# Design and Implementation of Netdude, a Framework for Packet Trace Manipulation

Christian Kreibich
*University of Cambridge Computer Laboratory*
*JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom*
christian.kreibich@cl.cam.ac.uk

## Abstract

We present a framework for inspection, visualization, and modification of `tcpdump` packet trace files. The system is modularized into components for distinct application purposes, readily extensible, and capable of handling trace files of arbitrary size and content. We include experiences of using the system in several real-world scenarios.

## 1 Introduction

In today's computer networks, traffic varies greatly in content and volume, making network analysis a difficult process. Researchers, developers, and system administrators use traffic capturing tools (*sniffers*) to obtain traces of network traffic to gain better understanding of traffic characteristics. Storing traffic flows in a standardized form allows them to investigate the effects of misconfigurations and programming errors, to process traffic using appropriate tool chains, and most importantly, to make the occurrence of observed phenomena *reproducible*.

Among the plethora of tools available for this purpose, three freely available ones constitute the defacto standard: the `libpcap` library[1] provides a low-level application programming interface (API) to filter and intercept packets, `tcpdump` presents these packets in textual format, and `ethereal`[2] provides a graphical user interface (GUI) for capturing, filtering, and inspecting packets, supporting a large number of networking protocols and sniffers.

Interestingly, tools that also allow the user to *edit* captured traffic have so far been limited to problem-specific solutions. The current state of the art is disappointing: developers create repositories of typically unreleased, purpose-specific, throw-away programs, inconveniently written at the `libpcap` level. Yet many of these tools would be useful to a larger audience. This practice violates a number of well-accepted software engineering principles, such as component reuse and the avoidance of cut-and-paste practices, code redundancy, and duplication of effort.

To improve this situation, we present *Netdude*, the *net*work *du*mp data *d*isplayer and *e*ditor, a framework designed to support different packet manipulation paradigms (from APIs to convenient GUIs), emphasizing code reuse, extensibility, and scalability. All components presented in this paper are fully implemented and publicly available.

The rest of this paper abstract is structured as follows: Section 2 presents the architecture of framework, including design goals and implementation details. Section 3 gives examples of using the framework, followed by a description of our experiences using the system in real-world scenarios in Section 4. Finally, Section 5 summarizes the paper.

## 2 Architecture

We first present our design goals for the framework and then describe how these goals lead us to the architecture we eventually implemented. In the remainder of this section, we illustrate how the framework components can be used in selected scenarios.

### 2.1 Design Goals

1. Multiple usage paradigms
   The user must be able to manipulate trace files at the desired level of interactivity and abstraction. We neither want to enforce only an API,

[1] `http://www.tcpdump.org`
[2] `http://www.ethereal.com`

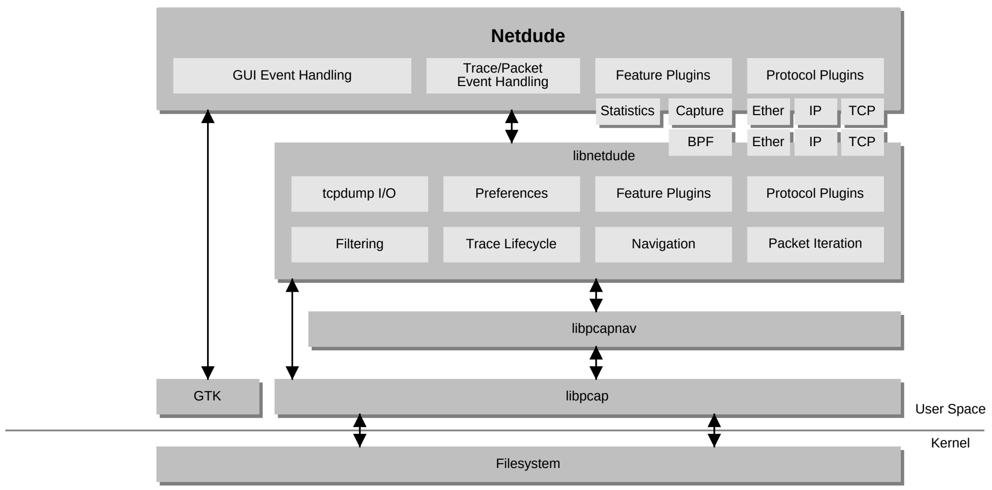


Figure 1: Architecture of the Netdude Framework.

