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

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

Abstract

We present the design and implementation of a framework for inspection, visualization, and modification of `tcpdump` packet trace files. The system is modularized into components for distinct application purposes, readily extensible, accessible through programmatic and graphical interfaces, 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 network misconfigurations and programming errors, to perform forensic investigation, 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, where the state of the art is disappointing: developers create repositories of frequently unreleased, purpose-specific, throw-away programs, inconveniently written at the `libpcap` level. Yet many of these tools would be useful to a larger audience. Publicly available tools typically have varying calling conventions, and while they normally are fairly easy to use in scripts, they are often not reusable at the API level because their functionality is only available in a standalone executable. These practices violate multiple 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. We think of Netdude as a *workbench* for the creation of new tools, integrating the efforts of other developers. All components presented in this paper are fully implemented and publicly available. We present the architecture of the framework, including design goals and implementation aspects, in Section 2. Section 3 gives usage examples at the API and GUI levels and demonstrates the extensibility of the framework. We describe our experiences in using the framework in a number of real-world scenarios in Section 4. Section 5 discusses the system and presents future work, before Section 6 summarizes the paper.

2 Architecture

We first state our design goals for the system. We then present the architecture of our framework, and walk through its components with a detailed explanation of the implementation.

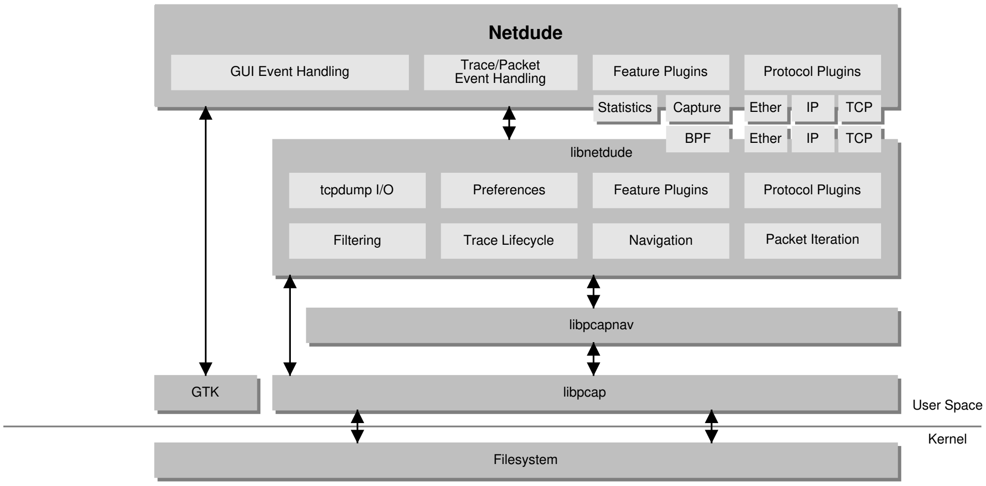

Figure 1: Architecture of the Netdude Framework.

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, 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
   Our goal is to provide programmers maximum flexibility in making their code interact with our framework. Since networking code is typically written in low-level languages, 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 the 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. The remainder of this section describes the higher layers of the architecture.

2.2.1 `libpcapnav`: Random Packet Access

`libpcapnav` is a thin wrapper around `libpcap` that removes the limitations of sequential read access to packets stored in a trace file. 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). After jumping to a random byte offset in a trace file, the challenge is to properly realign the packet extraction process to the packet sequence contained in the file. The `libpcap` file format currently provides no markers to identify the beginning of a packet in the file. Even if such a marker was used, it could always occur inside packet data as well. Therefore, a heuristic approach is called for. The algorithm used by `libpcapnav` is based on the one introduced by the `tcpslice`[3] tool. Our algorithm uses similar sanity checks on a number of `libpcap` packet header fields to identify possibly valid *packet chains*, but does not trust a chain of packets to be valid as easily. In cases like trace files containing a file transfer of another trace file (over NFS, via FTP, etc) the danger is to end up in a small chain of `libpcap` packet headers that actually comprise only the payload transported over the network. To avoid this, `libpcapnav` scans a window whose size is calculated from the maximum packet size captured in the trace and the maximum length of a chain of packets that the algorithm follows. While scanning, the algorithm keeps track of the chain lengths encountered, keeping only the longest chain.

