Netsniff – Passive Monitoring of Networked Applications

Jason But
jbut@swin.edu.au

Passive Monitoring

- Sniffing network traffic at a midpoint of a data flow
- Inferring behaviour
  - Network
  - Application
  - User
Passive Monitoring – Goals

- Acquiring useful data
  - Real World Traffic
  - Real Users

- Privacy a concern
  - Users identity
  - Internet Service Provider Identity
  - We are only interested in service type and network performance, not WHICH web site was visited or WHAT content was accessed

IP Address Anonymisation

- Variation on written description of algorithm applied by tcpdpriv
  - Basic Algorithm – For each new IP address seen:
    - Allocate an anonymised IP address such that if the first $n$ bits of the seen address ($IP_A$) match the first $n$ bits of a previously seen IP address ($IP_B$), then the new anonymised address has the first $n$ bits in common with the anonymised mapping of $IP_B$
    - Repeated monitoring of an IP address will map to the same anonymised address
    - Implemented with a sparse binary tree
IP Address Anonymisation

$P_A = 00110110$
IP Address Anonymisation

$IP_A = 00110110$

![Diagram of IP Address Anonymisation]

$IP_A = 00110110$

![Diagram of IP Address Anonymisation]
IP Address Anonymisation

$IP_A = 00110110$

![Diagram showing IP address anonymisation process]

http://www.caia.swin.edu.au
jbut@swin.edu.au
September 15, 2004
Page 9
IP Address Anonymisation

\[ \text{IP}_A = 00110110 \]

\[
\begin{array}{c}
\text{1} \quad \text{0} \\
\text{0} \quad \text{1} \\
\text{0} \quad \text{1} \\
\text{1} \quad \text{0}
\end{array}
\]
IP Address Anonymisation

IP_A = 00110110 ➔ 01010101

IP_B = 00110001

CAIA Seminar
IP Address Anonymisation

IP_A = 00110110 \rightarrow 01010101
IP_B = 00110001

IP_Address_Anomalisation
IP Address Anonymisation

$IP_A = 00110110 \rightarrow 01010101$

$IP_B = 00110001$

---

IP Address Anonymisation

$IP_A = 00110110 \rightarrow 01010101$

$IP_B = 00110001$
IP Address Anonymisation

IP_A = 00110110 01010101
IP_B = 00110001 01010001

TCP Stream Analysis

- Reconstruct TCP Streams
  - Out-of-Order Packets
  - Retransmitted Packets
  - Lost Packets

- Reconstruct Stream Payload

- Estimate Network Parameters
  - Server-to-Client Hops
  - Round Trip Time (RTT) and Jitter
  - Packet Loss Rate
**Hop Count Estimation**

- Based on IP Header TTL
  - Initial values of 64, 128 or 255 set at source
  - Estimate from listening point to each end-host
  - Average for each direction and sum

![Diagram of Hop Count Estimation](http://www.caia.swin.edu.au)

**RTT and Jitter Estimation**

- New RTT estimation algorithm
- Existing algorithms
- Single estimate made for entire stream
- Running estimate
- Generates an upper bound RTT estimate
RTT and Jitter Estimation

• New algorithm
  • Estimates two RTT values – one from host 1 to listening point, one from host 2 to listening point
  • Minimises measurement error
  • Introduces other errors
    • Lower accuracy for heavy uni-directional stream
  • Not better or worse – just different
  • Produces RTT estimate with lower bound

RTT and Jitter Estimation

• Estimates are smoothed using Jacobsons Smoothing Algorithm
  • RTT estimates track actual RTT
  • Jitter estimates are made from current packet RTT sample and current RTT estimate
  • Jitter is a running estimate of the Average Absolute Deviation of the RTT samples
RTT Estimation

- DummyNet box configured with a fixed network delay
- Delay changed about midway through a 10MB file transfer
- Packets caught and analysed on DummyNet box

RTT Estimation

- Replace DummyNet with a NISTNet Linux enabled router
  - Can configure mean and standard deviation delay
  - Correlated delays minimise amount of packet re-ordering
Jitter Estimation

- Estimates are lower than configured values
  - Configured value is $\sigma$
  - Estimated value is AAD

Packet Loss Rate Estimate

- Uses Benko-Veres Algorithm

- Bases estimate on retransmitted and out-of-order packets

- Accurate Packet Loss Rate estimation for streams > 2000 data packets

- Proposed running estimate (Armitage and Favi) not implemented in netsniff yet
Packet Loss Rate Estimation

- Estimate accurate regardless of jitter characteristics of link

- Accuracy dependent on:
  - Stream size
  - Actual PLR
- Best for streams under 3MB ≈ 2000 packets

HTTP Analysis

- Reconstructed TCP Payload is passed to higher layer modules for Application Layer Analysis
- Netsniff supports modular expansion to support different Application Layer Protocols
- Current implementation parses
  - DNS over UDP
  - HTTP over TCP
HTTP Analysis

- Parses HTTP Headers
  - HTTP Command type (GET, POST, etc.)
  - HTTP Response Codes
  - Content type (image/gif, text/html, etc.)
  - Content size
  - Server information (if not anonymised)
  - Cacheability

Netsniff – TCP Stream Analysis

- Algorithms implemented as described
- RTT and Jitter estimates are too numerous
  - These estimates are kept and sampled every 10 seconds
    - Produces one estimate every 10 seconds rather than one on every TCP ACK
    - Sampled RTT and Jitter estimates are also averaged for the duration of the stream
      - Mean RTT and Mean Jitter
  - PLR estimates are for the duration of the stream
Netsniff

- Netsniff produces a set of output files
  - notparsed.dump – tcpdump format file of all captured packets that are not DNS or HTTP traffic
  - dns.log – Text file containing a log of all DNS traffic
  - http.log – Text file containing a log of all HTTP transfers
  - tcpstream.log – Text file containing a log of all TCP stream information that is not HTTP traffic

Netsniff – notparsed.dump

- Formatted to be read later by tcpdump
- Only first 68 bytes of each packet is stored so payload is discarded – default tcpdump operation
- If netsniff is running in anonymised mode
  - IP addresses of all packets are anonymised
  - IP, TCP and UDP Checksums are modified such that they are correct for the new IP addresses encoded in the headers
Netsniff – http.log

As per tcpstream.log

HTTP Request
Type
Requested URL
Host

Referrer
Content Type
Content Size
Cacheable?

HTTP Return
Code

Some Lessons we Learnt

• FreeBSD + DummyNet cannot simulate Jitter
  • Can vary delay setting on DummyNet periodically
  • Fine for situations where packets are widely interspersed
• In situations such as TCP Stream
  • Packets tend to arrive in clumps
  • Each clump subjected to same delay
  • RTT estimate tracks clump RTT and has greater variation
  • Jitter estimate close to 0 – all packets in a clump have same delay and therefore no jitter
Some Lessons we Learnt

• Linux + NISTNet
  • Setting a delay of 0ms with a high data rate results in large numbers of packets being dropped by NISTNet – possibly due to buffer overflow
  • Jitter implemented with mean and standard deviation delay
    • Default configuration results in each packet independently delayed by a statistically determined time – large degree of packet re-ordering
    • Correlated configuration
      • Packets delayed by period similar to last delay – reduces but does not rule out packet re-ordering
      • Delay standard deviation MUST be less than 20% of configured delay

Some Lessons we Learnt

• Capturing packets under Linux
  • All packets captured with correct timestamp
  • Captured packets not processed in chronological order
    • Difficulties in inferring network behavior if packets are not in timestamp order
  • Problem with underlying Packet capture code on Linux
  • Known problem with nobody working on a solution