Parsing Streaming Network Protocols

By
Evan Hughes

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario, Canada

2006

Abstract

Analysing captured Internet data is difficult. If one wishes to explore the content of application streams, one must defragment IP datagrams, and then reconstruct TCP streams, before parsing the application stream itself. Surprisingly, there are no tools in existence the provide a programmatic interface for this task, while existing tools are difficult to extend.

This thesis focuses on the hardest part of stream reconstructing: parsing application streams. RFC-specified application streams often contain constructs that are difficult to handle using standard parsing methods. We propose a machine-readable grammar for specifying stream protocols, and develop a parser generator to build parsers for network monitors. Our grammar is compatible with the grammar used to describe existing protocols, meaning that minimal work is required to convert existing specifications to our new format. In doing so, we propose that network analysis climb the protocol stack, from the physical and transport layers where it is today, to the application layer.

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Parsing Streaming Network Protocols

submitted by

Evan Hughes

Dr. Frank Dehne
(Director, School of Computer Science)

Dr. Anil Somayaji
(Thesis Supervisor)

Carleton University
2006

Contents

List of Figures

Chapter 1

Introduction

The Internet is a free and open place. Excluding the occasional repressive government and locked down workplace, anyone is free to send packets to anyone else. Those messages can be in any format, and may contain an arbitrary payload. Of course, if the sender wants the receiver to do something with the message, the receiver must have a rough idea of what to do with the packet. When both the sender and the receiver know how to handle a packet, we say that they have a protocol.

The freedom of the Internet is a boon to most, but a headache for a few. The freedom is a boon because protocol designers and application writers can introduce new applications (and their attendant protocols) easily, which means that it's easy for users to get their hands on new applications. The headaches are reserved for those of us monitoring the Internet: every new protocol is a new language that we have to learn in order to be able to understand what is happening on our networks.

When we understand the traffic on a network, we have *network awareness*[1][30]. We are able to accurately answer the questions "Who is using the network? Why?

[1] "Network awareness" is adapted from the military term *situational awareness*[27] which has a similar definition, although it is generalised beyond the network context to the operational military context.

And how?" Network awareness requires familiarity with the network in question. Which brings us to the crux of the problem: we have no tools that reveal the full spectrum of activity occurring on computer networks.

1.1 Reasons to Monitor Network Traffic

Reasons for monitoring network traffic vary, depending on the individuals involved and their goals. Those who create and maintain networks have different concerns from those who sell network connectivity; just as software and protocol developers have their own concerns.

Operational reasons to monitor network traffic. From a day-to-day perspective, network administrators and operators want assurance that the equipment they have deployed is behaving properly. They wish to know when the use of the network changes, requiring new hardware, or configuration changes on existing hardware. Although this information may be available from the hardware products themselves, network monitoring software provides flexibility to the network administrator and the opportunity to verify that the hardware is reporting properly. Administrators and operators also want to ensure that their networks are not being used for nefarious purposes. Network monitoring provides a window onto network use, possibly revealing malicious outsiders who have exploited one or more hosts on the network, or authorised insiders using the network for unauthorised purposes.

Financial and legal reasons to monitor network traffic. Aside from provisioning, network debugging, and security concerns, system administrators must also fulfil financial and legal obligations. If access to network is being sold, the users

of the network may be billed for the bandwidth they consume or the volume of data that they exchange. Administrators and their employers need to monitor the network to calculate charges to be sent to customers. Meanwhile, the users will want to know what they are being billed for, forcing the sellers to provide some indication of the usage patterns of the network.

Network monitoring also serves a legal purpose, as legislation may require network administrators to verify that sensitive information is not carried by the network.

Engineering reasons to monitor network traffic. Developers wish to ensure that their products behave in the manner they expect. Software developers debug their implementations of protocols by monitoring data being exchanged between hosts, verifying that it meets their specifications. Network and protocol designers wish to understand how their networks and protocols behave in real world situations. Protocol features that seem sensible at design time may not function as expected when they are implemented, requiring the designer to monitor the implementation and improve the specification. Such improvements may be necessary to ensure that upper layer protocols do not thwart optimisations provided by lower level protocols.

These hard requirements are compounded by curiosity. Computers provide few cues to their internal activity, meaning that even a knowledgeable user has virtually no way of telling what their computer is doing. Most computers provide only gross generalisations of their network traffic through blinking lights on network cards, or graphs of traffic volume. We want something more.

1.2 Network Traffic Is Difficult to Monitor

There are many good reasons to monitor network traffic, but traffic monitoring is hard to do. Various factors, including the volume of data, the design of networks, and missing information all conspire to make monitoring difficult.

1.2.1 Difficulties Caused by Traffic Volume

Computers do a lot. Every computer is busy performing tasks requested by the user(s) that are logged into it, as well as the tasks necessary to maintain itself. Users trigger network communication directly when they request information that is off of the current host, by viewing a web page, triggering a peer-to-peer download, or by sending or checking email. At the same time, the computer is performing background tasks, such as updating lists of virus signatures, checking for software updates and patches, silently acquiring licenses for media and software, as well as many other tasks.

That communication adds up, causing large amounts of traffic to be exchanged across networks. The structure of the Internet causes traffic to be concentrated from individual host machines, through a series of gateways, onto a set of backbones. At each gateway, the traffic of more and more machines are mixed together, causing higher data volumes.

All of this data has an impact on network monitoring. The volume of data is challenging because it forces the analysis tools to work on data outside of memory.

1.2.2 Difficulties Caused by Protocol Layering

The design of the Internet, and the applications that use it has been incremental. Many advancements in the functionality of the network have been due to small ad-

ditions to protocols, or the addition of new data types to be carried by protocols. These small changes and additions have created a complex and varied environment, requiring protocol analysis tools to be adept at handling many different formats and protocols, layered upon one another.

Protocol stacks require analysis tools aimed at a high level protocol to be able to reconstruct the conversations being run on each lower level protocol. If we wish to analyse a BitTorrent stream[2], our analysis tool must be able to analyse each of the lower layer protocols: in this case, IP, TCP, and HTTP.