2.2.2 `libnetdude`: Packet Manipulation

`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, packet iterators, and a few other features described below. `libnetdude` can handle trace files of up the maximum file size permitted by the file system: it never loads more than a configurable maximum number of packets into memory at any time. Just `mmap`ing regions of the trace file into memory is not an option, since our design goals include the ability to perform arbitrary packet insertions and deletions, and no data can be inserted in the middle of a `mmap`ed memory region. Rather, trace files are edited at the granularity of *trace areas*, whose borders are defined using timestamps or fractional

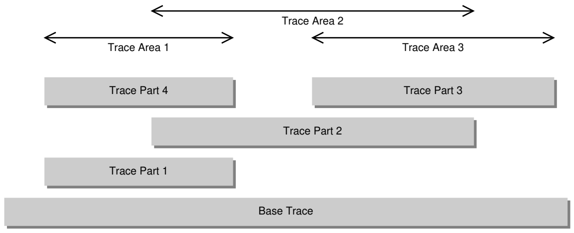

Figure 2: Editing different trace areas causes resulting trace parts to be layered on top of the original trace file.

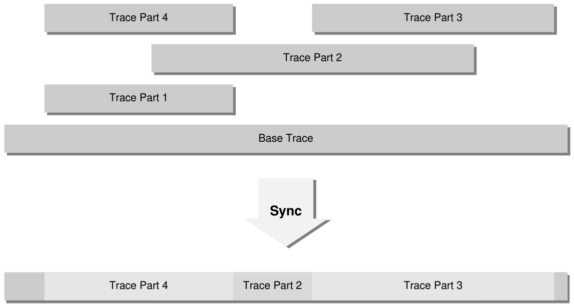

Figure 3: When saving a trace file, the layered parts are flattened onto the original trace file.

offsets understood by `libpcapnav`. The modified trace areas are stored in temporary storage as *trace parts*, and are logically layered on top of the original trace file, with new trace parts sitting on top of all the other trace parts in the trace area covered. Trace areas and trace parts are carefully maintained by `libnetdude`, always providing a consistent view of the trace file to the user. 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. The result is a new trace file that contains all modifications made to the original input file. The process is illustrated in Figure 3. The flattening process is performed implicitly through packet iteration: starting at the beginning of the base trace, iteration moves up to areas at higher layers as soon as they are encountered, and returns to lower layers at the end of each trace part.

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 merged onto lower parts at the original

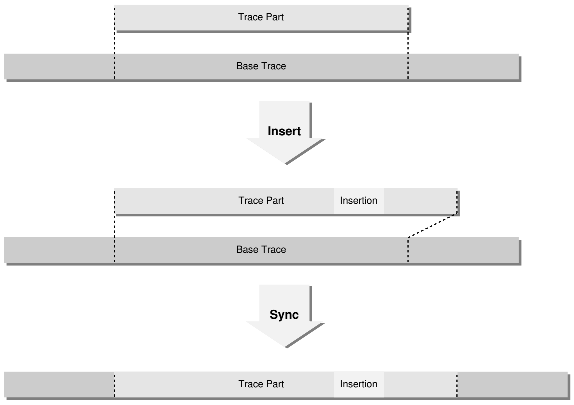

Figure 4: Packet insertion: a trace part is growing in size. When merged onto the base, the original boundaries of the modified trace part are maintained.

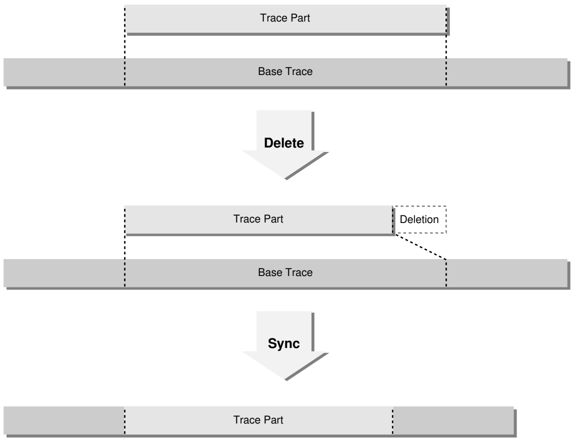

Figure 5: Packet deletion: a trace part is shrinking. Again, the original boundaries of the modified trace part are maintained when merging.

boundaries, regardless of the new higher part's size. This is illustrated in Figures 4 and 5. Furthermore, this approach makes it easy to provide "undo" functionality: removal of the most recent trace part from the stack reverts the most recent modification of the trace file.