thus asking all users of the framework to become developers, nor a GUI, forcing developers to use a graphical interface that may not be flexible enough. Programmers must find the framework usable at a convenient level of abstraction that allows them to focus on relevant aspects of their algorithms without getting distracted by details of packet reading & writing, trace file navigation, etc. The framework must eliminate the need to hand-write trace file access, filtering, iteration and protocol demultiplexing code anew for every application.

2. Openness & Extensibility
To programmers, we want to provide maximum flexibility in making their code interact with our framework. Since networking code is typically rather low-level, the programming language must not limit the usability of the framework to a certain language or execution environment. Both programmers and GUI users must have a means to extend the framework using components that they develop themselves or obtain from other developers.

3. Small-scale Editing
The framework must allow the manipulation of packets at a fine-grained level of detail, down to individual bits in the protocol headers and byte sequences in packet payloads. It also must provide the user with means to delete, move, swap, duplicate and erase packets, and to allow easy saving of changes made to a trace file.

4. Large-scale Editing
The framework must allow the manipulation of arbitrarily large trace files (subject to the maximum allowable file size on the operating system used), particularly files that are much larger than the system memory capacity. Traffic trace files easily reach sizes in the Gigabyte range, thus simply loading files into memory at startup is not an option.

The first goal excludes library-only or application-only designs since either would exclude one of the desired user groups. The second goal demands a widely used system programming language; we have decided to implement all library components in the C language to facilitate easy binding to other languages and to provide a largest-possible common denominator. The remaining two goals suggest concentrating the packet manipulation code in a library that can then be used by other programs.

## 2.2 Implementation

These goals lead to a layered architecture, illustrated in Figure 1. In the bottom layer, `libpcap` handles elementary trace file operations: opening and saving traces, sequential reading and writing of packets.

`libpcapnav` is a thin wrapper around `libpcap` that removes the limitations of sequential read access to packets stored in a trace. Between packet reads, users can jump to arbitrary locations in the trace file, identified by packet timestamps or fractional offsets in the file (e.g., 0.5 identifies the middle of the file). This library works similarly to `tcpslice`[3], but features a more robust algorithm for packet stream synchronization.

`libnetdude` is the core of the framework, where most of the packet editing functionality is implemented. It provides abstract data types and APIs for handling trace files, regions of trace files, packets, filters, and packet iterators. `libnetdude` can handle arbitrarily large traces: it never loads more than a configurable maximum number of packets into memory at any time. Since one of our design goals was the ability to perform arbitrary packet insertions and deletions, simple `mmap()`ing of regions of the trace file is not an option. Rather, trace files are edited at the granularity of *trace areas*, whose borders are defined using timestamps or fractional offsets understood by `libpcapnav`. The layers of modified trace areas are carefully maintained by `libnetdude`, always providing a consistent view of the trace file to the user. The modified trace areas are stored in temporary storage, as *trace parts*. Figure 2 illustrates these concepts.

When accessing a packet, the library always uses the trace part in the uppermost layer at the current offset. When a trace file is saved, the trace area layers are flattened onto the original trace file, honoring any inserted or removed packets, to yield the trace file the user created. The process is illustrated in Figure 3. Note that packet insertion and deletion are straightforward in this approach: the actual composition of packets in a trace area can change but trace parts are still aligned on their original boundaries.