An analysis tool must be able to interpret more than just protocols. Often, protocols carry information in formats that are not specified by the protocol that act to obscure the content. An example is SMTP. If we wish to monitor text carried by email, we must be able to analyse the SMTP protocol itself, but we must also be able to analyse the content layered on top of it: MIME encoding carried by SMTP, as well as the payload of the MIME attachments. Due to the layers of payload, the SMTP protocol is effectively more complex than in its original specification.

1.2.3 Difficulties Caused by Chain Reactions

Often, one protocol will require the use of another protocol to gather enough information for the host to be able to use it properly. An example that reaches back to the dawn of the Internet is that of address lookup: before a host *Client* can connect out to some other host *Server* with an IP-based protocol, *Client* must perform an domain lookup with DNS to ascertain *Server*'s address, an ARP to determine how to contact *Server*, before communication can begin. In such a situation, we must reach into the ARP, DNS, and IP packets to stitch these three interactions together. Using a high level protocol has triggered a chain reaction, causing two other protocol

[2] `http://bittorrent.org/protocol.html`

events.

That example is showing its age, however, as ARP, DNS, and IP packets are fairly easy to parse. The problem becomes more complex with dealing with peer-to-peer protocols, such as BitTorrent, which requires a user to download a torrent file, feed that to their client, which then connects to a remote tracker and a set of peers. From a traffic perspective, those transactions occur over HTTP, and share the single host in common. To determine that those connections are BitTorrent connections, and to determine that they are part of a larger interaction, we must dig into the HTTP streams and look for commonalities imposed by the BitTorrent protocol. Again, the high level protocol, BitTorrent, has caused a number of lower level protocol events.

1.2.4 Difficulties Caused by Repurposing

The use of HTTP and HTML allows an almost client/server like interaction between two hosts that can make the purpose of a conversation very difficult to guess. Using HTML, a server can construct web pages that act like desktop applications. If a server provides a full mail application to a user via HTTP and HTML, it has effectively repurposed the protocol to be in the same category as SMTP, POP, and IMAP. When looking at such a conversation, a naive analyst will see only HTTP conversations (mixed in with HTTP sessions used by other protocols, such as Subversion, and various flavours of instant messaging), when they should be aware that they are seeing mail transactions.

1.3 Monitoring Network Traffic Well

The ideal network monitoring tool would extract and cross reference all information available in a network conversation. The tool would provide an infrastructure for the

user to understand the data, and verify the correctness of their understanding. Such a tool would climb the network stack, operating at the lower, datagram-oriented layers where most existing tools operate, as well as the application layer, where most data is carried. Ideally, the tool would export a programmatic interface, allowing users to write programs to perform novel analysis of data. No such tool appears to exist. In an effort to create such a tool, we propose adding a new arrow to the quiver of network researchers: a library for the extraction of information from network conversations.

The library we propose is stream oriented. To avoid the effort of implementing every known RFC-compliant protocol from scratch, we also propose a grammar for specifying stream oriented protocols. The grammar, which we have titled the *Network Protocol Grammar*, is the core of this thesis. The NPG provides a mechanism to easily define a protocol with an easy to use grammar, and then build a parser for that protocol. NPG's novelty is its ability to specify structures that are impractical to parse with ordinary context free grammars.

A protocol-definition grammar is superior to hand-crafting parsers for individual protocols for the same reasons that using high level languages and compiling to machine code is superior to writing machine code directly.

Hand crafted parsers are prone to errors (such as Wireshark's[3] reported buffer overflows [15]). Those errors may be exploited by an attacker, and turned into an attack vector; as was the case with the Witty worm, which attacked a protocol reconstruction feature in a firewall product[55]. It seems likely that such a parser could run at comparable speeds to hand written code, as the parser generator could be tuned to generate efficient code.

One of the strongest reasons we have found for using the NPG is its ease of use.

[3]Wireshark was known as Ethereal until a recent name change. For more information, see `http://www.wireshark.org/faq.html\#q1.2`. Even though we will refer to Wireshark by its new name, it is occasionally necessary for us to refer to the old Ethereal website.

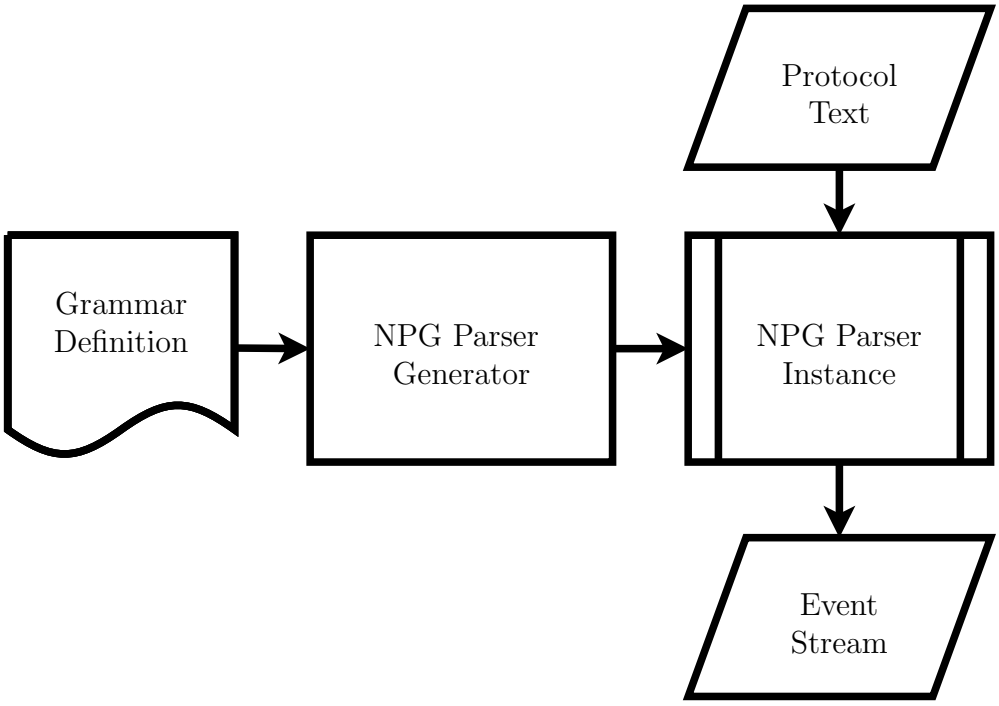


Figure 1.1: Flowchart illustrating the position of the Network Protocol Grammar in the network monitoring tool chain.