In addition, `libnetdude` provides a number of other features:

- A plugin architecture that enables extensibility in two ways: Firstly, *protocol plugins* allow interpretation of arbitrary protocol data. Each protocol plugin provides the knowledge necessary to parse a protocol's header and to select the next protocol in the chain of protocol headers contained in a packet. Secondly, *feature plugins* provide reusable building blocks for functionality that the framework itself does not contain. By linking to other libraries, feature plugins can leverage any functionality accessible at the API level. `libnetdude` provides mechanisms for handling plugin dependencies by allowing plugins to check whether other required plugins are installed. Plugins are dynamically loaded and registered when `libnetdude` bootstraps, using `dlopen` and `dlsym` calls. This happens transparently: plugin authors only need to define a number of well-known functions to make their plugin's capabilities known.

- Structured packet content: once analyzed, packet contents are represented as a sequence of protocol headers. When a packet is initialized, `libnetdude` starts the packet analysis process by consulting the data link type given in the `libpcap` header of the trace file. The corresponding protocol plugin is queried, and control of the analysis is passed to that plugin and then onward as this plugin sees fit. The data structures representing the packet data are created on the fly. Analysis stops when no plugin for a given protocol can be found, or when a plugin does not need to pass analysis on to another protocol. Once the process is finished it is easy to obtain, say, the TCP header of a packet. Nested protocols (such as IP in IP) and arbitrary tunneling are supported. Developers thus need no longer write their own protocol demultiplexers for each application.

- Access to the familiar `tcpdump` output: `libnetdude` can associate each open trace file with its own `tcpdump` process through a bidirectional pipe. The user can then obtain `tcpdump` output at the granularity of individual packets with a single function call[4]. Full control over the output format is preserved by allowing configuration of `tcpdump`'s command line options. Since `libnetdude` can be configured to use any locally installed `tcpdump` executable, changes made to `tcpdump` remain visible inside the framework.

- An observer/observee API for objects like trace files, packets, packet iterators, packet filters, and trace parts. This feature 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

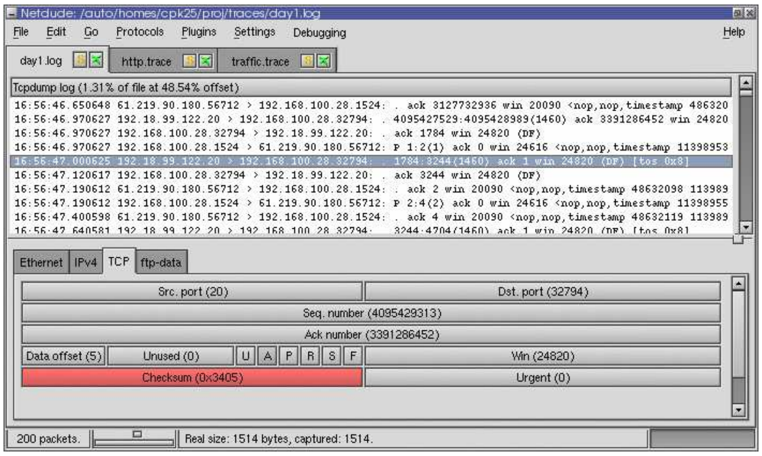

Figure 6: Main window of the Netdude GUI, with three trace files opened, 200 packets of the first file loaded into memory, and the TCP header of the selected packet displayed. The red highlight indicates that the TCP checksum in this packet is incorrect.

the monitored items, such as packet insertions and deletions, trace navigation, or advances in packet iteration.

2.2.3 Netdude: GUI Frontend

The Netdude framework provides a GUI application that leverages the functionality `libnetdude` provides. The main window is shown in Figure 6. The application provides graphical interfaces for all underlying abstractions: users can open and save trace files, navigate to arbitrary locations in the trace, inspect packets, configure trace areas to which packet modifications are applied, modify protocol header fields and payload content, and access add-on features through installed feature plugins.

The plugin concept of `libnetdude` is mirrored at the GUI level: protocol plugins allow the visual presentation of a particular protocol's header data, while feature plugins provide GUI access to underlying functionality. The visual representation of header data is entirely up to the plugin author: fixed-width cells can be rendered in a tabular layout, header fields can be color-coded depending on the field value, whereas string-based data may be better represented using list or tree elements.