`libnetdude` provides a number of other features:

- Its plugin architecture enables extensibility in two ways: *protocol plugins* allow interpretation of arbitrary protocol data, whereas *feature plugins* provide building blocks (e.g., anonymizers, statistical analyzers or flow demultiplexers) in a reusable fashion.
- Packet initialization turns raw packet data into structured protocol headers, as much as the installed protocol plugins permit. After that, it is easy to obtain, say, the TCP header of a packet. Nested protocols (such as IP tunnels) are supported. Developers thus need no longer write their own protocol demultiplexers in each application.

[3]`http://www.tcpdump.org/other/tcpslice.tar.Z`

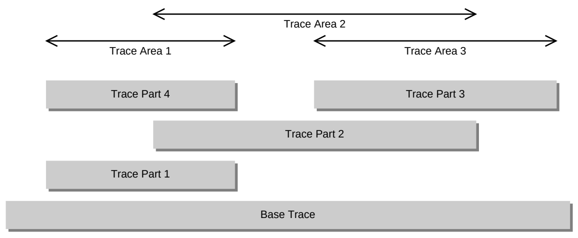


Figure 2: Editing different trace areas causes resulting trace parts to be layered on top of the original trace file.

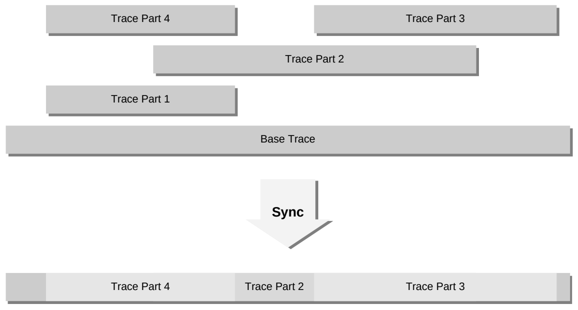
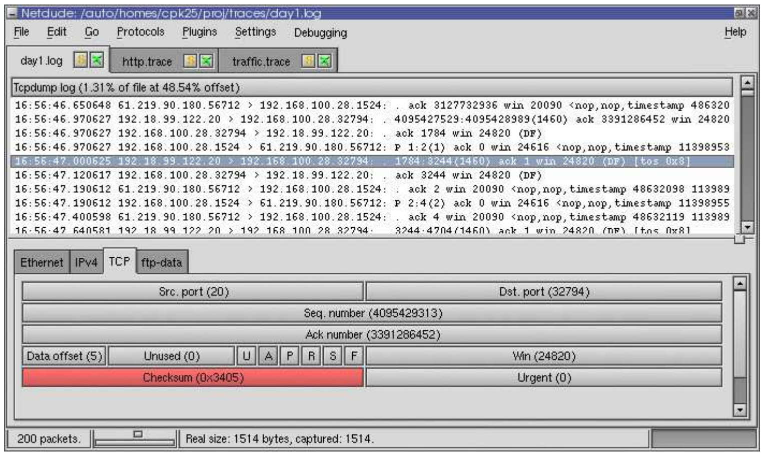


Figure 3: When saving a trace file, the layered parts are flattened onto the original trace file.

- Access to the familiar `tcpdump` output: `libnetdude` can associate each open trace file with its own `tcpdump` process. The user can then obtain `tcpdump` output at the granularity of individual packets with a single function call. As usual, details of the output can be controlled using `tcpdump`'s command line options. Since `libnetdude` can be configured to use any locally installed `tcpdump` binary, changes made to `tcpdump` remain visible inside the framework.
- An observer/observee API for objects like trace files, packets, packet iterators, and trace parts allows seamless integration of the library into the surrounding application, without exposing unnecessary internal state. Users can register callbacks that are invoked when certain events occur in the monitored items.

Finally, the Netdude framework provides a GUI application that uses all components described above. The main window is shown in Figure 4. The application allows non-developers to open and save trace files, jump to arbitrary locations in the trace files, modify protocol header fields and payload content in configurable trace areas, and access the installed feature plugins.



