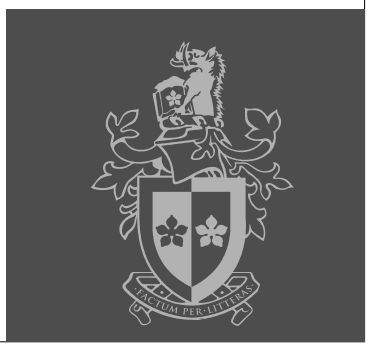
SWINBURNE

CENTRE FOR ADVANCED INTERNET ARCHITECTURES

# ICE³: Ongoing development and results around Netsniff

Urs Keller

urs.keller@epfl.ch



## Talk abstract

- In the last three months Netsniff has been extended with additional protocols and capabilities. The data it gathers can now conveniently be analyzed by querying a database built from Netsniff's log files. The talk will describe the ongoing development progress and some preliminary results.

# Talk Outline

- ICE³ / Netsniff overview
- My doings
- Some Results
- Questions



# ICE³ / Netsniff overview

- Goals of ICE³
- Netsniff summary
- Netsniff design decisions
- Some inner workings of Netsniff
- Output from Netsniff

# Goals of ICE$^3$

- The ICE3 deals with the following question: “What would happen if we inverted this capacity hierarchy?”
- And defines the following goals:
  - Evaluate the performance characteristics of some existing and emerging content distribution methods
  - Develop plausible, alternative IP network architectures ...
  - Evaluate the consumer's likely experience ...

SWINBURNE CENTRE FOR ADVANCED INTERNET ARCHITECTURES



# How Netsniff fits in

- Netsniff is a passive measurement tool written by Jason But.
- Netsniff is part of the first goal defined in ICE$^3$.
- It is currently used to observe traffic in a home user network.
- As of now it parses some common packet and “stream level” protocols.
- It logs characteristics of each protocol

SWINBURNE CENTRE FOR ADVANCED INTERNET ARCHITECTURES

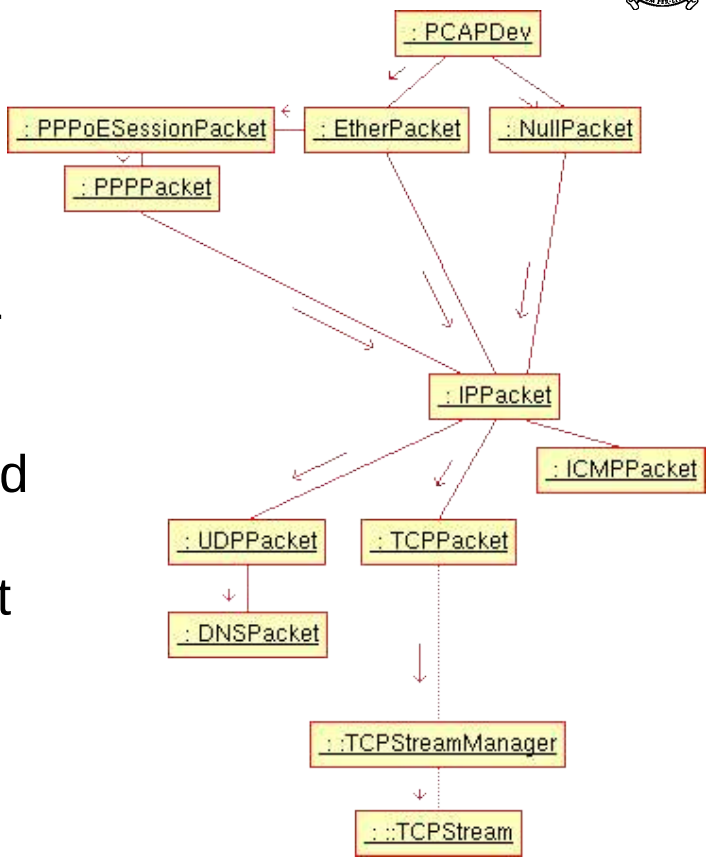
# Netsniff design decisions

- Uses the PCAP library to passively monitor traffic.
- Is written in C++/C
- Is single threaded
- For the protocols it can parse it produces log files in text format.
- Keep log files small (Upstream connection from DSL has low bandwidth)
- Assumed to sit at a midpoint of the network



