# 國立交通大學

## 網路工程研究所

## 碩士論文

低儲存空間消耗的錄製真實流量

與回復有效狀態的重播真實流量技術

Low-Storage Capture and Loss-Recovery Stateful Replay

of Real Flows

研 究 生：鄭宗寰

指導教授：林盈達　教授

中 華 民 國 九 十 九 年 六 月

# 低儲存空間消耗的錄製真實流量

# 與回復有效狀態的重播真實流量技術

# Low-Storage Capture and Loss-Recovery Stateful Replay of Real Flows

研 究 生：鄭宗寰　　Student: Tsung-Huan Cheng

指導教授：林盈達　　Advisor: Dr. Ying-Dar Lin

**國立交通大學**

**網路工程研究所**

**碩士論文**

**A Thesis**

**Submitted to Institutes of Network Engineering**
**College of Computer Science**
**National Chiao Tung University**
**in partial Fulfillment of the Requirements**
**for the Degree of**
**Master**
**In**

**Network Engineering**

**June 2009**

**HsinChu, Taiwan, Republic of China**

**中華民國九十九年六月**

# 低儲存空間消耗的錄製真實流量

# 與回復有效狀態的重播真實流量技術

**學生: 鄭宗寰** **指導教授: 林盈達**

**國立交通大學網路工程研究所**

## 摘要

網路產品在真實流量上仍會遇到許多實驗室模擬網路流量測試所無法找到的問題，而這些問題可藉由重播真實流量的測試來找到。由於真實流量由許多真實使用者所產生，在錄製時會快速消耗儲存空間使得錄製時間無法很長，與漏錄影響重播的準確性。在重播時要追蹤流量的有效狀態以應付待測物對流量的反應，並且要能很快的重製事件的發生以便開發者除錯或尋找原因。因此本論文以(N, M, P)錄製機制針對每條連線只錄製連線的前 N 位元與錄製剩餘 P 個封包的前 M 幾位元來節省儲存空間，達到節省 87%的儲存空間但保留 99.74%的攻擊事件。並且實作實作 SocketReplay 重播工具以回復漏錄重播追蹤 TCP 串流使得漏錄對觸發事件的數量成比例的下降而不會驟降，有效狀態的重播使待測物認為流量是真實的，選擇性的重播以漸增方式階段性尋找造成事件的少數流量，達到重製事件僅需從千條的連線中挑出幾十條連線重製攻擊或病毒的事件。

**關鍵字: 流量重播、流量錄製、真實流量、測試、缺陷**

# Low-Storage Capture and Loss-Recovery Stateful Replay of Real Flows

**Student: Tsung-Huan Cheng** **Advisor: Dr. Ying-Dar Lin**

**Department of Computer Science**

**National Chiao Tung University**

## Abstract

Model-based traffic generated in the laboratory might not trigger some device defects found only by replaying traffic flows. However, capturing real flows might result in high storage cost and capture loss; the latter affects the accuracy of replay. Replaying real flows should be accurate and also stateful enough to adapt to device reaction. It should reproduce a defect efficiently in helping developers to identify the flows triggering the defect. Therefore, this work first presents the *(N, M, P)* capture scheme to capture *N* bytes per flow of data and *M* bytes of *P* packets after the N bytes. This scheme reduces 87% storage cost while retaining 99.74% of attack traffic. Next we develop a tool named SocketReplay with the mechanisms of loss-recovery, stateful replay, and selective replay to track TCP sequence numbers to identify capture loss, recover these incomplete flows, follow the TCP/IP protocol behavior, and incrementally select flows to replay. Numerical results show that SocketReplay retains the accuracy and statefulness in triggering device defects and could reduce replayed flows from thousands to tens.

**Keywords: traffic replay, traffic capture, real flows, testing, defects**

# Content

# List of Figures

# List of Tables



# Chapter 1 Introduction

Generating network traffic as if hosts interact with each other for testing new network applications, systems, and protocols is highly demanded for network research community, developer and tester. Currently, there are two main approaches, model-based traffic generation and trace-based traffic replay, to generate network traffic. The model-based approach generates network traffic according to protocol specifications and it is easy to configure the desired parameters such as request formats for each test case. The trace-based approach replays packet traces captured in real environments called real flows and it includes more realistic and varied network conditions and behaviors such as P2P, video streaming, on-line game and proprietary protocols which is hard to model. Therefore, using trace-based traffic replay can trigger many unexposed device bugs and alerts that are difficultly discovered by adopting model-based traffic generation.

The intuitive solution is to replay traffic as captured such as Tcpreplay [1]. However, the quality of replay might be low due to insufficient efficiency and accuracy. Efficiency means how fast the traffic can be replayed while accuracy represents whether replay is meaningful on testing DUT (Device Under Test). Many previous works had contributed to raise the efficiency of traffic replay [2] and improve its accuracy [3-8]. Some studies tried to replay network layer or transport layer accurately [4-6] while the others tried to replay application layer accurately [7, 8]. However, these existing studies all require the complete trace of packets to guarantee the accuracy of traffic replay.

In large-scale networks, the miss in capturing, called capture loss throughout the thesis, may happen due to limited I/O speed of network card, memory, or disk,

causing that the previous approaches are not very suitable to the environments. On the other hand, storing all captured traffic needs a lot of storage. Therefore, an appropriate method for large-scale networks is required to conquer the problems of capture loss and huge requirements of storage. Currently, the study of [9] showed how to capture packets in 10Gbps by splitting traffic and tuning parameters. Also, some studies [10-11] proposed a method to only capture partial traffic by using the heavy-tail nature of traffic to reduce storage cost.

In this thesis, we design and implement a tool named SocketReplay, which can provide an effective way to capture and replay large-scale network traffic. SocketReplay mainly includes four features: (1) *low-storage capture* records partial network traffic according to types of concerned traffic so that storage cost can be significantly reduced; (2) *loss-recovery* recovers incomplete connections due to capture loss so that a complete TCP stream can be replayed; (3)*stateful replay* mimics TCP/IP protocols and replay payloads so that TCP semantics can be maintained; and (4) *selective replay* reproduces the abnormal events, such as bugs or alerts, with minimal replaying traces so that the events can be analyzed efficiently.