Protocols can contain constructs that are difficult to describe in programmatic code, such as repetition and recursive regions. Context free grammars, (which NPG is based on) are effective at parsing that class of structure.

Unlike normal context free grammars, NPG contains a number of features to reduce the size of grammars for common protocol idioms. The NPG allows a grammar to modify itself during a parse. The changes are parametric, either changing the number of times a symbol is allowed to match, or the text of a terminal in the grammar. Although a context free grammar *can* be used to describe such a structure, it would have to contain every possible combination of parameters. Such a grammar would be impractically large; both for a human to describe, and to fit into the memory of a modern machine.

1.4 Contributions

The contributions of this thesis can be summarised as follows:

- We have described two idioms used by streaming protocols that make traditional context free grammars impractical for expressing streaming protocols.
- We have created the Network Protocol Grammar, a grammar for concisely expressing streaming protocols.
- We have implemented a parser generator for the Network Protocol Grammar parser and made it available at `http://qcap.sourceforge.net`.

1.5 Contents

The remainder of this document is split into six chapters.

Chapter 2 provides background on network protocols and grammars. It describes existing tools for network monitoring, and existing languages for defining protocols.

Chapter 3 describes the Network Protocol Grammar, and explains design choices that were made throughout its development.

Chapter 4 describes `qcap`, a network analysis library that contains a parser generator for the NPG.

Chapter 5 contains an evaluation of our grammar, and a comparison to existing protocol parsing techniques.

Chapter 6 provides a discussion of the Network Protocol Grammar, including its limitations and its implications for grammar design and protocol development.

Chapter 2

Background and Related Work

Exploring and understanding network data is difficult. Networks carry large volumes of data, in a variety of formats, using many protocols. Every format and protocol presents a challenge to a would-be researcher, as it presents another layer of encoding that must be deciphered. Any attempt to untangle protocols requires many programmer hours to cobble together the existing set of meager tools to build an ad hoc solution to provide an answer to some desired question.

To understand what is happening on networks, we need network monitoring tools that can analyse any part of the network stack and cross reference behaviour of various parts of the stack. These tools should provide a programmatic interface to allow novel explorations to be built. Currently, these tools do not exist.

The Network Protocol Grammar and the `qcap` library allow access to the network stack, providing building blocks for more comprehensive analysis tools. Before we can discuss either contribution in depth, we must provide background on computer networks, network monitoring tools, formal languages, and previous use of formal languages for network analysis.

2.1 Computer Networks

Computer networks are complex systems. They are composed of many communicating computers using many different protocols to exchange information. This section briefly reviews the actors on a network and the common structure of networking protocols.

When we talk about computer networks, we are almost always talking about some variant of the Internet. The Internet is a set of intercommunicating computer networks that share a basic communication protocol and addressing scheme. The networks are populated by computers, which we term either *hosts*, *machines*, or *nodes*. Running on each computer is a *kernel* which handles low-level networking[1], and a number of *applications* which perform tasks on behalf of a user.

The network is a communications medium. It relays data from one computer to some set of others. In order for computers to communicate the data exchanged must follow some prearranged format. The format is called a *protocol*. Hosts communicating with each other are often termed the *endpoints* of communication.

We define a protocol as requiring three pieces of information: roles for the participating computers, languages to be used when communicating from any one role to any other role, and a set of semantics explaining the significance of the communication. The semantics of communication are interesting, because they allow a protocol to provide features that are not available on the medium it is using to communicate. We will explore this idea in the next section.

Internet protocols have become ubiquitous. Even if hosts on a network are not part of the Internet, it is likely that they use some of the protocols in the Internet

[1]We do not claim that the kernel must handle all aspects of networking, but we do observe that production operating systems seem to make the kernel or one of its subsystems responsible for the task.

suite. Given that omnipresence, we will focus solely on Internet protocols for the remainder of this thesis.

2.1.1 The Internet Environment

As its name suggests, the Internet is a network of networks. Each network may be running different protocols, providing different features to the computers using them, but they are still able to communicate, thanks to the Internet Protocol (IP). IP hides the differences between the different types of network, and provides a common languages for all computers to speak.

One of the core ideas behind the Internet is that computers need not be physically connected to one another to communicate. Some source computer S may be located on one network, while the destination computer D is on an entirely different network, possibly separated by some set of other networks. So long as there is a path between S and D, and S knows D's Internet address, they should be able to communicate.[2]

The Internet suite consists of many protocols layered on top of IP. We shall focus on the protocols between IP and the stream-oriented application layer, ignoring all others. The layering can be seen in Figure 2.1. Each layer in the stack adds capabilities to the layer below.

Protocol Stack	**Layer**
Stream-oriented: HTTP, SMTP, FTP	Application layer
TCP	Transport layer
IP	Network layer
Ethernet	Physical layer

Figure 2.1: A subset of the Internet protocol suite.

[2]Ignoring incidentals, such as firewalls.

The bottom-most layer is the *physical layer* or *link layer*. It may be any protocol (from Ethernet to carrier pigeons[57]) that provides a mechanism for hosts to exchange blocks of data on a local network.

Above the physical layer is the *network layer* that allows hosts to exchange messages (called *datagrams*) across network boundaries. In the Internet protocol suite, the network layer is provided by the *Internet Protocol*[50]. Although multiple versions of the Internet Protocol exist, we are most interested in the most commonly used version at the moment: IP version 4. We will refer to IPv4 as IP.

IP provides the core Internet addressing scheme, and it provides a mechanism for datagrams to be fragmented and reassembled. Fragmentation and reassembly are necessary because some link layers may have a limit on the maximum allowable datagram size. To allow a large datagram to enter networks with a small datagram size, IP breaks the datagram up into multiple pieces and forwards them on. Each piece is given the same ID, and numbered, to allow the recipient host to rebuild the pieces to create the initial datagram.

IP is an unreliable protocol, meaning that it makes no guarantee that a datagram will be delivered when it is sent. Similarly, it does not guarantee that datagrams will be received in the same order that they were sent.

IP is used by a number of *transport layer* protocols. We limit our discussion to the Transmission Control Protocol (TCP)[51].

