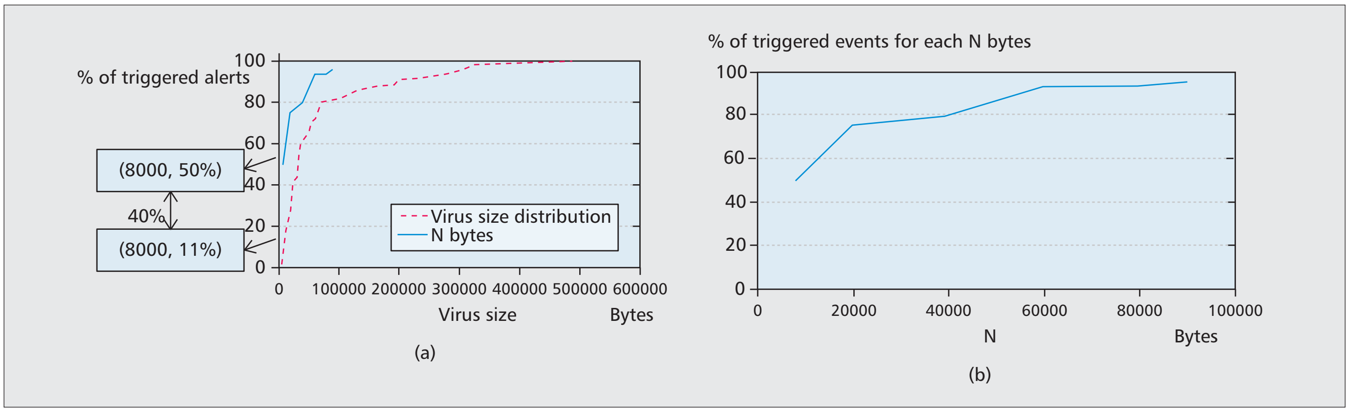

Figure 4. *Virus size distribution and thresholds (*N*, 0, 0) for reproducing virus events.*

system log were collected after the replaying process.

Test Results for Low-Storage Capture

We tried several combinations of the thresholds (*N*, *M*, *P*) described earlier to find out the best combination that produces the most events and reduces storage space efficiently for these two types of events. Since different security appliances analyze network traffic in various ways (e.g., using different detection rules), it is not generally possible to specify the optimal thresholds for every appliance. SocketReplay users can use an approach similar to that below to tune the thresholds for a specific appliance.

Attack — This work collects 1929 attack events triggered by replaying real flows to the security appliance. `SocketReplay` can reproduce these events with step 1 of selective replay. Also, only 333 connections that trigger an event have data lengths longer than 2000 bytes.

We set the threshold to (*N*, 0, 0), meaning `SocketReplay` replays the first *N* bytes of application payload in the connections, and observe whether the security appliance can detect the events. As Fig. 3a shows, we found that 317 of 333 events from the connections longer than 2000 bytes were triggered by simply replaying the first 2000 bytes of data per connection, and a few more events were triggered when we adjusted *N* from 5000 to 50,000. We picked 16 events not triggered when using the thresholds (2000; 0; 0), and increased *N* to see the percentage of triggered events and storage cost, as Fig. 3b shows. If we want to cover most events, *N* should be very large, demanding large storage space. Therefore, replaying 2000 bytes of data in each connection should be sufficient to trigger most attack events.

Next, we conducted another experiment with the thresholds (2000, *M*, ∞) to replay 16 events that cannot be triggered by using thresholds (2000, 0, 0). As Fig. 3c shows, we found that when *M* is 200 bytes, 11 of 16 events are triggered.

We then adjusted the threshold *P* to find out the relation of storage cost and events triggered by the thresholds (2000, 200, ∞). As Fig. 3d shows, we found that when *P* is set to 1300, 11 events are all triggered, and 87 percent of storage is reduced. Also, when *P* is set to 200, 8 of 11 events are triggered, and 90 percent of storage is reduced.

To sum up, besides *N*, the threshold *M* is effective to trigger more attack events. If we set the thresholds (*N*, *M*, *P*) to be (2000, 200, 1300), the low-storage capture scheme can record 99.74 percent of events that can be triggered by `SocketReplay` and reduce 87 percent of storage cost. We set *P* to 1300 in order to trigger rare events that cannot be triggered with the thresholds (*N*, *M*, 200).