The rest of this thesis is organized as follows. Chapter 2 shows the background issues of capture and replay in large-scale environments and related work. Chapter 3 and Chapter 4 describe the design and implementation of our proposed method SocketReplay, respectively. Chapter 5 displays evaluation of our work. Finally chapter 6 concludes this work and gives some future directions.

# Chapter 2 Background

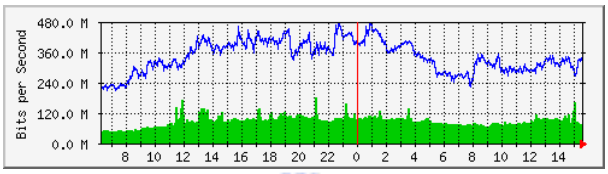
## 2.1 Issues of traffic capture in large-scale environment

The quality of traffic capture affects the quality of experiments based on these

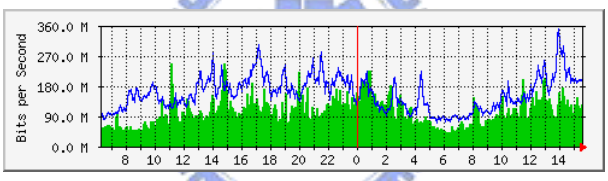
packet traces. However, capturing network traffic at high bit rate in large-scale environment is not an easy task due to limited speed and storage. Therefore, storage cost and completeness of packet traces are concerned issues.

### 2.1.1 Storage Cost

Because high bandwidth usage from lots of hosts, it can fill up a hard disk in few hours. Figure 1 drawn by MRTG (http://oss.oetiker.ch/mrtg/) shows daily bandwidth usage of 1374 hosts on campus. The average of total bandwidth usage is 600 Megabits per second which can fill up 1 Terabyte hard disk in 4 hours.




(a) Bandwidth usage between ISP (Internet Service Provider) and campus




(b) Bandwidth usage between TANET (Taiwan Academic Network) and campus

Figure 1. Daily bandwidth usage of 1374 hosts. The bar chart represents traffic from outside to campus and the line represents traffic from campus to outside.

### 2.1.2 Completeness

It is hard to capture complete sessions or even a complete connection in large-scale environment. The first reason is capture loss due to inherent system limitation of I/O speed. The second reason is that partial connections are established in the network before starting to capture. As a result, it may reduce the accuracy of replay because some critical packets are missed during the traffic replay.

## 2.2 Issues of traffic replay

The goal of traffic replay is to trigger more events and help testers to analyze

events. As described in the introduction, efficiency and accuracy are two major issues. The efficiency of traffic replay affects the time consumed and the difficulty of event analysis. If the traffic is not replayed accurately, it failed to test DUT validly and trigger events.

To replay traffic regarded as valid network traffic by DUTs, it must send out the correct packets in the correct order and direction so that it follows the states of protocols, especially TCP protocol and application protocols, to test DUTs which modify the network traffic passing by.

For example, replaying a TCP connection established from host A to host C on NAT (Network Address Translation) is in Fig. 2. First, SYN packet must be replayed from private network to public network. Second, the other interface must wait SYN packet and then send SYN_ACK packet with mapped network address and port number back otherwise this packet will be filtered by NAT and the connection cannot be established as Fig. 2(b) shows. Therefore, the program of traffic replay needs to inspect TCP header modified by NAT so that it knows the correct destination port of SYN_ACK packet. Besides the TCP header, this program needs to inspect content of application layer. For example, in the passive mode of the control connection in a FTP session, the client may ask the server to open a socket and the server puts IP address and port number in the payload. The socket information may be modified by NAT as Fig. 2(c) shows. Before establishing the data connection, the program needs to inspect the FTP header of the control connection to know the correct destination of a FTP data connection as Fig. 2(d) shows.

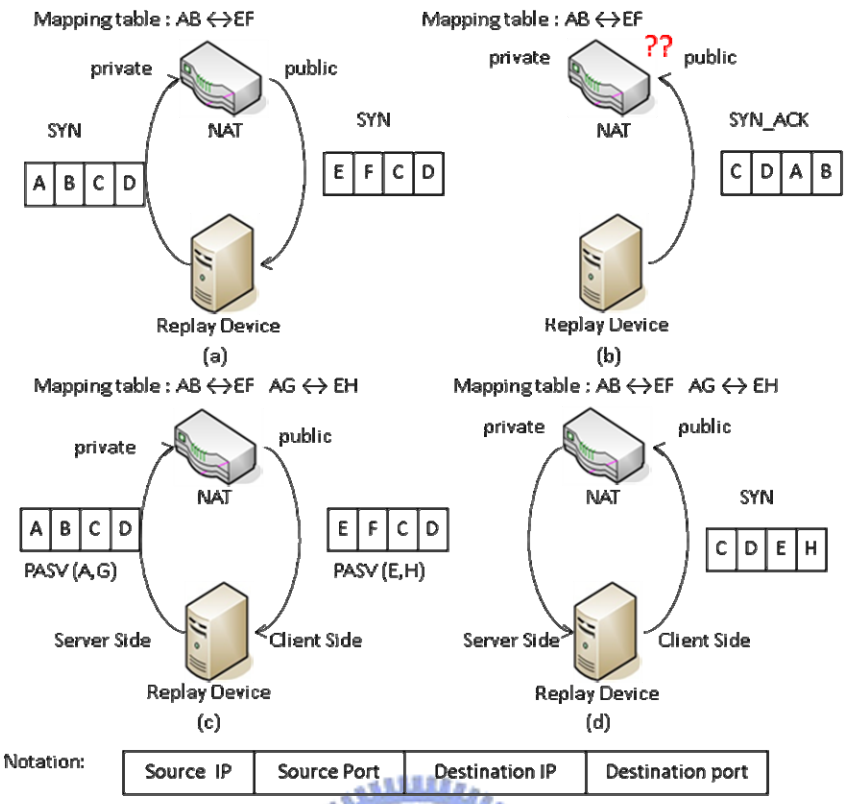



Figure 2. Replay network traffic on NAT. (a,b) replay a TCP connection (c,d) replay a FTP session.