# Some inner workings

- Internet Protocols are usually stacked.
- For the packet parser this results in packet objects creating other packet objects (right)
- TCP packets are added to a stream object by mapping them by port and IP pairs



# My doings (Outline)

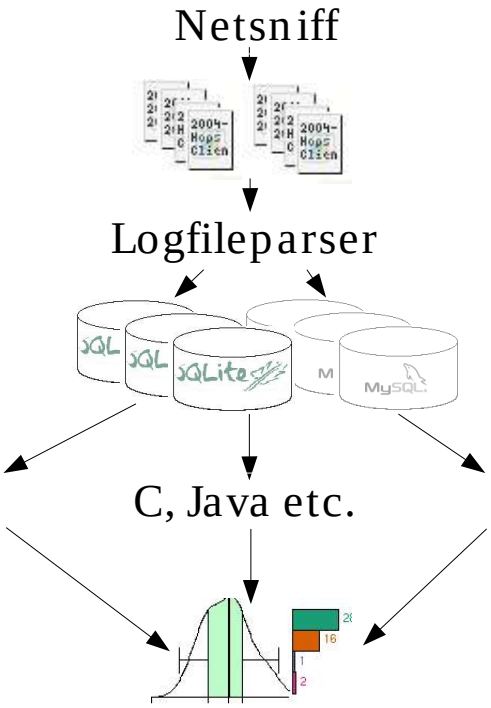
- Status when I joined CAIA
- Application level parsing
- Log files and database



# Status when I joined CAIA

- Netsniff designed and implemented.
- Very stable TCP parser, very few bugs
- Netsniff parsed protocols up to HTTP and DNS
- Good privacy mechanism for “user” privacy.
- Few protocol parsers
- No data analysis tools
- IP anonymization not consistent among “Netsniff sessions”

# Application level parsing




- Netsniff reassembles TCP streams from packets and passes the data streams up to an application parser.
- I built a couple of these: SMTP, POP3, IMAP4, FTP, TLS
- As you can see I spent a lot of time building protocol parsers for Netsniff.



# Parser generators?

- There are a lot of tools available to build parsers. The best known are probably Lex and Yacc.
- Generated parsers are in general very efficient.
- Parser is generated from grammar specification.

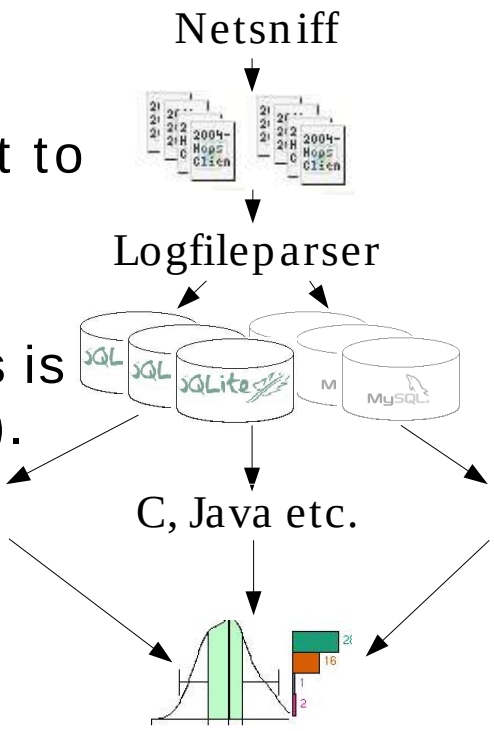
# Parser generators? (cont'd)

- Need to graft additional code in these specification.
- Hard to debug
- They are not really C++
- Make them reentrant is a pain
- Hard to make them recover
- Overkill: Many Internet protocols are regular languages.
- Adds another dependency to external tool.



# Interpret log files

- Netsniff generates log files in text format.
- The size of the log files is kept to a minimum.
- Text files can be compressed with a high ratio (not that this is better than optimal encoding).
- Only one disadvantage: We need to build tools to **parse** those files again.