Tcpdump log (1.31% of file at 48.54% offset)

```
16:56:46.650648 61.219.90.180.56712 > 192.168.100.28.1524: . ack 3127732936 win 20090 <nop,nop,timestamp 486320
16:56:46.970627 192.18.99.122.20 > 192.168.100.28.32794: . 4095427529:4095428989(1460) ack 3391286452 win 24820
16:56:46.970627 192.168.100.28.32794 > 192.18.99.122.20: . ack 1784 win 24820 (DF)
16:56:46.970627 192.168.100.28.1524 > 61.219.90.180.56712: P 1:2(1) ack 0 win 24616 <nop,nop,timestamp 11398953
16:56:47.000625 192.18.99.122.20 > 192.168.100.28.32794: . 1784:3244(1460) ack 1 win 24820 (DF) [tos 0x8]
16:56:47.120617 192.168.100.28.32794 > 192.18.99.122.20: . ack 3244 win 24820 (DF)
16:56:47.190612 61.219.90.180.56712 > 192.168.100.28.1524: . ack 2 win 20090 <nop,nop,timestamp 48632098 113989
16:56:47.190612 192.168.100.28.1524 > 61.219.90.180.56712: P 2:4(2) ack 0 win 24616 <nop,nop,timestamp 11398955
16:56:47.400598 61.219.90.180.56712 > 192.168.100.28.1524: . ack 4 win 20090 <nop,nop,timestamp 48632119 113989
16:56:47.640581 192.18.99.122.20 > 192.168.100.28.32794: . 3244:4704(1460) ack 1 win 24820 (DF) [tos 0x8]
```

Src. port (20) | Dst. port (32794)
Seq. number (4095429313)
Ack number (3391286452)
Data offset (5) | Unused (0) | U | A | P | R | S | F | Win (24820)
Checksum (0x3405) | Urgent (0)

200 packets. | Real size: 1514 bytes, captured: 1514.

Figure 4: Main window of the Netdude GUI, with three trace files opened and the TCP header of the selected packet displayed. The red highlighting indicates that the TCP checksum in this packet is incorrect.

## 3 Framework Usage

We illustrate the usage of the framework in two examples: packet iteration and accessing selected protocol headers in a packet. Figure 5 shows `libnetdude` code for these scenarios.

### 3.1 Packet Iteration

Using `libnetdude`, packet iteration is done in two steps: first the area of the trace that the user wants to iterate is specified, then an iterator instance is used in a `for()`-loop. Using the GUI, the user defines the trace area conveniently using a dialog. The iteration is then done implicitly when the user performs an action that can be applied to multiple packets (e.g., setting a header field to a certain value, or fixing checksums).

### 3.2 Accessing Protocol Data

Using `libnetdude`, the user obtains a handle for the desired protocol by specifying the layer in the network stack and the identifier of the protocol commonly used at that layer (e.g., `IPPROTO_`xxx values at the network layer). Then the user requests a pointer to this protocol's header data in a packet at the desired nesting level. Using the GUI, the user first selects a packet from the list of packets currently loaded into memory. The GUI then provides access to the individual protocol headers contained in that packet. The user selects the desired protocol header and directly manipulates the header bit fields as visualized by the responsible plugin (e.g., using pull-down menus for fixed-range values, or entry fields for variable fields).

```
#include <libnd.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

void iterate_tcp_dports(const char *tracefile)
{
  LND_Trace           *trace;
  LND_PacketIterator  pit;
  LND_TraceArea        area;
  LND_Protocol        *tcp;
  struct tcphdr       *tcphdr;

  /* Obtain handle to TCP protocol */
  if (! (tcp = libnd_proto_registry_find(LND_PROTO_LAYER_TRANS, IPPROTO_TCP))) {
    /* Protocol not found -- handle accordingly. */
  }

  /* Open the trace file: */
  if (! (trace = libnd_trace_new(tracefile))) {
    /* Didn't work -- appropriate error handling. */
  }

  /* Set the trace's active area to the second half of the file. */
  libnd_trace_area_init_space(&area, 0.5, 1.0);
  libnd_trace_set_area(trace, &area);

  for (libnd_pit_init(&pit, trace); libnd_pit_get(&pit); libnd_pit_next(&pit)) {

    /* Request the TCP header of the current packet. */
    tcphdr = (struct tcphdr *) libnd_packet_get_data(libnd_pit_get(&pit), tcp, 0);
    if (tcphdr)
      printf("Dest. port: %u\n", ntohs(tcphdr->th_dport));
  }
}
```