The efficiency can be measured by how fast is the traffic replay andhow much traffic volume is required to reproduce an event. A higher speed of traffic replay can get a better progress of testing since the testing data (i.e. traffic) is transmitted to perform the test at a higher rate. However, the efficiency and the accuracy of traffic replay are trade-off because inspecting APP/TCP/IP protocols is time-consuming. For event analysis, the smaller volume of traffic replayed to reproduce events, the easier and faster testers analyze them.

## 2.3 Related work

Some research papers and tools proposed ways to capture in large-scale environment and traffic replay. The study of [9] did lots of experiments on testing the fully capture ability of community hardware configured with the combination of

parameters including number of preprocessors, operating systems, and buffer size. To reduce storage, Time Machine [10] uses a cutoff of 10K~20K bytes for each connection because the bulk of traffic in high-volume streams comes from just a few connections so that it can store small complete connections. Furthermore, the number of bytes can be adjusted dynamically by NIDS (Network Intrusion Detection System) [11].

There are many works on replaying the packet traces. TCPreplay [4] replays the same packets as recorded without inspecting any packet back from DUTs. On the other hand, Tomahawk [3] inspects the packet back from DUTs and then sends the next packet. These tools replay traffic without maintaining any state of network protocols.

Next there are studies solving network layer and transport layer of traffic replay. TCPopera [4] follows TCP/IP stack by using four heuristics. Monkey [5] replays web traffic by using socket programming to emulate TCP stack and using Dummynet (http://info.iet.unipi.it/~luigi/ip_dummynet.html) to emulate network conditions. Avalanche (http://www.spirentfederal.com/IP/Products/Avalanche), which is a commercial product, uses a trace file to emulate high-volume network traffic of concurrent users. These tools resolve network layer and transport layer protocols of traffic replay. Furthermore, some studies [7, 8] solve the application layer. However, the above available tools are not enough for replay network traffic from large-scale environment. The reasons are that although most of them inspect states of TCP/IP stacks, some of them cannot replay incomplete connections caused from capture loss and all tools cannot replay traffic selectively that reproduce the events and help testers to analyze them. In this work, we develop a tool named SocketReplay to solve these problems. Table 1 compares the available tools in terms of three demands needed for replay the traffic from large-scale environment.

Table 1. Comparison of available tools including open source program and commercial product

| Name | TCP/IP Stateful | Complete Connection | Selective Replay |
|---|---|---|---|
| TCPreplay | No | Not required | No |
| Tomahawk | Yes | Required | No |
| Monkey | Yes | Not required | No |
| Avalanche | Yes | Required | No |
| SocketReplay | Yes | Not required | Yes |

# Chapter 3 Design of SocketReplay and capture scheme

This chapter details the design of SocketReplay and capture scheme that are total solution of traffic capture and replay in large-scale environment.

## 3.1 Design Goals

The objective of capture is to store valuable traffic that is enough to trigger events. In other words, this capture scheme ignores part of payload of some packets which have low probability of triggering events and decreases the replayed traffic volume to trigger events.

The three objectives for traffic replay: (1) compact with incomplete TCP connections due to capture loss or above capture scheme; (2) statefully replay at TCP/IP layer because most of DUTs including NAT, proxy and IPS (Intrusion Prevention System) modify TCP/IP headers; and (3) replay selective packet traces to reproduce events. It helps designers to analyze easily what sessions or connections trigger events.

## 3.2 Low-Storage capture scheme

In this work, the capture scheme uses three thresholds (N, M, P) for the payload length of a connection being captured. The threshold N defines the number of bytes of data should be stored for each connection, the threshold M defines the number of bytes of data should be stored for each packet of a connection after length of stored data exceeds the threshold N, and the threshold P defines the number of packets should be stored after exceeding the threshold N. The N is set because we believe most of events can be triggered within the first bytes per connection. The M is set

because we believe after N bytes of connection, most events can be analyzed by application header and most application header is only small bytes per packet. The P is set because we believe most events can be triggered by first packets of each connection. Figure 3 illustrates an example, if the payload length of first three packets already meets the threshold of N bytes, the following P packets will be capture only first M bytes of payload and the other will be ignored.

<table>
<tr><td>Packet Sequence</td><td>1</td><td>2</td><td>3</td><td>4</td><td>…</td><td>P+3</td><td>P+4</td><td>…etc</td></tr>
<tr><td rowspan="2">Payload</td><td colspan="3" rowspan="2">N bytes</td><td>M bytes</td><td>M bytes</td><td>M bytes</td><td colspan="2" rowspan="2"></td></tr>
<tr><td colspan="3">Ignored data</td></tr>
</table>

Figure 3. Ignored data of three thresholds (N, M, P)

## 3.3 SocketReplay

SocketReplay is a stateful traffic replay tool that is suitable in large-scale environment. There are several stages described as follows. *Loss-recovery* reconstructs complete streams from capture scheme. *Stateful replay* minics hosts to generate traffic without breaking protocol semantic. After triggering events from stateful replay, *selective replay* narrow down the scale of replayed packet trace to reproduce events.

### 3.3.1 Loss-Recovery

*Loss-recovery* is an stage that parses the incomplete connections, i.e., broken streams which come from previous capture scheme or capture loss, into complete streams so that SocketReplay can replay the connection with the original length of the stream by inserting dummy bytes. The length of ignored data can be calculated from the TCP/IP header of the packet. The payload length of capture loss can be found by inspecting sequence number and acknowledge number of each packets.