TCP is more complex than the protocols running beneath it. Where IP exchanges datagrams between hosts, TCP erects a connection. Where IP sends packets in the hope that they arrive at the host, TCP tracks every byte sent, ensuring that none are lost and that every byte arrives in order. The most notable difference between TCP and IP is that TCP is *stream oriented*, meaning that applications using TCP do not send blocks of data; instead, they send a continuous stream of bytes. This

allows applications to treat TCP connections almost as files, reading and writing bytes without having to handle packet loss or reordering issues. One side of a TCP connection is often called a *stream*.

TCP uses ports to demultiplex incoming streams to specific applications.

The top layer of the Internet stack is the *application layer*. Application layer protocols are used by user level applications to exchange data for more specific purposes. Examples of these protocols include the HyperText Transfer Protocol, HTTP[17] which is used to download web pages; the Simple Mail Transfer Protocol, SMTP[52], which is used to transfer emails from sender to recipients; and the File Transfer Protocol[53], which is used for bidirectional file transfer.

2.1.2 Existing Tools for Understanding the Network

As we said in Section 1.3, the ideal network analysis tool would extract and cross reference information about observed network data, allowing a user to verify that their understanding of the network data is correct. Given the discussion of network protocols in Section 2.1.1, we extend the definition of the the ideal tool to say that it must defragment IP packets, and reconstruct TCP streams.

This section lists some of network analysis tools that are currently available. The listing is not intended to show every available tool; instead, it is intended to highlight classes of tools that may be of interest to researchers and protocol implementors. This listing originally appeared in our paper "Towards Network Awareness"[30].

This list is ordered by complexity. The simplest tools appear first, while the more complex tools follow.

BSD Packet Filter

The root of many existing packet capture tools is the BSD Packet Filter [41], which defines an elegant approach for winnowing packets based upon a textual predicate. The predicate is compiled from a restricted language into instructions for a tiny virtual machine, which can run in the packet capture device driver within the kernel. The BPF architecture has been widely accepted and incorporated into the `pcap`[56] packet capture library.

The BSD Packet Filter is designed to capture a subset of packets quickly. Although it is extremely fast, it only provides an application with access to the physical layer of the network.

On its own, the BSD Packet Filter does little to enhance a user's understanding of network traffic, although it does provide functionality necessary for building network analysis tools, namely datagram capture and storage. `pcap` does not provide any information on protocols above the physical layer.

Network Debugging Tools

`pcap` and its parent application, `tcpdump`, have spawned a number of command-line based open-source progeny, including `tcpstat`[28] and `tcptrace`[46], that are well suited to auditing and debugging network traffic. In the more complex niches, we find ChaosReader[25], a command-line based stream reconstruction tool that is able to rebuild application streams and store them as files; Snort[2], a signature-based intrusion detection tool; and Wireshark[3], a graphical packet display tool. Commercially available tools [12, 11, 31] are useful for monitoring networked devices, diagnosing faults, and other administrative tasks.

These tools are designed with basic network monitoring in mind. They provide users with access to network data, and perform some degree of cross referencing.

`tcpdump`, `tcpstat`, `tcptrace`, provide the user with basic statistics about datagrams seen. Snort matches packets based on signatures, performing IP defragmentation and TCP reconstruction, but does not provide information on the nature of streams, or their payload. These tools provide access to lower layers of protocols, without providing cross referencing, and analysis of data in higher level protocols.

ChaosReader and Wireshark each provide users with access to the contents of TCP streams, but do not provide any cross referencing between the datagrams seen at the TCP or IP layers, and the higher level streams. None of the tools export an API, nor do they provide information about the content and behaviour of application level protocols.

Tools from VizSec 2004

The next level of abstraction we consider are network monitoring and understanding tools. These are explicitly designed to provide network administrators with a picture of the state of the network, usually for security purposes. An entire crop of these tools were presented at VizSec 2004 and 2005 [37, 61, 6, 44], although older tools exist as well[7].

In general, most of these tools are graphical and use some variant of a two or three dimensional display to render network activity. The displays usually place network endpoints on two axes and the volume of traffic travelling between those endpoints on the third axis. VizSec 2005 presented many tools that rendered network activity as scatterplots[37, 6, 44] and dual axes graphs[61]. Many of the tools presented at VizSec 2005 share the same display modes[39], although there are some with different approaches [18, 45].

For the most part, the VizSec tools provide statistical information on data travelling between source and destination endpoints. The endpoints may take the form

of one or more networks, hosts, or ports. The data may be presented in terms of packets, connections, bytes, or some other volumetric measurement.

Although volumetric data gives some indication of who is talking to whom, it does not provide an indication of what is happening on the network. At best, an analyst or tool can guess at the role of each host, and the protocol in use, and make a supposition about the nature of the communication. Volumetric analysis does not provide enough information to evaluate interactions between layers of the protocol stack. None of the tool surveyed exported an API.

Forensic Tools

Forensic tools[1, 12] delve into the contents of data streams. They provide reconstructions of data stream contents, often indexing it for searching, and support retrieval of text deemed to be of interest to users. They provide some idea of the classes of information that can be detected with full packet analysis. In particular, these tools can aggregate network events into conceptual events and display those conceptual events grouped by type. For example, a listing of all discrete "login" events for the network can be presented, indexed by the credential used, and the resource acquired; as can all downloads (via HTTP, FTP, or BitTorrent); message sends (via SMTP, IM, SMS, or IRC); or file access (via SAMBA, NFS, or DAV). They provide access to the payload of some application layer streams using an appropriate renderer (such as reconstructing and playing the audio portion of VoIP traffic).

While forensic tools might appear to be ideal tools for exposing higher layers of the network stack, their list-oriented interfaces are biased towards answering specific (not general) questions, provide little support for correlating high-level semantics with low-level packet behaviour, and they offer few mechanisms for comparisons and anomaly detection. These limitations arise because these tools are designed to help dissect the

specifics surrounding a particular incident rather than to help detect patterns that are not known in advance. Also, the forensic tools that the author tested were closed source, meaning that they are difficult to modify for use with problems that were not envisioned when the tool was designed.