Virus — This subsection finds out the capture scheme for collecting virus events. This work collected computer viruses from VX Heavens (vx.netlux.org), which contains massive, continuously updated virus samples and sources. We manually transferred viruses in 44 FTP sessions to trigger events from anti-virus systems and captured these sessions. Next, we applied the three thresholds of SocketReplay to replay these sessions and observe whether they can trigger these events again.

The dotted curve of Fig. 4a presents the distribution of virus sizes. This curve shows that 20 percent of viruses are larger than 100 kbytes, longer than the payload length of attack traffic. Figures 4a and 4b present the percentage of triggered events by SocketReplay to replay 44 FTP sessions with the thresholds (*N*, 0, 0) in the solid curves. We do not consider adjusting *M* and *P* in this case because the virus samples are a continuous byte stream during transmission, and it is meaningless to retain the application headers in individual packets by adjusting the two thresholds. Although 89 percent of the viruses are longer than 8 kbytes, replaying the first 8 kbytes of each connection can trigger 50 percent of the virus events. We observed that replaying the first 60 kbytes is sufficient to trigger 93 percent of virus events and reduce 70 percent of storage cost.

Test Results for Loss Recovery

Completeness is an important factor to accurately replay on security appliances. We prove this by conducting two simple experiments on a complete connection that can trigger an event by TCPreplay. First, we removed the three-way handshake and replayed the connection again.

We found it failed to trigger the event because security appliances did not track this connection on its session table. Second, we removed a data packet after the three-way handshake and replayed it again. We found it also failed to trigger the event because the sequence numbers of packets are unreasonable for security appliances.

We sampled 22,185 real flows within 30 s. Figure 5 illustrates the status of each flow. There are 10,660 (9017+1643) flows established before the capture starts, so we cannot capture their three-way handshake. There were 7753 (7527+226) unidirectional flows because some hosts sent a SYN packet, but the destination host refused or did not reply to the establishment. There are also 3772 (3323+449) flows that have complete three-way handshake, 254 flows that have capture loss found by inspecting acknowledgement numbers, and 2318 (1643+226+449) flows are closed by a reset packet (RST). From the experiment, if we use TCPreplay to replay all the traffic, the shaded area of 10,914 (9017+1643+254) flows cannot be accurately replayed on security appliances. However, SocketReplay can replay them accurately by the loss-recovery mechanism.