For example, figure 4 shows an established connection of host A and host B. In real environment, the six packets are transmitted to destination successfully. During the capture, the fourth packet is lost due to capture loss. The following described the mechanism packet by packet as Fig. 4 illustrated. (1) The 1st packet is queued because we are not sure whether this packet can reach the destination. (2) The sequence number of 2nd packet is checked to see whether these two segments are overlapped. Again, this packet is queued because we're not sure whether the packet can reach the destination. (3) The ACK packet of host B verifies successful transmissions of 1st and 2nd packets. Therefore, we put 20 bytes of data into the stream. (4) The 4th packet is lost. (5) The sequence number of 2rd and 5th packet is not continuous. It can be happened when 4th and 5th packets are out of order. Therefore, we are not sure whether the 4th packet is lost. (6) This ACK verifies the 4th packet is lost and the data of 5th packet is transmitted successfully. Therefore, we put 20 dummy bytes and the data of 5th packet into the stream. After theses operation, the steam contains 50 bytes.

| Packet No. | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Hosts | A→B | A → B | B → A | A → B | A → B | B → A |
| Seq. , Ack. | a, b | a+10, b | b, a+10 | a+20, b | a+40, b | b, a+50 |
| Data length | 10 bytes | 10 bytes | 0 byte | 20 bytes | 10 bytes | 0 byte |

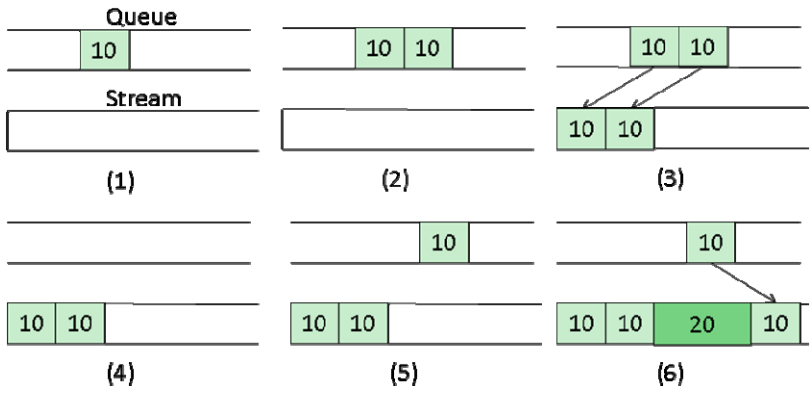



Figure 4. An example of loss-recovery for Established TCP connection with one packet capture loss

### 3.3.2 Stateful Replay

Because loss-recovery constructs a complete stream, this stage focuses on emulating all TCP and UDP connections. Previous work [8, 9] proposes methods to determine packet order and emulate TCP connections. This work solves packet order by the sequence of data inserted into the stream in the loss-recovery stage. Also, this work follows the work [5] using socket API to emulate TCP and UDP connections.

### 3.3.3 Selective Replay

After the stage of stateful replay, some events are triggered. To analyze how an event is triggered, i.e., to analyze what sessions, connections, or packets trigger an event, it is better to replay selectively from large packet traces. This work can achieve selective replay according to the event information and replay log from stateful replay. An event may include time information, connection information, and message of errors or alerts. Also, the time of connection established and closed can be obtained by replay log. Therefore, this work selects the potential connections to test whether it can reproduce the event. If it cannot reproduce the event, including more connections is needed. Figure 5 shows an example of an event includes the time information and the address of connection 5. The procedure of selected connection to replay is described as follows: (1) SocketReplay replays connection 5 to see whether it can reproduce the event. (2) If it cannot, SocketReplay includes connections with the same IP addresses of connection 5. In this case, it includes connection 1 and 5. (3) If it still cannot, SocketReplay includes established connections at time t. In this case, it includes connection 1, 2, and 5. (4) If it still cannot, SocketReplay includes last connections that are closed before time t. In this case, it includes connection 1, 2, 3, and 5. (5) If it still cannot, SocketReplay includes more connections that are closed before time t. Note that if the event does not provide connection information, SocketReplay will

skip step 1 and step 2.

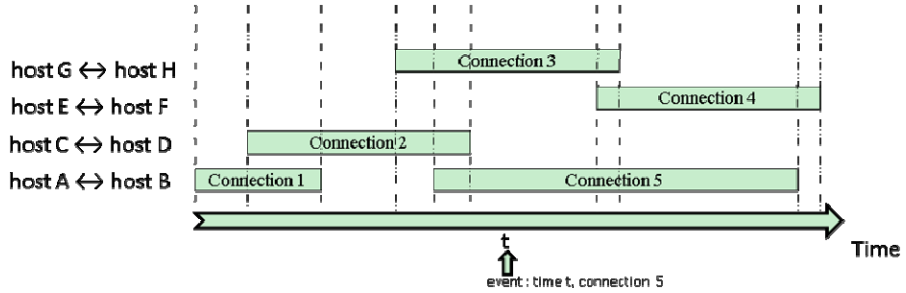



Figure 5. An example of selective replay

# Chapter 4 SocketReplay Components and Implementations

An installed SocketReplay host emulates interactive hosts by two network interface as a client or a server. Figure 6 illustrates the components of SocketReplay, which achieve the design as previous chapter described. Because the volume of real flows is too large to load all packets into the memory, SocketReplay has to read and replay packets simultaneously. We explain the details of these components and implementations in the rest of this chapter.