# Requirements

- Adding future extensions to Netsniff only causes minimal additions in the log file parser.
- Different statistics can be easily done, once the data is parsed (statistical tests, plotting ...)
- Extendibility (data size)

➔ I took the decision to write the log file parser in C++ and store the data in an SQL database. The data then can be queried by the user's favorite tool.




# Tools used

- Sqlite as the SQL back end
- DBDesigner4 as the DB design tool. Schemata are stored in its format. SQL create and drop scripts can be exported. And ... it runs on FreeBSD. Very nice tool. Only drawback is that it uses the MySQL dialect.
- R (R project for statistical computing). It has a database driver, which supports Sqlite (version 2). This avoids some data import / export.

# What Output looks like

```
0xf785c1ab1c74c59e2f52b20d7765b526cb1bec380426d91c3c1ad675231e398b 0 0x301
0x4 0x0 9801 4774 14 9
2004-09-21 12:13:22.823435
2004-09-21 12:13:23.995159
2004-09-21 12:13:23.995020
...
```

```
tlssession_output              ::= tlssession_output_line1
                                   tlssession_output_lines
tlssession_output_line1        ::= tls_session_id SP
                                   tls_hadError SP
                                   tls_version SP
                                   tls_cipher SP
                                   tls_compression SP
                                   tls_payload_length SP
                                   tls_overhead SP
                                   tls_num_tcpstreams SP
                                   tls_session_duration CRLF
tlssession_output_lines        ::= 1*(tcp_timestamp CRLF)
```



# Output parsing

- Straight forward from the grammar specification.

```
tlssession_output_line1
::= tls_session_id SP
    tls_hadError SP
    tls_version SP
    tls_cipher SP
    tls_compression SP
    tls_payload_length SP
    tls_overhead SP
    tls_num_tcpstreams SP
    tls_session_duration
    CRLF
tlssession_output_lines
::= 1*(tcp_timestamp CRLF)
```

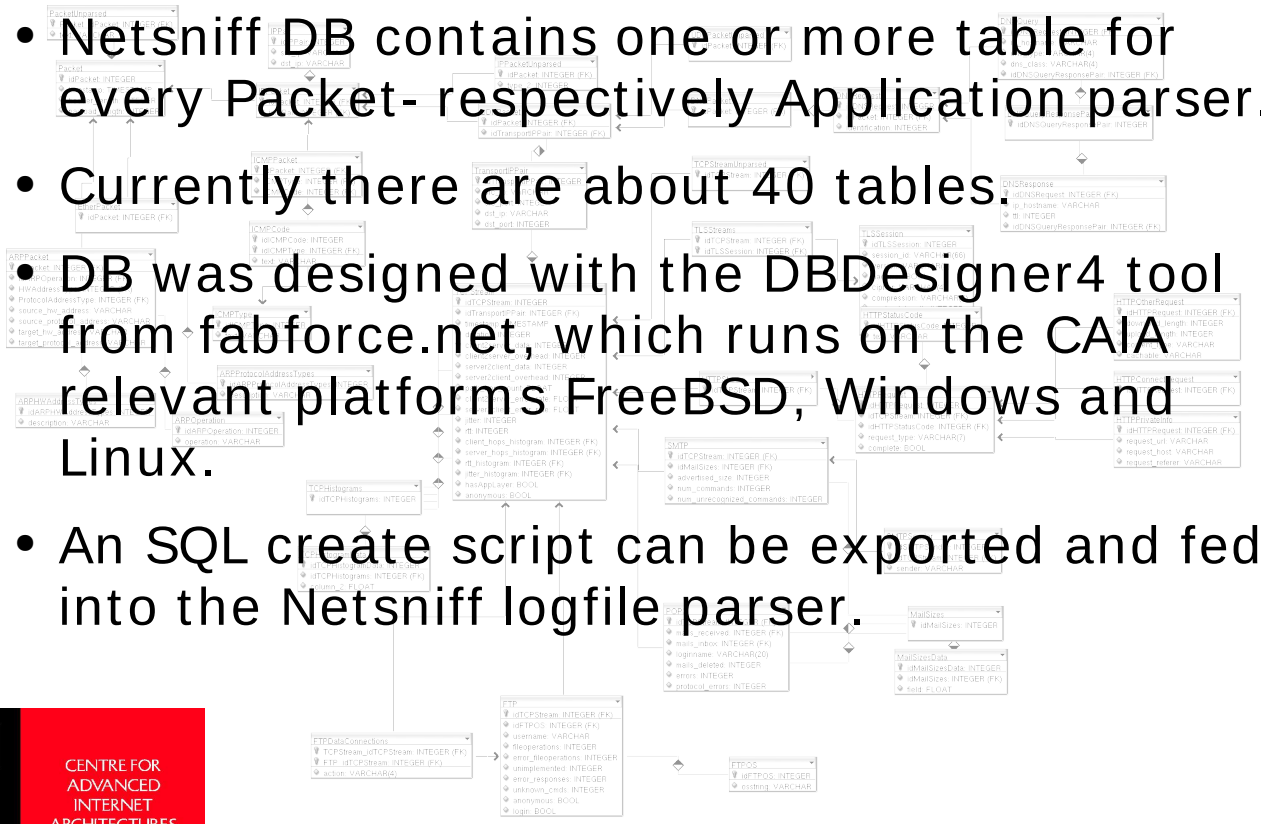
```
tlssession_output::tlssession_output(Parser& parser)
        : Parser(parser)
{
  tls_session_id = getString();       getSP();
  tls_hadError = (bool) getInteger(); getSP();
  tls_version = getString();          getSP();
  tls_cipher = getString();           getSP();
  tls_compression = getString();      getSP();
  tls_payload_length = getInteger();  getSP();
  tls_overhead = getInteger();        getSP();
  tls_num_tcpstreams = getInteger();  getSP();
  tls_session_duration = getInteger();
  skipLine();
  if (errors>0) return;

  for (unsigned i = 0; i < tls_num_tcpstreams; i++)
    {
      timestamps.push_back(getTimeStamp());
      skipLine();
    }

  skipLine();
}
```