2.1.3 Programming Idioms in Network Analysis Tools

There are few open source network analysers that reach into the application layer. Two examples of those that do are ChaosReader and Wireshark. We will use those as examples to explore how application layer data is currently parsed by analysis tools.

The most recent version of ChaosReader (.94) is a 6,836 line long Perl script. It extracts payloads from HTTP, NFS, SSH, FTP and SMTP, and allows a user to replay X11 sessions and VNC sessions. Each extraction routine is written in hand-coded Perl, using heuristics to extract interesting portions of each conversation.

Wireshark is a more mature tool, written in C with a full datagram-oriented GUI. Although Wireshark is datagram oriented, it uses heuristics to provide application-level information about datagrams. The heuristics are packaged up into an API for dissecting individual packets. Like ChaosReader, Wireshark uses hand-coded heuristics to guess at the nature of the given packet.

Writing stream parsers by hand is complex, as can be seen in Wireshark's documentation[3] and Wireshark's protocol parsers[4]. It is also prone to error, as can be seen by the security advisories for Wireshark's protocol analysers[5].

[3] `http://anonsvn.wireshark.org/wireshark/trunk/doc/README.developer`
[4] `http://anonsvn.wireshark.org/wireshark/trunk/epan/dissectors/`
[5] See `http://www.ethereal.com/appnotes/enpa-sa-00023.html`, and [15]

2.2 Grammars

As we mentioned above, protocols consist of three parts: roles for the participants, languages describing communication between the participants, and semantics explaining how the participants should behave. To improve the understanding of network traffic, we focus on the languages involved in network communication, because they provide a window on the interactions the hosts are taking part in.[6]

The languages used by communicating hosts are defined in protocol specifications. The specifications are typically a mixture of human-readable text and formal grammar. To build a network monitor, the language specification must be transformed from prose and grammar into code that is executable by a computer. The nature of the transformation depends on the nature of the specification.

Ideally, the specification will contain a formal grammar that *completely* describes the languages that the participants use. A formal grammar can be transformed from a textual specification to a parser, meaning that we could generate a portion of the network monitor for a protocol directly from the specification.

Many protocol specifications, however, split the language definition between the prose and the formal grammar, meaning that a parser cannot be generated automatically. Instead, a human must be involved in the transformation, which likely involves writing Turing complete code to parse the conversations.

Formal grammars are a powerful method for concisely defining a language, and, at the same time, specifying a parser. In the following section we discuss the nature of languages, and define what exactly we mean by the terms *language*, *grammar*, and *parser*.[7]

[6] Those communications are often available in plain-text to anyone with a network tap. So long as the proper legal and ethical steps have been taken, the communications are much easier to monitor than the state of the kernel and applications running on the endpoints of communication.

[7] Unless otherwise noted, the source of most information from this section is [4].

2.2.1 Terminology

A *language* L consists of an *alphabet* and a set of *strings*. The alphabet is a set of *symbols* that are unique. The set of letters in English is an alphabet, with each letter being a symbol. The ASCII and Unicode character sets are also alphabets, albeit with many more symbols. If we join zero or more symbols together in a sequence, the result is a string.

There is no limit to the possible set of strings that compose a language. For simple languages, we can exhaustively define every possible instance of the language in a finite set. But the protocols we are interested in contain an infinite number of possible strings. In such a situation, the only practical way to define which string is part of a language, and which is not, is to define a set of rules. Informally, we could write a rule operating on the English alphabet by saying "any vowel followed by two consonants, followed by a vowel", which would match "*abba*", "*alto*", and "*ulna*", along with 11,022 other strings.

A set of rules is usually called a grammar. If we have a grammar G, the language defined by those rules is written as $L(G)$.

When we have a grammar G and some string of input text i, we can perform one of two operations: we can test to see if G *recognises* i, or we can use G to *parse* i. Recognition is a test to see if i is an element of $L(G)$. If the recognition test passes, the input can be described by the grammar, otherwise the input is part of some other language. Parsing involves using G to decompose i into its component parts. The exact process that is followed depends on the nature of G. We will discuss parsing techniques for formal grammars in Section 4.5.4.

When a grammar is recognising or parsing an input, there are two possible results. The grammar may either *accept* the input, meaning that it recognised the input; or it may *reject* the input, meaning that the grammar cannot describe the input string.

If a grammar accepts an input, we may also use the term *match* to indicate that the input is a member of the grammar's language.

2.2.2 Formal Grammars

Even though rule sets can be arbitrary, a bulk of academic research has been done on a subset of grammars called *formal grammars*. Formal grammars were pioneered by Chomsky in the 1950s[26], and have formed a basis for most research and progress on formal languages, parsers, compiler construction, not to mention Chomsky's home turf of linguistics.

The formal grammars we will be dealing with are *generative grammars*, meaning that they can be used to create every possible instance of a language. More precisely, given some generative grammar G, we can create every instance of $L(G)$ through a process of *expansion*.

Parts of a Formal Grammar

Formal grammars are comprised of four parts: a set of *terminals*; a set of *nonterminals*; a set of rules; and an indicator denoting which rule forms the start of the grammar. Terminals are very similar to the alphabets that we discussed earlier, except that they may consist of strings of characters; nonterminals are sets of terminals grouped together.

Occasionally, it will be convenient to refer to terminals and nonterminals interchangeably. The standard word for either a terminal or nonterminal is *symbol*, which also happens to be the standard word for a member of an alphabet in a language. Unless otherwise noted, our use of the word "symbol" will refer to a terminal or nonterminal.

Rules are the interesting part of a formal grammar, as the other three parts support

the rules. The format used to describe rules is called *phrase structure grammar*[26]. Figure 2.2 shows an example of phrase structure grammar.

```
LawOfTheJungle = Predator " eat " Prey

Predator = "Giant spiders"

Prey = "gazelles"
```

Figure 2.2: A small example grammar showing off phrase structure grammar. The grammar shall be referred to as G_{Jungle}.

A phrase structure grammar is made up of rules. Each rule looks like an assignment and is terminated by a newline. On the left hand side (LHS) of the assignment we have one or more symbols. On the right hand side (RHS), are zero or more symbols. Whenever symbols appear in the LHS of a rule, they can be replaced with all of the symbols on the RHS.