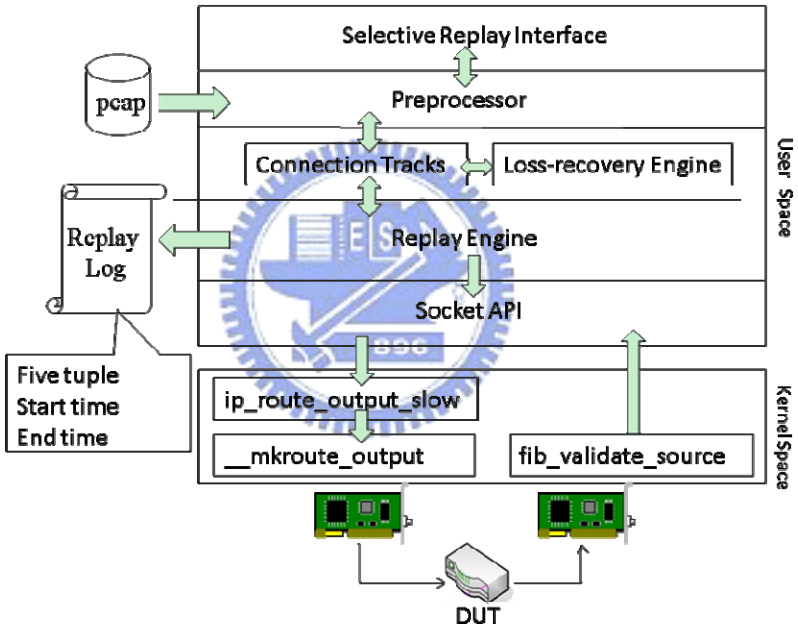



Figure 6. SocketReplay Components

## 4.1 Preprocessor

The processor uses libpcap library to read packets from a hard disk and reassemble the IP fragments into an IP datagram. Because most of packets are UDP or TCP packets, the preprocessor parses two kinds of packets to connection tracks.

## 4.2 Connection Tracks and Loss-recovery Engine

Because real flows contain lots of connections, SocketReplay should allow

constant time lookups of each packet which belong to a connection track by perfect hashing. Figure 7(a) shows how to computer hashes from IP addresses. First, SocketReplay sorts the source IP address and the destination IP address so that both directions of packets in a connection are mapped to the same hash. Second, for each IP address, SocketReplay splits 32-bits address into two 16-bits numbers and performs a bitwise XOR on them. Third, SocketReplay uses 8-bit left-shift operation on the 16-bit number which comes from larger IP address at previous step and performs a bitwise XOR on the 24-bit number and the 16-bit number. Finally, SocketReplay gets a 24-bit hash. This hash is used by hashing table as Fig. 7(b) illustrates. Because a session is usually composed of connections between two hosts, SocketReplay track these connections in a linked list. After tracking each packet, the loss-recovery engine will inspect the TCP states and recover incomplete connections to complete connections, i.e., TCP streams.

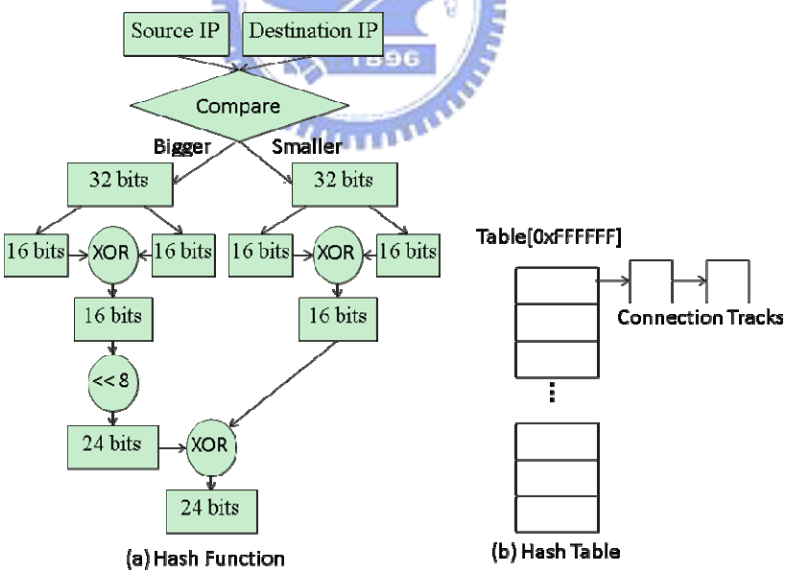



Figure 7. Implementation of Connection Tracks

## 4.3 Replay Engine

The replay engine is a reverse engineering implemented by socket programming to establish each connection. Therefore, SocketReplay needs to bind lots of IP

addresses and port numbers. In order to manage the connections easily, SocketReplay mapped each IP address to a class C IP address individually so that SocketReplay can assign the whole class C IP addresses to the network interface in advance.

The replay log reports the time and the connection information replayed by SocketReplay. After a packet is sent through an interface, SocketReplay checks whether each packet is received on the other network interface. If SocketReplay failed to receive the sent packet, it also logs the event on the replay log.

## 4.4 Kernel Modification

Since SocketReplay uses Socket API to emulate clients and servers at the same host, it replays packets in a virtual network, i.e. a loopback interface, and fails to transmit packets to the real network interfaces. Therefore, kernel modification of routing policy is needed. This work modified three functions of Linux kernel 2.6.20.3. First, the function ip_route_output_slow of the source code net/ipv4/route.c is modified to overwrite the outgoing interface so that packets can be sent to the real network interface with the source IP address of the packets assigned. Second, the function __mkroute_outpu of the source code net/ipv4/route.c is modified to overwrite the default gateway so that SocketReplay can replay network traffic on gateway devices such as NAT. Thirds, the function fib_validate_source of the source code net/ipv4/fib_frontend.c is modified to accept packets that comes from the same host.

## 4.5 Selective Replay Interface

The selective replay interface implements the mechanism as described in chapter 3. The user can indicate the event message from DUT and the replay log from replay engine so that SocketReplay knows how to replay selectively. At least an event

message contains a timestamp of triggered event. If it contains five-tuple information, SocketReplay starts from step 1. If it contains a source addresses and a destination address, SocketReplay starts from step 2. If it contains only the timestamp, SocketReplay starts from step 3.

# Chapter 5 Evaluation

In this chapter, we evaluated the capability of low-storage capture, loss-recovery and selective replay of SocketReplay in large-scale environment. The evaluations of low-storage capture focus on storing three types of network traffic: attack, virus, and P2P. The evaluations of loss-recovery and selective replay focus on the ability of recovering capture loss and traffic volume of selective replay.

## 5.1 Test environment for SocketReplay

In our evaluation test, we mirrored the network traffic of 1743 hosts as Fig. 1 shows to a capture device. Then we use SocketReplay to replay real flows to an IPS (intrusion prevention system) and collect its events from system logs.