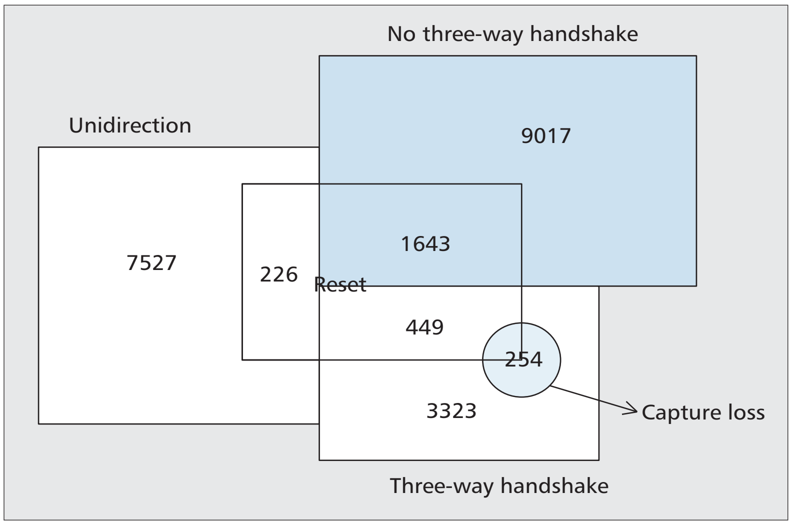

Figure 5. *Status of real flows.*

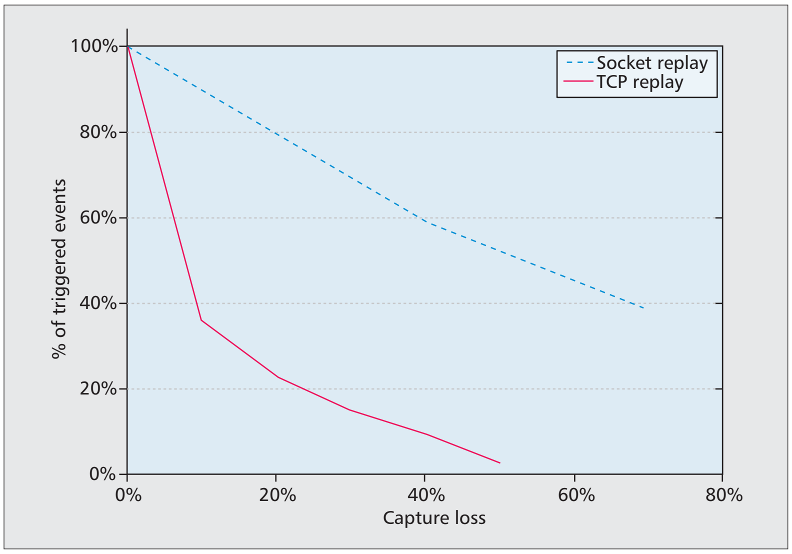

Figure 6. *The effect of capture loss on event reproductions for* `SocketReplay` *and TCPreplay.*

Next we examined the effectiveness of loss recovery and stateful replay by using TC (tldp.org/HOWTO/ Traffic-Control-HOWTO) to simulate capture loss. We compared SocketReplay and TCPreplay to see how the capture loss affects the reproduction of 1929 attack events. Figure 6 presents that the proportion of triggered events of SocketReplay and the rate of capture loss are decreasingly proportional because some of the lost packets are critical to trigger events. However, the proportion of triggered events of TCPreplay drops quickly with increasing capture loss because capture loss affects the accuracy of all the replayed connections.

Test Results for Selective Replay

Attack Events — This work collects 1929 attack events from real flows triggered by a security appliance. SocketReplay successfully reproduced the events with the first step of selective replay (i.e., replaying the connections with the same 5-tuple information as that of the events), which is efficient to replay from large packet traces.

Virus Events — An FTP session contains a control connection and a data connection. Both should be included to reproduce the complete session. Moreover, in the active mode or the passive mode of FTP protocol, the control connection sends the IP address and port number, which should be coherent with those of the data connection. In this work, we collect virus events triggered by transmitting virus through FTP protocol to ensure that FTP sessions can be replayed accurately on security appliances. SocketReplay can successfully reproduce all these virus events in the second step of selective replay (i.e., replaying the connections of the same pair of source and destination hosts as that of the events), which is efficient to replay from large packet traces.

Conclusions

The low-storage capture scheme and replay tool SocketReplay provide a total solution to testing with real flows from large-scale environments. The thresholds of the capture scheme should be adjusted according to the traffic source, traffic of concern, and types of DUT. We develop a three-step procedure to determine the thresholds (N, M, P). In a campus-scale environment, the thresholds (2000, 200, 1300) are suitable for capturing attack traffic, which triggers 99.74 percent of original events and reduces 87 percent of storage; the thresholds (60000, 0, 0) are suitable for capturing traffic with viruses, which triggers 93 percent of original events and reduces 70 percent of storage. By loss recovery replay, `SocketReplay` can first recover the lost data and then replay the recovered traffic trace to trigger events accurately. Furthermore, `SocketReplay` can reproduce events efficiently with selective replay. This work provides an accurate and efficient way to play with real flows.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Council in Taiwan, and in part by grants from Cisco and Intel.