In order to ensure good performance of trace file operations regardless of the file size, the application relies on `libnetdude`'s approach of limiting the number of packets loaded into memory to a configurable number. When the user jumps to a different location in the trace file, up to this number of packets are loaded into memory and presented as a list in the GUI. Individual packets are analyzed by selecting them from that list. The user can then browse the protocol data in a notebook containing one tab per protocol header contained in a packet. When no plugin can be found to visualize a protocol header, a fallback hex editor allows for inspection and modification of packet data using two different modes: a hexadecimal mode that presents each byte in ASCII and hex, and a pure ASCII representation suitable for text-based data.

3 Framework Usage

We illustrate the usage of the framework at the GUI and API levels using two examples: iterating over packets, and accessing selected protocol headers in a packet. Figure 7 shows `libnetdude` code for these scenarios. To illustrate the flexibility of the plugin mechanism, we then present a few feature plugins for `libnetdude`.

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, a packet iterator instance is used in a `for`-loop. In each iteration, the current packet can be obtained from the packet iterator. `libnetdude` differentiates between read-only

```c
#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 a handle to the 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);

  /* Iterate over all packets in that 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 a TCP header was found, print its destination port. */
    if (tcphdr)
      printf("Dest. port: %u\n", ntohs(tcphdr->th_dport));
  }
}
```

Figure 7: A `libnetdude` example, iterating the second half of a trace and printing out the destination ports of all TCP packets in that area.

and read/write iteration because packet modifications require the creation of a new trace part for the trace area iterated. In this case, the user can selectively drop packets during the iteration.

Using the GUI, the user first defines the trace area using a dialog. The iteration is then performed implicitly when the user modifies a packet (for example by setting a header field to a certain value, or fixing checksums): the same modification is applied to all packets in the configured trace area, subject to configured packet filters.

3.2 Accessing Protocol Data

Using `libnetdude`, the user obtains a handle for the desired protocol by specifying the protocol's 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). The user then requests a pointer to this protocol's header data in a packet, for the desired nesting level. Note that this querying mechanism does not imply that the protocol must be used at the specified protocol layer in the packet data. The layer is only used to obtain a handle to the data structure representing the protocol in `libnetdude`.

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's bit fields as visualized by the responsible plugin (e.g., using pull-down menus for fixed-range values, or entry fields for variable fields).

3.3 `libnetdude` Feature Plugins

When making new features available at the API and GUI levels, we prefer to first provide functionality in `libnetdude` feature plugins and then add Netdude GUI frontends later on. `libnetdude`'s plugin-based architecture has important implications for developers: consider a programmer who wants to develop an application that uses functionality provided by a feature plugin. When using C, he or she would typically use the plugin's API by including the plugin's header file(s). However, this causes problems at link time due to the late-binding design of `libnetdude`'s plugins: when linking the new application's code to `libnetdude`, the symbol definitions of the required plugins are not available since the object files are only linked in after the library's bootstrapping process completes. The result is an undefined symbol error. Adding the plugin's shared object files at link time is insufficient in case the plugin itself requires other plugins; this approach would quickly result in *all* plugin object files being added at link time, violating our design goal of flexible extensibility.

As a more scalable solution, we ask plugin developers to provide the new functionality in `libnetdude` plugins themselves. This way, symbols are only resolved at runtime (since the plugins are built as shared objects) and linking can remain constrained to the new plugin. Undefined symbols can still be caught at compile time using suitable compiler options, such as `-Werror-implicit-function-declaration` when using GCC.

To make the plugin's functionality accessible, developers have two options: the first is to provide their own executable that initializes the library, queries the plugin, and runs it with appropriate parameters. This only needs a few lines of code. The second option is to use the `lndtool` command line frontend that is provided by `libnetdude`. This tool can be used to query several parameters of the local `libnetdude` installation. As an example, Figure 8 shows how `lndtool` lists installed plugins.

```
cpk25@ghouls:/auto/homes/cpk25 > lndtool --plugins
libnetdude protocol plugins:
--------------------------------------------------
Ethernet                         0.5
ICMP                             0.5
IPv4                             0.5
SLL                              0.5
LLC/SNAP                         0.5
TCP                              0.5
UDP                              0.5
ARP                              0.5
FDDI                             0.5

libnetdude feature plugins:
--------------------------------------------------
BPF-Filter                       0.5
Checksum-Fix                     0.5
PHDL                             0.1
TCP-Filter                       0.1
TCP-State-Tracker                0.2
Trace-Set                        0.1
Traffic-Analyzer                 0.3
```

Figure 8: Running `lndtool` to obtain a list of installed plugins.