Completeness is an important factor to replay on IPS accurately. We prove them by conducting two simple experiences on a complete connection that can trigger an event by TCPreplay. First, we removed 3-way handshake and replayed it again. We found it failed to trigger the event because IPS did not track this connection on its session table. Second, we removed a data packet after 3-way handshake and replay it again. We found it also failed to trigger the event because the sequence numbers of packets are not reasonable for IPS.

We sample 22185 connections from real flows within 30 seconds. Figure 8 illustrates the status of each connection. There are 10660 connections established before staring the capture so that we can’t capture the 3-way handshake, 7753 uni-direction connections because some of hosts sent SYN packet while the destination host did not reply or refuse the establishment, 3682 connections that has complete three-way handshake, 254 connections that have capture loss by inspecting

acknowledge numbers, and 2309 connections closed by a reset packet. From previous simple experience, if we use TCPreplay to replay all the traffic, the gray area of 10914 connections could not accurately replay on the IPS. However, SocketReplay can replay them accurately by loss-recovery mechanism.

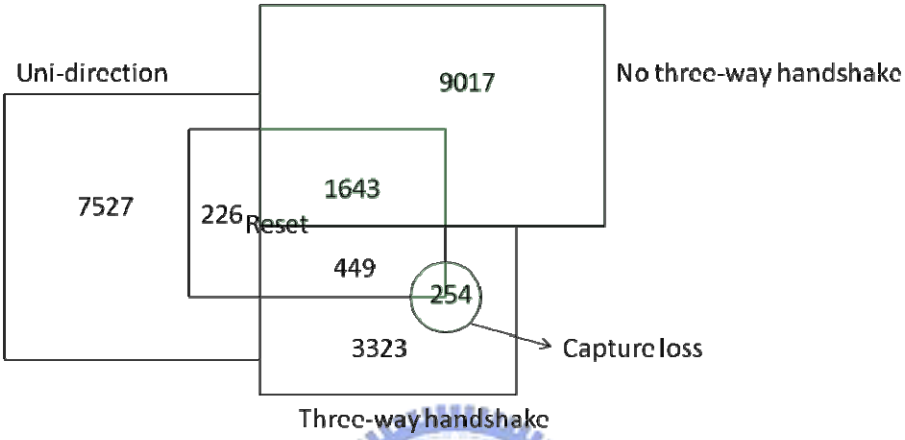



Figure 8. Status of connections in real flows

## 5.2 Test Results for SocketReplay

### 5.2.1. Attack

This work collects 1929 attack events from real flows triggered by an IPS and we try to reproduce these events. SocketReplay successfully reproduced events at the step 1 of selective replay, which is efficient to replay from large packet traces. Next we examined the effectiveness of loss-recovery by using TC (tldp.org/HOWTO/Traffic-Control-HOWTO/) to simulate the condition of capture loss. We compared SocketReplay and TCPreplay to see how the capture loss effects the event reproduction. Figure 9 shows that the proportion of triggered events of SocketReplay and the rate of capture loss are inversely proportional because some of packets with capture loss are critical to trigger the events. However, the proportion of triggered events of TCPreplay drop quickly when capture loss grows because a capture loss can affect the accuracy of whole replayed connections.

Next, we examined the percentage of capture loss in a commercial Network Monitoring System which uses libpcap to capture packets. We observed the number of packets dropped by kernel and received by filter after capturing 10 minutes of real flows and found that the percentage of capture loss is varied from 3% to 20% which is relative to the total packet numbers.

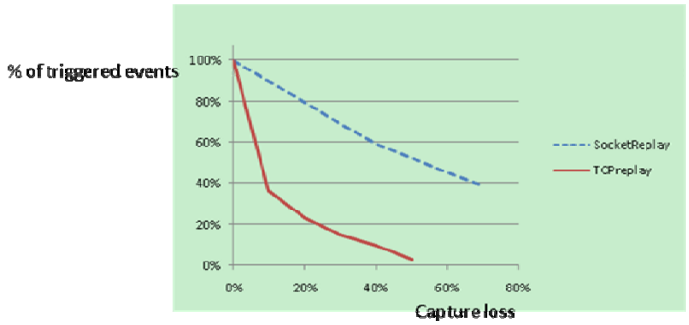



Figure 9. The effect of capture loss on event reproductions for SocketReplay and TCPreplay

### 5.2.2. FTP session

A FTP session contains a control connection and data connections. To reproduce an event of a data connection, replaying the data connection is not enough and it should include its control connection. Besides, in the active mode or the passive mode of FTP protocol, the control connection sent IP address and port number, which should be coherent with the data connection. In this work, we collect virus events that triggered by transmitting virus through FTP protocol to make sure that FTP sessions can be replayed accurately on IPS. SocketReplay successfully reproduced these virus events at the step 2 of selective replay, which is efficient to replay from large packet traces.

## 5.3 Test environment for low-storage capture

After using SocketReplay to replay captured traffic to an IPS (intrusion prevention system) and collect its events from system logs including attack and virus,

we include P2P events and use three thresholds (N, M, P) as Section 3.2 described to replay part of connection to figure out the best configuration of (N, M, P) that can produce more events and reduce more storage cost for these three types of events.

## 5.4 Test results for low-storage capture

### 5.4.1 Attack

This work collects 1929 attack events triggered by replaying real flows to an IPS and we can reproduce these events at the step 1 of selective replay. Table 2 lists top 10 types of attacks that cover 98.4% of 1929 attacks. The most frequent attack is Microsoft Windows RPC DCOM Service Buffer Overflow attack (www.cert.org/advisories/CA-2003-16.html) and its data length is 1828 bytes. Also, we found that only 333 connections' data length exceed 2000 bytes.