REFERENCES

[1] W. Feng *et al.*, "TCPivo: A High-Performance Packet Replay Engine," *Proc. ACM SIGCOMM Wksp. Models, Methods, and Tools for Reproducible Network Research*, Aug. 2003.
[2] G. A. Covington *et al.*, "A Packet Generator on the NetFPGA Platform," *Proc. 17th Annual IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 2009.
[3] T. Ye *et al.*, "Divide and Conquer: PC-Based Packet Trace Replay at OC-48 Speeds," *Proc. Testbeds and Research Infrastructures for the Development of Networks and Communities*, Feb. 2005.
[4] P. Kamath *et al.*, "Generation of High Bandwidth Network Traffic Traces," *Proc. Int'l. Symp. Modeling, Analysis and Simulation of Computer and Telecommun. Sys.*, Oct. 2002.
[5] G. H. Hong and S. F. Wu, "On Interactive Internet Traffic Replay," *Proc. Symp. Recent Advanced Intrusion Detection*, Sept. 2005.
[6] Y. C. Cheng *et al.*, "Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying," *Proc. 2004 USENIX Annual Tech. Conf.*, June 2004.
[7] W. Cui *et al.*, "Protocol- Independent Adaptive Replay of Application Dialog," *Proc. 13th Annual Network and Distrib. Sys. Security Symp.*, Feb. 2006.
[8] J. Newsome *et al.*, "Replayer: Automatic Protocol Replay by Binary Analysis," *Proc. ACM Conf. Computer and Commun. Security*, Oct. 2006.
[9] S. Kornexl *et al.*, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," *Proc. ACM Internet Measurement Conf.*, Oct. 2005.

BIOGRAPHIES

YING-DAR LIN (ydlin@cs.ccu.edu.tw) is a professor of computer science at National Chiao Tung University (NCTU), Taiwan. He received his Ph.D. in computer science from the University of California at Los Angeles (UCLA) in 1993. He served as the CEO of Telecom Technology Center during 2010–2011 and a visiting scholar at Cisco Systems in San Jose, California, during 2007–2008. Since 2002 he has been the founder and director of the Network Benchmarking Laboratory (NBL, www.nbl.org.tw), which reviews network products with real traffic. He also cofounded L7 Networks Inc. in 2002, which was later acquired by DLink Corp. He recently, in May 2011, founded the Embedded Benchmarking Laboratory (www.ebl.org.tw) to extend into the review of handheld devices. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms, quality of service, network security, deep packet inspection, P2P networking, and embedded hardware/software codesign. His work on multihop cellular has been cited over 500 times. He is currently on the editorial boards of *IEEE Transactions on Computers*, *IEEE Network*, the *IEEE Communications Magazine* Network Testing Series, *IEEE Communications Surveys and Tutorials*, *IEEE Communications Letters*, *Computer Communications*, and *Computer Networks*. He recently published a textbook, *Computer Networks: An Open Source Approach* (www.mhhe.com/lin), with Ren-Hung Hwang and Fred Baker (McGraw-Hill, 2011). It is the first text that interleaves open source implementation examples with protocol design descriptions to bridge the gap between design and implementation.

PO-CHING LIN [M] (pclin@cs.ccu.edu.tw) received his B.S. degree in computer and information education from National Taiwan Normal University, Taipei, in 1995, and his M.S. and Ph.D. degrees in computer science from NCTU in 2001 and 2008, respectively. He joined the faculty of the Department of Computer and Information Science, National Chung Cheng University (CCU), Chiayi, Taiwan, in August 2009. He is currently an assistant professor. His research interests include network security, network traffic analysis, and performance evaluation of network systems.

TSUNG-HUAN CHENG (raijin@cs.nctu.edu.tw) received his B.S. and M.S. degrees in computer science from NCTU in 2007 and 2009, respectively. His research interests include network security and network forensics. He is currently a software engineer with MediaTek Company since 2010.

I-WEI CHEN (iwchen@nbl.org.tw) is the executive director of NBL at NCTU. He received B.S. and M.S. degrees in computer and information science from NCTU. He joined NBL in 2003. At NBL he is engaged in development of testing technologies for network and communication devices. He is especially interested in technologies using real-world network traffic to test products.

YUAN-CHENG LAI (laiyc@cs.ntust.edu.tw) received his Ph.D. degree in computer science from NCTU in 1997. In August 2001 he joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology, Taipei, where he has been a professor since February 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking.

By loss recovery replay, SocketReplay can first recover the lost data and then replay the recovered traffic trace to trigger events accurately. Furthermore, SocketReplay can reproduce events efficiently with selective replay. This work provides an accurate and efficient way to play with real flows.