`lndtool` also provides a command line interface for accessing the plugins, passing command line arguments through to the selected plugin. Examples of `lndtool`'s usage are given in the following selection from feature plugins that have been developed so far:

- Trace Sets: The need to operate on a set of traces occurs frequently. To allow easy reuse of this functionality, we have developed a plugin that manages the life cycle of sets of trace files and provides a mechanism to iterate over trace files contained in such a set.
- TCP Connection Tracking: Most TCP-based packet manipulation requires TCP connection state tracking. This plugin provides such functionality and allows the user to maintain and query connection state for a number of flows, and check whether the three-way handshake or connection teardown were fully observed. The plugin is useful in itself: when run as `lndtool -r tcp-state-tracker <trace>`, it prints the familiar `tcpdump` output but augments each TCP packet's line by including the current connection state.
- Filtering incomplete TCP flows: This plugins scans a set of trace files and removes all incomplete TCP flows present. In a first scan, the connection tracking plugin is used to establish and update state for each flow found in the trace, before a second scan checks each packet's flow for complete three-way handshake and connection teardown. If those were observed, the packet is kept, otherwise it is dropped. The plugin can be accessed from the command line using `lndtool -r tcp-filter <trace1> [<trace2> <...>]`.
- Traffic Statistics: To quickly get an idea of what is contained in a trace file, we have developed a simple traffic analyzer plugin that computes counters and percentage values for the number of packets and bytes contained, IP payload protocol usage, TCP/UDP port number usage, and TCP flows. The plugin can be accessed from the command line using `lndtool -r traffic-analyzer <trace1> [<trace2> <...>]`.
- Abstract Protocol Header Definition: `libnetdude` and Netdude allow the developer to make protocol analyzers arbitrarily smart and to design the visual representation of protocol header data in any way desired. For example, the standard IP plugin is able to check whether the IP header checksum is correct, and can also fragment and reassemble packets. Frequently however, sophisticated functionality is less important than basic understanding of the protocol structure. In this situation, it is not necessary to force developers to write their own code to make a protocol's structure accessible. Rather, a high-level *protocol header definition language* is desirable that allows the specification of a protocol header's layout in a simple text file.

  We have designed a language, PHDL, for this purpose and provide an interpreter as a separate `libnetdude` plugin. When `libnetdude` is initialized, the PHDL plugin reads all installed protocol definition files and creates protocol data structures accordingly, equipping each new protocol with a header blueprint. When a packet's protocols are analyzed, the header blueprint is used to build an instance of the structured protocol data that can then be queried and manipulated. The complementing PHDL Netdude plugin then uses this information to visualize the protocol data in a standardized tree view similar to `ethereal`, but with the added functionality of being able to modify the header fields.

  As an example, we show a PHDL definition for IPv4 in Figure 9.

```
# PHDL Definition for IPv4 based on RFC 791.

# structure of the header common to many IP options.
def "opthdr" {
        unsigned int "type" 8;
        unsigned int "length" 8;
}

# structure of an IPv4 address -- 32bit field, when output,
# chunk into 8bit units and separate using a ".".
def "ip4addr" {
        int "addr" 32 { unit = 8; sep = "."; }
}

# structure of IP options that contain a list of IPv4 addresses.
def "addropt" {
        opthdr "header";
        unsigned int "ptr" 8;
        chain "route" {
                ip4addr "addr";
        } until length ((.header.length - 3) * 8);
}

# Now the main header definition:
proto "IPv4" (net : 0x800) {

        block "fixed" {
                int "version" 4;
                int "hl" 4 { scale = 4; }

                block "tos" {
                        enum "ecn" 2 {
                                0 : "-";                1 : "ECT(0)2";
                                2 : "ECT(1)";           3 : "CE";
                        }
                        enum "tos" 4{
                                0x10 : "Low Delay"; 0x08 : "Reliability";
                                0x04 : "Low Cost";  0x00 : "None";
                        }
                }

                unsigned int "len" 16;
                unsigned int "id"  16;

                block "frag" {
                        int "rf" 1; int "df" 1; int "mf" 1;
                        unsigned int "off" 13;
                }

                int "ttl" 8;

                # The exclamation mark identifies this field as the
                # key to the selection of the next protocol header:
              ! enum "proto" 8 {
                        1       : "ICMP";
                        2       : "IGMP";
                        4       : "IPIP";
                        6       : "TCP";
                        # rest treated as "other"
                }

                hex "checksum" 16;
                ip4addr "src";
                ip4addr "dst";
        }

        chain "options" {
                union "option" {
                        int "noop" 8 if (.noop == 0);

                        block "security" {
                                opthdr "header";
                                unsigned int "s"   16;
                                unsigned int "c"   16;
                                unsigned int "h"   16;
                                unsigned int "tcc" 16;
                        } if (.header.type == 130);

                        addropt "lsrr" if (.header.type == 131);
                        addropt "ssrr" if (.header.type == 137);
                        addropt "rr"   if (.header.type == 7);

                        block "streamid" {
                                opthdr "header";
                                unsigned int "id" 16;
                        } if (.header.type == 136);

                        block "timestamp" {
                                opthdr "header";
                                unsigned int "ptr" 8;
                                unsigned int "oflw" 4;
                                unsigned int "flag" 4;
                                ip4addr "addr";
                                chain "ts" {
                                        unsigned int "tstamp" 32;
                                } until length ((.header.length - 3) * 8);
                        } if (.header.type == 68);
                }
        } until length (fixed.hl - 5) * 4 * 8;
}
```