Figure 5: A `libnetdude` example, iterating the second half of a trace and printing out the destination ports of all TCP packets in that area.

## 4 Real-world Use Cases

The original catalyst for the creation of Netdude was our work on TCP/IP network traffic normalization [HKP01]. This was a typical scenario for small-scale editing. In order to test our normalizations, we needed to create very specific packet constellations, for example specific values for the IP TTL field, the

TCP flag bits, and IP fragments with valid and invalid fragment offsets. Using the Netdude GUI, we gave individual packets the desired features and replayed the manipulated trace files through the normalizer.

The second use case was in the domain of high-speed network monitoring equipment. The subject of study was Nprobe, a scalable multi-protocol network monitor [MHK+03]. The goal was to evaluate system performance under various traffic loads. We used `libnetdude` to create traffic patterns that triggered different hotspots in the system. We then wrote an IP address mapping plugin for `libnetdude`, that maps those traces to disjunct IP address ranges so that we could replay multiple instances of the traces in parallel to expose the probe to high volumes of traffic.

At the moment we are using Netdude in order to test intrusion detection system (IDS) signatures. The classic approach is to experiment with a signature for a network-based IDS [Pax98][Roe99], testing whether the IDS reacts correctly when replaying a trace file. It is often more straightforward to manipulate the traffic itself and not the signature, particularly when testing the resilience of a new signature against variation in traffic patterns and corresponding false positive rates. Netdude's small-scale editing capabilities have proven most helpful in this scenario.

## 5 Summary

Netdude is a framework for `tcpdump` packet trace inspection, visualization and modification. Its modular design allows users to interact with the framework at different abstraction levels: a low-level trace navigation wrapper for `libpcap`, a high-level API with convenient types for performing common packet manipulation tasks in `libnetdude`, and a GUI application that allows both small- and large-scale editing previously impossible without writing code.

The system has been in development for three years now. The use cases that allowed us to apply the framework so far have confirmed our goals of simplifying the development of packet manipulation code and encouraging the re-use of components developed in other projects. We have implemented a number of plugins for purposes such as IP address translation, TCP flow demultiplexing, and statistical analysis. One of the main goals for future releases is a scripting interface to `libnetdude`.

We hope that the authors of networking code consider using the Netdude framework for their future packet manipulation needs, and provide useful functionality in the form of plugins for `libnetdude` or the Netdude GUI as a benefit to the community. Netdude is provided with a BSD license, hosted on SourceForge, and can be obtained at `http://netdude.sf.net`.

## Acknowledgments

We would like to thank Vern Paxson and Mark Handley for inspiration and helpful feedback. We also thank the Netdude user community for valuable ideas, comments, and contributions, particularly Daniel Stodden and Euan Harris, who also provided valuable comments on the paper. Thanks also to the Castle Pub in Cambridge for hosting our brainstorming sessions.

## References

[HKP01] Mark Handley, Christian Kreibich, and Vern Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, end End-to-End Protocol Semantics. In *Proceedings of the 9th USENIX Security Symposium*, August 2001.

[MHK+03] Andrew Moore, James Hall, Christian Kreibich, Euan Harris, and Ian Pratt. Architecture of a network monitor. In *Passive and Active Measurement Workshop Proceedings*, pages 77–86, La Jolla, California, April 2003.

[Pax98] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1998.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Conference on Systems Administration*, pages 229–238, 1999.