Table 2. Major types of attack events

| Event Count | Ratio (%) | Alert Message |
|---|---|---|
| 1493 | 75% | Microsoft Windows RPC DCOM Service buffer overflow attempt |
| 237 | 12% | Microsoft Windows LSASS buffer overflow attempt |
| 117 | 6% | FTP command overflow attempt |
| 18 | 0.9% | SQL Injection comment attempt |
| 11 | 0.5% | NETBIOS DCERPC NCACN-IP-TCP ISystemActivator RemoteCreateInstance little endian attempt |
| 7 | 0.4% | SHELLCODE x86 0x90 unicode NOOP |
| 6 | 0.3% | SQL sa brute force failed login unicode attempt |
| 4 | 0.2% | SQL SA brute force login attempt TDS v7/8 |
| 4 | 0.2% | Microsoft Windows MS08-067 attempt |
| 3 | 0.2% | FTP invalid MODE |

Therefore, we adjusted thresholds (N, 0, 0) which mean SocketReplay replays first N bytes data of connections and observe whether the intrusion prevention system can detect them as Fig. 10(a) shows. We found that 317 of 333 events were triggered

by simply replaying first 2K bytes of data per connection and a few events were triggered when we adjusted N from 5K to 50K. We pick 16 events that not triggered by using thresholds (2000, 0, 0) and increase the threshold N to see the percentage of triggered events and storage cost as Fig. 10(b) shows, we found that if we want to cover most of events, threshold N should be very high which cuase that percentage of storage cost is high. Therefore, we conclude that replaying 2K bytes of data of connections are enough to trigger most of attack events.

Next we picked four events that can't be triggered by replaying first 50K bytes of date and reproduce these events by adjusting thresholds (0, M, ∞) which means SocketReplay replays first M bytes data of each packet and found the minimum M as Table 3 shows. The first event is a false positive and the minimum M of last events is bounded by 200 bytes because these events were triggered from application headers. Therefore, we conduct another experience which uses thresholds (2000, M, ∞) to replay 16 events that can't be triggered by using thresholds (2000, 0, 0). As fig. 10(c) shows, we found that when M is 200 bytes, 11 of 16 events are triggered.

Next we adjusted the threshold P to find out the relation of storage cost and events that triggered by using thresholds (2000, 200, ∞). As fig. 10(d) shows, we found that when P is set to 1300, all 16 events are all triggered and 87% of storage is reduced. Also, when P is set to 200, 8 of 11 events are triggered and 90% of storage is reduced.

To sum up, besides the threshold 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% of events that can be triggered by SocketReplay and reduce 87% of storage cost. We set P to 1300 in order to trigger rare events that can't be trigger by using the thresholds (N, M, 200).

Table 3. Minimum M that triggers events that can't be triggered by two thresholds (50000, 0, 0)

| Alert Message | Total Payload Size | Minimum M |
|---|---|---|
| SHELLCODE x86 setgid 0 | 151611 | 1300 |
| SQL Injection comment attempt | 206085 | 140 |
| Web-CLIENT Windows Media Player zero length bitmap | 390745 | 200 |
| Adobe BMP Image Handler Buffer Overflow | 561305 | 90 |

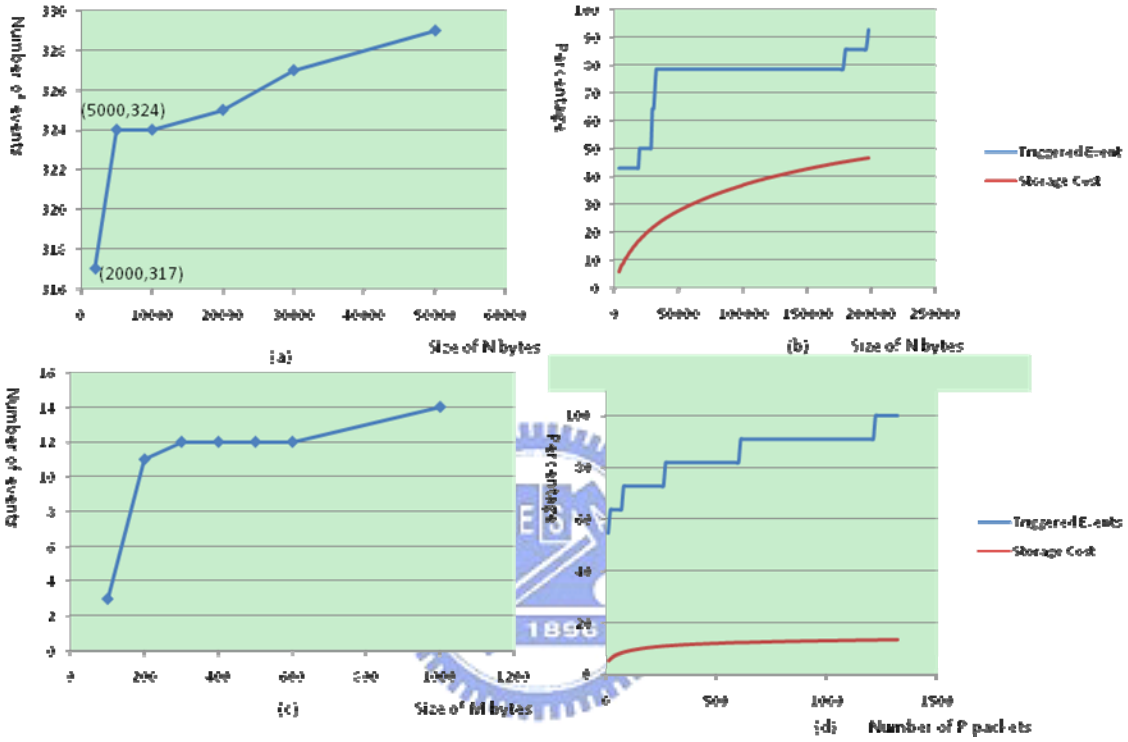


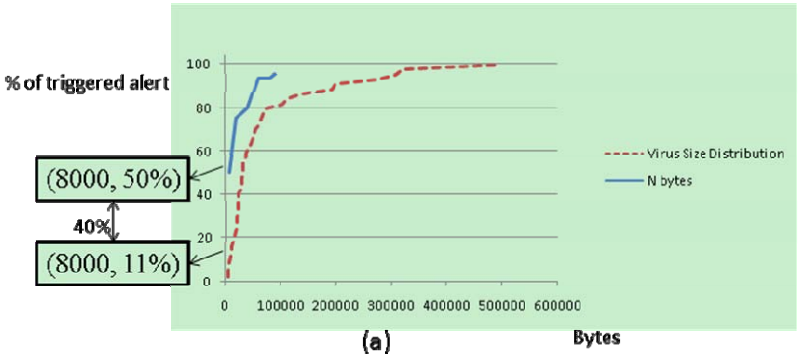

Figure 10. (a, c) Number of triggered events after setting the thresholds (N, 0, 0) and (N, M, ∞); (b, d) Percentage of Storage Cost and Triggered events after setting the thresholds (N, 0, 0) and (N, M, P)