Figure 9: PHDL code describing the IPv4 header layout.

Other potential applications include traffic anonymizers, address mappers, import and export filters for other file formats, and interfaces to other software for more advanced functionality like visualization and mathematical analysis.

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. This approach was particularly useful in our work on the Honeycomb IDS signature generator[Kre03], where the ability to make small-scale modifications to packet data was most helpful for testing the string-matching algorithm used by the system. Netdude's editing capabilities have worked very well in these scenarios.

5 Discussion & Future Work

In its current state, we find the framework useful for everyday work with sets of trace files of sizes ranging from a few kilobytes up to several gigabytes. The ability to access functionality through a command line interface is most valuable for scripting tasks for repeated execution. We mostly use the GUI application for the simpler editing tasks and for quick inspection of trace files. Netdude's ability to handle large trace files makes it a far better option than alternative tools like ethereal that lack this feature and that are restricted to files smaller than the system's physical memory capacity.

When using the command line, we have frequently found that it would be useful to be able to use the traditional UNIX approach of piping the output from one processing stage to the input of the next stage. Unfortunately this metaphor is not directly applicable to our problem setting: depending on the functionality provided, a stage may have to employ random access to various locations in the file, or scan a file repeatedly (as in the case of the TCP filtering plugin described in Section 3.3). Directly piping packet data from one stage into the next will not work here since the streams cannot be rewound. However, temporary files could be used transparently, and the piping could be kept within the lndtool command, for example using a syntax like lndtool '-r <stage1> -i <input> | -r <stage 2> | ... | -r <stage n> -o <output>'

Another useful feature would be the ability to use libnetdude in a scripting environment such as Python or Perl. Creating the necessary "glue" code using a tool like SWIG[5] should prove fairly easy and is one of our next items for future work.

6 Summary

Netdude is a framework for inspection, visualization, and modification of tcpdump packet trace files. Its modular design allows users to interact with the framework at different abstraction levels: a low-level trace navigation wrapper for libpcap called libpcapnav, 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 framework is readily extensible at the libnetdude and GUI levels through its plugin architecture, making it a workbench for the creation of new packet trace tools. A number of plugins have been developed so far and have already helped us in cutting down the development time for new features.

The system has been in development for three years. 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.

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

Since October 2002, this work has been carried out in collaboration with Intel Research, Cambridge. We would like to thank Vern Paxson, Mark Handley, and Jon Crowcroft for inspiration and helpful feedback. We also thank the Netdude user community for valuable ideas, comments, and contributions, particularly Andrew Moore, 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.

Notes

1. See http://www.tcpdump.org
2. See http://www.ethereal.com
3. See ftp://ftp.ee.lbl.gov/tcpslice.tar.Z
4. The implementation of this feature is significantly complicated by the fact that tcpdump's packet analyzer is currently not available as a library.
5. See http://www.swig.org

References

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

[Kre03] Christian Kreibich. Honeycomb - Automated NIDS Signature Generation using Honeypots, Poster Paper. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003. SIGCOMM.

[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.