# Netsniff DB

- Netsniff DB contains one or more table for every Packet- respectively Application parser.
- Currently there are about 40 tables.
- DB was designed with the DBDesigner4 tool from fabforce.net, which runs on the CAIA relevant platforms FreeBSD, Windows and Linux.
- An SQL create script can be exported and fed into the Netsniff logfile parser.

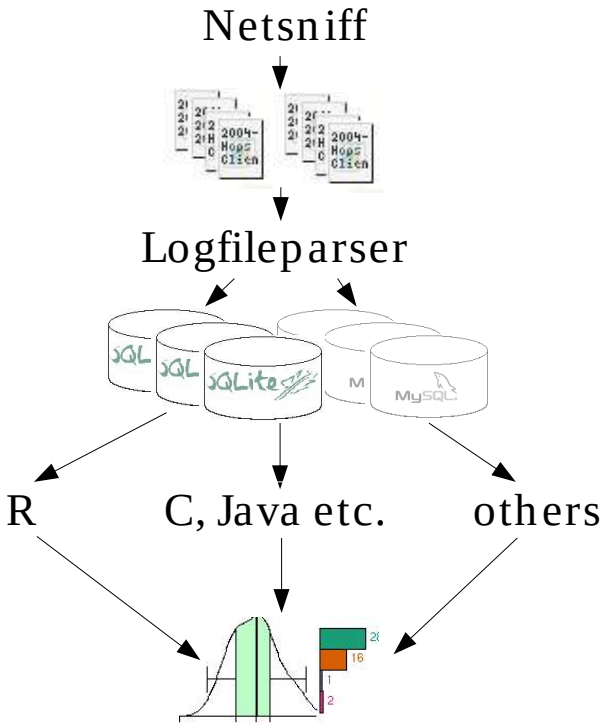


SWINBURNE CENTRE FOR ADVANCED INTERNET ARCHITECTURES



# Summary

- Netsniff captures traffic and logs to a series of log files
- Logfileparser parses the generated log files and writes them to a SQL DB.
- The DB can easily be queried by many tools and languages.