We can generate every legal instance of $L(G_{Jungle})$ by substituting the RHS of the productions <`Predator`> and <`Prey`> into <`LawOfTheJungle`>. In this case, the result is unspectacular consisting solely of the single string "*Giant spiders eat gazelles*". The process of expanding <`LawOfTheJungle`> is shown in Figure 2.3.

Phrase structure grammars would be useless if they could only generate a single string. Not surprisingly, phrase structure grammars provide a number of features

Action	**Result**
Starting state	<`LawOfTheJungle`>
Expand <`LawOfTheJungle`>	<`Predator`> **“ eat ”** <`Prey`>
Expand <`Predator`>	**“Giant Spiders”** “ eat ” <`Prey`>
Expand <`Prey`>	“Giant spiders” “ eat ” **“gazelles”**

Figure 2.3: Example expansion of a grammar
Expansion of G_{Jungle} from the topmost nonterminal <`LawOfTheJungle`> into a string. Each expansion is highlighted in **bold**. Note that G_{Jungle} only has one valid expansion.

that make them much more expressive.

```
LawOfTheJungle = Predator " eat " Prey

Predator = "Giant spiders"
Predator = "Lions"

PreyList = Prey ", " PreyList
PreyList = Prey

Prey = "gazelles"
Prey = "flowers"
```

Figure 2.4: An example of a phrase structure grammar showing alternation and recursion. The grammar shall be referred to as $G_{Jungle2}$.

Figure 2.4 augments our grammar to include *alternation*, that is, nonterminals with multiple possible values. When we generate $L(G_{Jungle2})$, each nonterminal choice will allow us to produce another string that is part of $L(G_{Jungle2})$. If we momentarily ignore <`PreyList`>, our language will consist of: "*Giant spiders eat gazelles*", "*Giant spiders eat flowers*", "*Lions eat gazelles*", and "*Lions eat flowers*".

The rule <`PreyList`> is recursive. When it is substituted into <`LawOfTheJungle`>, it forces the resulting text to contain two occurrences of <`PreyList`>, rather than just one. During a repeated process of substitution, we can therefore add an infinite number of <`PreyList`> instances to the result, making $L(G_{Jungle2})$ infinitely large. To illustrate the result of a recursive <`PreyList`>, we will show some text strings that can be generated by $G_{Jungle2}$: "*Lions eat gazelles*", "*Lions eat gazelles, flowers*", and "*Lions eat gazelles, flowers, gazelles, gazelles*".

To save space, the phrase structure grammar, allows us to combine alternatives onto a single line. Each RHS from the named production is separated by vertical bar ("|"). A more succinct version of $G_{Jungle2}$ is shown in Figure 2.5. When line-length allows, we will use this short-hand.

```
LawOfTheJungle = Predator " eat " PreyList

Predator = "Giant spiders" | "Lions"

PreyList = Prey ", " PreyList | Prey

Prey = "gazelles" | "flowers"
```

Figure 2.5: A condensed version of the grammar found in Figure 2.4. <Predator> and <Prey> have been collapsed from multiple productions into single productions.

More Complex Formal Grammars

So far, the LHS of our grammars have been populated by individual nonterminals. This is not a requirement: the LHS may contain any number of symbols.

```
LawOfTheJungle = Predator " eat " PreyList $End

Predator = "Giant spiders" | "Lions"

PreyList = Prey ", " PreyList | Prey

Prey = "gazelles" | "flowers"

", " Prey $End = " and " Prey $End
```

Figure 2.6: Phrase structure grammar with multiple symbols in the LHS of a production. Note that we use the special terminal <$End> which is simply used to represent the end of the string be generated. We use this terminal to limit the locations where a production will match. This grammar is termed $G_{NiceJungle}$.

Figure 2.6 makes generation of a grammar more interesting. The final production, with the LHS <", " Prey $End> will match any instance of a comma, followed by a <Prey>, followed by the <$End>. A sample expansion of $G_{NiceJungle}$ is shown in Figure 2.7. This expansion follows the same recipe we saw before, until it reaches the point where <", " Prey $End> are expanded. At that point, all three symbols are

Action	Result
Starting state	<LawOfTheJungle>
Expand <LawOfTheJungle>	<Predator> **“ eat ”** <PreyList> <$End>
Expand <Predator>	**“Lions”** “ eat ” <PreyList> <$End>
Expand <PreyList>	“Lions” “ eat ” <Prey> **“, ”** <PreyList> <$End>
Expand <Prey> and <PreyList>	“Lions” “ eat ” **“flowers”** “, ” <Prey> <$End>
Expand <", " Prey $End>	“Lions” “ eat ” “flowers” **“ and ”** <Prey> <$End>
Expand <Prey>	“Lions” “ eat ” “flowers” “ and ” **“gazelles”** <$End>

Figure 2.7: A sample expansion of a string in $L(G_{NiceJungle})$.

replaced by the RHS of the production, which has the effect of swapping the comma for an “and”. Note that the production will only match at the end of the string.

The phrase structure grammar provides us with two facilities: the ability to define replacement rules; and the ability to provide a set of alternative paths to use during generation. It may not be immediately clear how a phrase structure grammar can be used to describe a network protocol. However, as we will discover, we can make phrase structure grammars more useful by limiting their syntax.

2.2.3 Power Through Simplification

Formal grammars, as expressed with a phrase structure grammar, are extremely powerful. According to [26], it can be proven that there is no (known) stronger method for generating sets. Having said that, the complete phrase structure grammar that we discussed is computationally unwieldy: in its full form, a phrase structure grammar is too complex to transform into a parser.

We would like to simplify the phrase structure grammar to the point where we can run a transform on it and build a parser. When a limitation is placed on the phrase structure grammar, that reduces the set of languages it can be used to describe, we say that we have lessened the *expressiveness* of the grammar. We wish to limit the complexity of the phrase structure grammar to the point where it is possible to

transform it into a parser, while keeping it sufficiently expressive that we can still describe network protocols.

In the academic cannon, there are standard limitations that can be placed on a phrase structure grammar, to allow for the easy creation of parsers. The limitations are organised into what is known as the Chomsky Hierarchy[26], which orders formal grammars from "most expressive" to "least expressive". Each limitation is assigned a number, and, in most cases, a name.