### 5.4.2 Virus

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

The dotted line of Fig. 11(a) draws size distribution of viruses. This line shows that 20% of viruses are larger than 100000 bytes, which is absolutely larger than the payload length of attack traffic. The actual line of Fig. 11(a) and Fig. 11(b) draws the percentage of triggered events by using SocketReplay to replay 44 FTP sessions with thresholds (N, 0, 0). Although 40% of viruses' size is bigger than 8K bytes, these virus events can be triggered by replaying first 8000 bytes of each connection. We observe that replaying first 60000 bytes is enough to trigger 93% of virus events and reduce 70% of storage cost.

Besides threshold N, we conduct experience to answer whether threshold M is effective. Thethresholds (8000, 400, ∞) of replay can trigger 88% of events and reduce 67% of storage cost and another thresholds (8000, 1000, ∞) of reply can trigger 90% of events and reduce 35% of storage. Therefore, we found that the increase of threshold M is ineffective to trigger more events. Also, the benefit of two cases is not better than previous case with thresholds (60000, 0, 0) which trigger 93% of events and reduce 70% of storage. Therefore, we suggest setting threshold N to 60000 is enough to collect virus events of real flows and the thresholds M and P are set to zero.



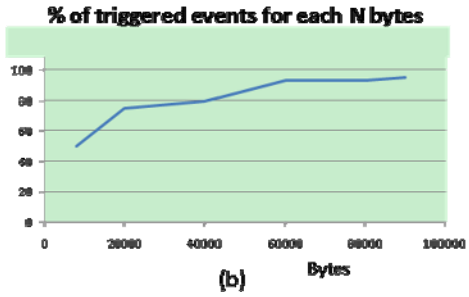



Figure 11. Virus Size Distribution and Thresholds (N, 0, 0) for reproduce % of virus events

### 5.4.3 Peer-to-Peer

The section focuses on how to collect minimum traffic to identify P2P events. Unlike attack and virus events, P2P applications use UDP protocol frequently to transfer unencrypted queries so that some of IPSs examines these queries and identify P2P applications such as BitTorrent, Skype and Edonkey. Furthermore, some techniques [12, 13] use statistical approach to identify P2P applications by TCP/IP headers. Therefore, the capture scheme simply needs to capture all UDP packets and headers of TCP packets.

# Chapter 6 Conclusion

The low-storage capture scheme and a replay tool SocketReplay provides a total solution on capture and replay in large-scale environment. The thresholds of capture scheme should adjust according to different traffic source, concerned traffic and types of devices that generate events. In campus-scale environment, the thresholds (2000, 200, 1300) are suitable for recording attack which triggers 99.74% of events and reduce 87% of storage, the thresholds (60000, 0, 0) are suitable for recording virus which triggers 93% of events and reduce 70% of storage. Also, capturing UDP packets and TCP headers are enough to trigger P2P events. By loss-recovery and stateful replay, SocketReplay can replay traffic from capture scheme and complete connections accurately that don't break protocol semantics regardless of capture loss. Furthermore, SocketReplay can reproduce the events efficiently by selective replay. This work provides an accurate and efficient way to play with real flows.

# Reference


[1] A. Turner, Tcpreplay, http://tcpreplay.synfin.net/trac/.

[2] W. Feng, A. Goel, A. Bezzaz, W. Feng, J. Walpole, "TCPivo: A High-Performance Packet Replay Engine," ACM SIGCOMM 2003 Workshop on Models, Methods, and Tools for Reproducible Network Research (MoMeTools), August 2003.

[3] Tomahawk, http://www.tomahawktesttool.org/, 2005.

[4] G. H. Hong and S. F. Wu., "On interactive internet traffic replay," in 8th Symposium on Recent Advanced Intrusion Detection (RAID), LNCS, Seattle, September 2005.

[5] Y.-C. Cheng, U. Holzle, N. Cardwell, S. Savage, and G. Voelker, "Monkey see, monkey do: A tool for tcp tracing and replaying," In Proceedings of the 2004 USENIX Annual Technical Conference, June 2004.

[6] A. Turner, "Flowreplay design notes," http://synfin.net/papers/flowreplay.pdf.

[7] Weidong Cui, Vern Paxson, Nick C. Weaver, and Randy H. Katz., "Protocol-Independent Adaptive Replay of Application Dialog," in Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS), Feb 2006.

[8] James Newsome, David Brumley, Jason Franklin, and Dawn Song, "Replayer: Automatic Protocol Replay by Binary Analysis," ACM Conference on Computer and Communications Security, October 2006.

[9] F. Schneider, J. Wallerich, A Feldmann, "Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware," Passive and Active Measurement Conference , April 2007.

[10] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann and R. Sommer, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," in Proceedings of ACM Internet Measurement Conference, October 2005.

[11] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson and F. Schneider, "Enriching Network Security Analysis with Time Travel," in Proceedings of ACM SIGCOMM, August 2008.

[12] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport Layer Identification of P2P Traffic", In Proceedings of the 4th ACM SIGCOMM conference on internet measurement 2004.

[13] L. Bernaille, R. Teixeira, I. Akodjenou, A. Soule, K. Salamatian, "Traffic Classification On The Fly", ACM SIGCOMM Computer Communication

Review 2006.