SWINBURNE CENTRE FOR ADVANCED INTERNET ARCHITECTURES

# Some results

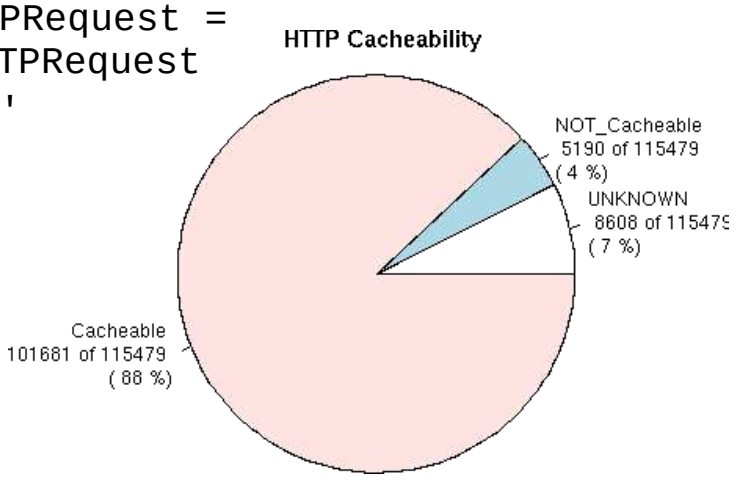
- Cacheability of GET requests
- Comparison to prior work
- User behavior over time (1)
- User behavior over time (2)
- DNS and HTTP correlated



# Cacheability of GET requests

- We want to know the percentage of cacheable GET requests of all GET requests completely captured:

```
select count(cachable), cachable
from HTTPRequest,HTTPOtherRequest
where HTTPRequest.idHTTPRequest =
  HTTPOtherRequest.idHTTPRequest
and request_type = 'GET'
and complete=1
group by cachable”
```



# Comparison to prior work

- Sebastian wrote a research paper in 2003 about cacheability. I tried to reproduce some of his results, with Netsniff. I will show some R code to give you an idea.
- We want to obtain a similar graph to the one below, which is 'stolen' from Sebastian's TR:

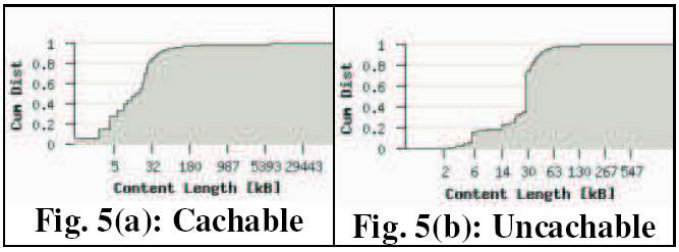



Fig. 5(a): Cachable Fig. 5(b): Uncachable



# Comparison to prior work cont'd

- The query part is straight forward:

```
(RSQLite)
# Database connection
DBString <- "/home/ice/logs/jaybee.homedns.org_2004-11-29.db";
con <- dbConnect(dbDriver("SQLite"),
                 dbname = DBString);

# Database query
qstring <- "select download_length \
from HTTPOtherRequest natural join HTTPRequest \
where complete = 1 and request_type = 'GET' \
and cachable = 'Cacheable' and download_length > 0";
q=dbGetQuery(con,qstring);

# Database disconnect
dbDisconnect(con);
```

# Comparison to prior work cont'd

- Plotting is a bit more complicated:

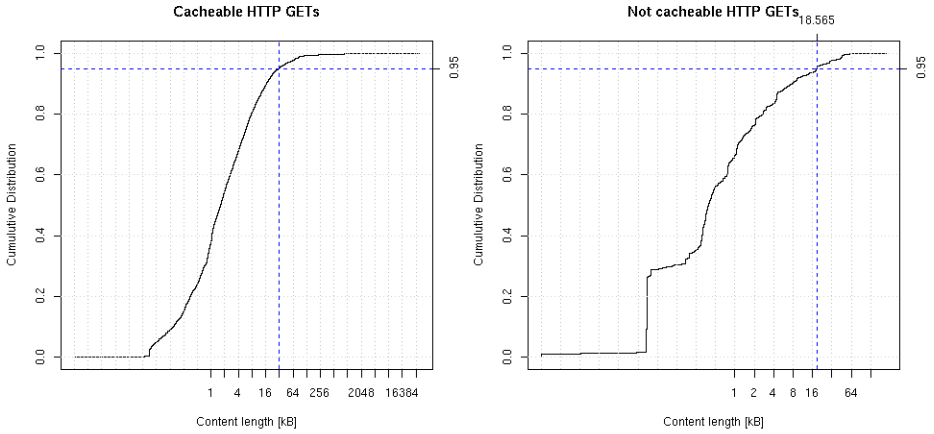
```
# Plot data
q_kb=q/1000;
pticks <- 2^(0:as.integer(log2(max(q_kb[[1]]))));
ticks <- 2^(as.integer(log2(1/1000)):
            as.integer(log2(max(q_kb[[1]]))));
plot(sort(q_kb[[1]]), (1:length(q_kb[[1]]))/length(q_kb[[1]]),
     log="x", type="s", xaxt="n",
     xlab="Content length [kB]", ylab="Cumulutive Distribution",
     main="Cacheable HTTP GETs")
axis(1,at=as.integer(pticks))
abline(v=ticks, lty="dotted",lwd=0.001, col="grey");
abline(v=1/1000, lty="dotted",lwd=0.001, col="grey");
grid(nx=0, ny=NULL,lty="dotted")
abline(h=0.95, lty="dashed",lwd=0.001, col="blue");
abline(v=32, lty="dashed",lwd=0.001, col="blue");
axis(2,at=0.95)
```

- The steps to produce the graph for non-cacheable objects are similar.



# Comparison to prior work cont'd

- The end results look very similar



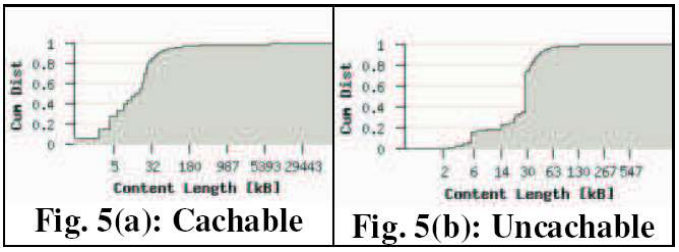
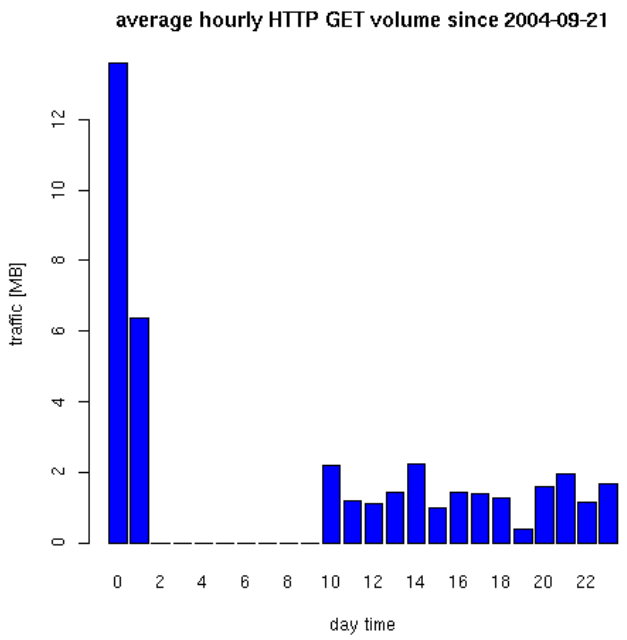