Type-0

Type-0 grammars are the fully featured phrase structure grammar that we have already discussed. There are no limitations on the number of symbols that may appear in either the left or right hand sides of the production.

Type-1: Context Sensitive Grammars

Grammars are context sensitive if every production contains only one non-terminal on the LHS that is changed by the RHS. In other words, each rule modified one non-terminal in the LHS, all other symbols in the LHS are considered context. The RHS provides the same values for each symbol of the context, and a new value for the for the modified symbol.

```
c1 c2 Target c3 = c1 c2 Some New Values c3
```

Figure 2.8: A Type-1 grammar. Rules <c1>, <c2>, and <c3> are context. Rule <Target> is changed.

Figure 2.8 provides an example of a context sensitive grammar. The special non-terminal is <Target>, and the context is <c1>, <c2>, and <c3>. Each of <c1>,

<c2>, and <c3> remain effectively unchanged by the rule, while <Target> is replaced by a set of new values.

Type-2: Context Free Grammars

Context free grammars (or CFGs) have a single symbol on the LHS of a production, and an unlimited number of symbols on the RHS. This means that any non-terminal that is expanded in a context free grammar is completely independent of the surrounding symbols.

Even with the limitation of a unary LHS, context free grammars are still very expressive. They can describe grammars with nested elements, and offer recursion to arbitrary depths. They are often used to describe programming languages and network protocols.

Because of their limitations, type-2 grammars may be parsed in linear time by finite automata equipped with a stack.

Type-3: Regular Grammars

Regular grammars are an even more highly constrained language. Like CFGs, regularly grammars may only have a single non-terminal on their LHS. However, unlike any of the preceding types, their RHS is limited to containing a sequence of terminals, followed by a non-terminal.

This structural limitation prevents regular grammars from generating any nested or balanced structure. With such a limitation, a simple finite automata is sufficient to represent a regular grammar.

2.2.4 The Drawback of Formal Grammars

As powerful as formal grammars are, they suffer a single drawback that makes them unattractive for defining protocol grammars: they require explicit enumeration of parameters. We term a parameter of a protocol to be some value that changes while two hosts are communicating. When the parameter changes, it changes how the recipient of some communication parses that data.

Practically, protocol parameters are used to control when the parser detects the end of a symbol. There are a number of idioms for the length of a symbol can be determined:[8]

1. The length of the symbol can be explicitly stated in the protocol specification.

2. The end of the symbol may be delimited by some terminal stated in the protocol specification.

3. The length of the symbol is encoded in some preceding nonterminal. We refer to the preceding symbol as the parameter. We refer to the variable-length symbol as having *dynamic length*.

4. The end of the symbol may be denoted by some string defined by a preceding symbol. We refer to the preceding symbol as the parameter, and the symbol whose value changes as having *dynamic text*.

The first two idioms are perfectly compatible with formal grammars, as they can easily be stated with any grammar more complex than a regular grammar. Figure 2.9 and Figure 2.10 show each idiom.

In a context free grammar, dynamic length and dynamic text can only be stated by exhaustively enumerating every possible parameter. If we define a length in a

[8]This list has been highly adapted from that supplied by Lambright in [38], which was more closely associated with datagram oriented protocols.

```
List = Thing Thing Thing Thing Thing
Thing = "A" | "B" | "C"
```

Figure 2.9: An example of explicit length definition in a formal context free grammar. Note that this grammar will match exactly five instances of the letters "A", "B", or "C".

```
List = Thing "AXAX"
Thing = "A" | "X" | Thing
```

Figure 2.10: An example of explicit symbol delimiter in a formal context free grammar. Note that this grammar will match exactly any number of "A" and "X", so long as the end is delimited by "AXAX".

context free grammar, we must list each possible length, as shown in Figure 2.11. The grammar defines <List> in a manner that allows us to correctly parse all instances of the language, but it is not scalable. Consider a scenario where we have a list that may contain a few thousand values. We would have to include *e*very possible length in the grammar. The "Content-Length" field in HTTP [17] is a practical example of this type of parameter.

```
List = "0" | "1" L | "2" L L | "3" L L L
L = "A" | "B" | "C"
```

Figure 2.11: An example of explicit length definition, for a list of variable length in a formal context free grammar.

The same is true for dynamic text. The dynamic text idiom involves three symbols: a definition, a variable length field, and a terminator. The definition sets the text for the terminator. The terminator delimits the end of the variable length field. Figure 2.12 shows a grammar with the given delimiter. Even in this example, if we allow the delimiter to be any English letter, we will have 26 branches. If we also allow the delimiter to be a string of multiple letters, the size of the grammar will increase

exponentially. The boundary delimiter of MIME [21] is a practical example of this type of parameter.

```
List = "X" L "X" | "Y" L "Y" | "Z" L "Z"
L = "A" | "B" | "C" | L |
```

Figure 2.12: An example of the dynamic text idiom. In each branch of the alternation in <List>, the first terminal is the definition, the nonterminal is the variable length field, and the second terminal is the terminator.

In order to refer to both dynamic length and dynamic text together, we use the term *dynamic elements*.

When using a formal grammar to define a protocol, parameters make life difficult, but protocol designers have good reason for using dynamic elements. If a sending node wishes to send a block of data, of a known size, it can use dynamic length to tell the receiver how much data it is about to receive. If the sending node does not know the size of data it is about to send (because it is being compressed on the fly, for example) dynamic text allows the sender to specify a terminator that the receiver can use to delimit the end of the data block.

If we are to use a formal grammar to describe network protocols, we must support dynamic elements.

2.2.5 Existing Grammars Describing Network Protocols

Before defining our own grammar to define network protocols, it is worth exploring previous work on this problem. The author found six different grammars for protocol specification, which are described below. Half of the specifications are designed to describe datagram oriented protocols, while the other half are used to describe stream oriented protocols.

After we describe each grammar, we compare the grammars to each other.

Protocol Filter Specifications

When monitoring network data, it is often desirous to create a *filter* to detect different classes of datagram. Creating a filter is a two step process: first, a textual predicate containing assertions about fields in a datagram must be written; and second, the predicate must be compiled, transforming the predicate into a series of instructions, telling a recogniser values that must be present in a datagram for the predicate to pass.

To simplify the compilation of predicates, datagram protocols are defined in a language similar to a context free grammar. During compilation, the predicate compiler can resolve the location of a field by consulting the context free grammar.