Fig. 5(a): Cachable Fig. 5(b): Uncachable

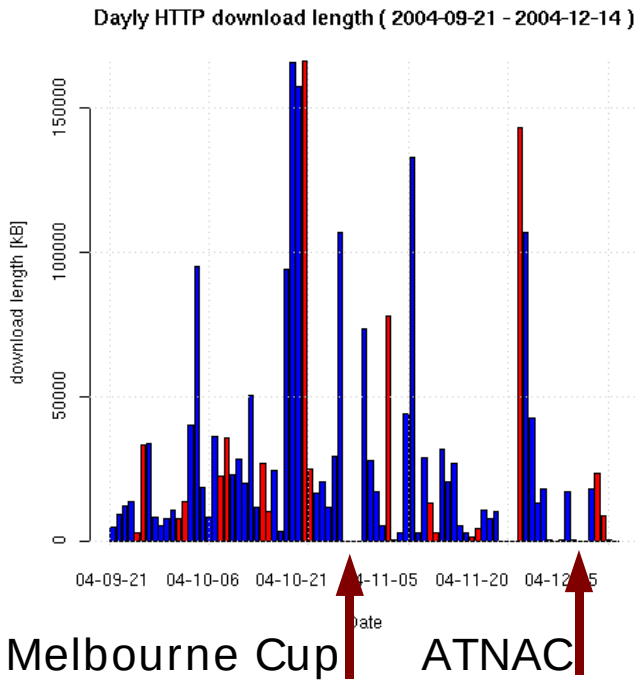
# User behavior over time (1)




- HTTP GET length averaged averaged over 24 hours
- Some sixty lines of R code
- Shows some user behavior

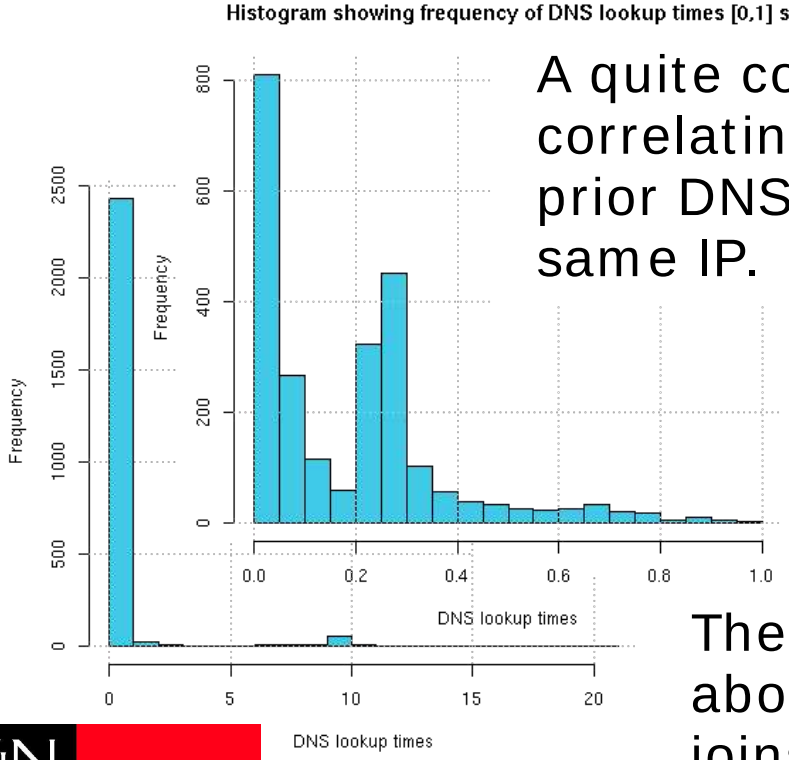


# User behavior over time (2)




- Admittedly not all statistics I have done so far, were that simple.
- The graph on the left is a bit more complicated.
- It takes four natural joins and about one page of R code to generate.

# DNS and HTTP correlated




A quite complex example is correlating HTTP requests to prior DNS requests with the same IP.

The resulting histogram show the distribution of lookup times.

The query needed uses about a dozen table joins and the R/SQL code is around 200 lines



# Where / What next?

- Anonymization
- Fixing some bugs
- Commenting code
- Polishing up
- Collect more data
- Analyze data we are collecting
- Some performance (disk size) problems

# Summary

- Overview of Netsniff
- Shown some tools / techniques used
- Hopefully demonstrated the power of Netsniff



# References


- DBDesigner4: http://www.fabforce.net/dbdesigner4/
- SQLite: http://www.sqlite.org/
- CryptoPaN: http://www.cc.gatech.edu/computing/Telecomm/cryptopan/
- S. Zander, G.J Armitage, C. Malcolm, "Dynamics and Cachability of WebSites: Implications for Inverted Capacity Networks," (pdf) CAIA Technical Report 030405B, April 2003

# Thank You.

Questions?

Comments?