Chandra and McCann[10, 40] propose *Packet Types*, a specification language for datagram filters. Programmers using Packet Types specify datagram structure by using rules. Packet Type rules perform a role similar to those of the phrase structure grammar, but are structured as a series of name-value pairs making the specification look like a series of C structures.

In order to handle dynamic elements, each symbol in a Packet Type grammar has a series of verifiable attributes, including `value`, and `numelems`. The `value` attribute provides the ability to test a terminal against a value seen in the protocol text. `numelems` provides the ability to validate the number of times a symbol repeats. Each rule has a `where` clause that contain verifiable predicates that can test the values of any symbol.

Lambright and Debray[38] create a similar language for packet filters, called APF. Like Packet Types, APF embeds constraints into a set of rules. Although the aim of APF is to provide a mechanism for generating packet filters (similar to BPF[56, 41]),

the grammar appears to be sufficiently expressive to specify protocol text.

The primary distinction between Packet Types and APF is that Packet Types provides a section for constraints and assignments at the end of each production. APF embeds each statement into the text of the production itself. Like Packet Types, APF does not provide alternations in its grammar.

Not all filter specifications recognise the need to describe dynamic entities. Jayaram and Cytron [33] assert that "it is simple to express such varying length fields in grammar form." Their paper only describes grammars for protocols with dynamic entities up to 16 repetitions in length.

Protocol Specifications

Anderson and Landweber[5] create a methodology to express the semantics of streams of datagrams. Their methodology, "Real-Time Asynchronous Grammars" (RTAG) provides a language with the same name. RTAG is similar to a context free grammar, whose terminals are events in a network data stream, instead of blocks of data. Events are detected by hand-written code which parses incoming datagrams, extracts useful data, and creates data structures suitable for consumption by the RTAG processor.

Like APF, the RTAG grammar provides a mechanism to embed blocks of code directly into grammar productions. Each block can modify attributes of other productions, run conditional blocks, and execute external functions.

Note that RTAG is designed to specify the behaviour of an entire protocol, it does not specify the syntax of individual datagrams.

McGuire's Austin Protocol Compiler[43, 42] is the most ambitious datagram specification language surveyed. Not only does it attempt to define the on-the-wire format of datagram protocols, it also specifies how the protocol should behave; using a language called "Timed Abstract Protocol" or TAP. TAP provides a limited pro-

gramming language that can be translated into C by the Austin Protocol Compiler.

The purpose of TAP and the Austin Protocol Compiler is to create verifiable protocol definitions that can be converted directly into reference implementations.

Borisov et al[8] of Microsoft Research create a full specification for protocols aimed specifically at intrusion detection systems. Their closed source specification framework, called *Generic Application-level Protocol Analyser* or *GAPA*, utilises a language called *Generic Application-level Protocol Analyser Language* (or *GAPAL*). GAPAL provides a layered description of protocols, including a context free grammar for describing protocol text, a state transition graph to handle protocol state, and a set of handlers containing arbitrary code. According to [8], GAPAL is expressive enough to build an entire protocol stack, including IP, TCP, as well as stream oriented application protocols on top of TCP. GAPA can monitor HTTP traffic at rates of 60mbps. The authors of that paper suggest that GAPAL is an efficient way to develop protocol monitors.

Outside of the academic realm, there are three other language specifications worth noting: Augmented BNF (ABNF), Abstract Syntax Notation One (ASN.1), and the eXensible Mark-up Language (XML). Although these languages are not aimed explicitly at stream oriented protocols, they are often used in the definition of stream oriented protocols.

Augmented Backus-Naur Form is used in many Request For Comments documents to define stream oriented network protocols.[9] ABNF is almost identical to the phrase structure grammar discussed above, with a few minor syntactic changes. Unlike ASN.1 and XML, there are no known support libraries for parsing text given an unmodified ABNF definition.

[9]RFCs are the standards definitions of the Internet Engineering Task Force (IETF). Most popular non-proprietary protocols in use on the Internet today are specified in RFCs.

Abstract Syntax Notation One[32], provides three mechanisms. The first allows a programmer to define a *schema* for data. The schema is a set of one or more arbitrarily complex data types that are to be made available to a program. The second mechanism allows the programmer to manipulate objects of the requested type in the program. The final mechanism allows the programmer to transmit or receive those data types, with ASN.1 automatically handling serialisation and deserialisation.

The eXtensible Markup Language[13] is similar to ASN.1. It provides a mechanism for specifying a schema that defines a serialised representation of data. The programmer is responsible for writing data out in a manner that adheres to the schema. Parsers exist that read data adhering to an XML specification, make it available to a program through callbacks, or queryable in-memory representations.

Analysis of Existing Definitions

Of the six approaches, note that none are targeted at the problem we have selected. We wish to create a machine-readable protocol specification language that supports dynamic elements, that can be used to create stream-oriented network monitors. Packet Types and APF support dynamic elements, but are designed for dealing with datagram-oriented protocols.

All of the definitions we have discussed are based on context free grammars. The exact nature of the definition varies, but the form remains the same: a single root structure, consisting of sequences and alternations of either terminals or non-terminals, defined recursively by other structures. Each definition is context free.

We can consider the differences between each approach in terms of datagram/stream orientation, availability of libraries, complexity of the extension to phrase structure grammar.

Most of the solutions considered are datagram oriented, meaning that they are explicitly intended to operate on datagrams. Although the general concepts that are applicable to a datagram oriented approach map easily onto streams (and vice versa) there are likely to be dramatic differences in implementation techniques. Packet Types, APF, and TAP/APC are explicitly datagram oriented. While ABNF, ASN.1, and XML are agnostic in their orientation, they are comparatively verbose languages, meaning that they do not necessarily lend themselves well to the compact nature of datagrams. Note that RTAG parses events instead of datagrams or streams, making it slightly different than the other approaches.

The greatest axis of difference between the approaches is the use of assignments and embedded code. At one extreme, we have ABNF, ASN.1, and XML, that are simply grammar definitions, in the spirit of the phrase structure grammar. At the other extreme is the Austin Protocol Compiler, which embeds executable code defining behaviour into the protocol specification. Between the two are Packet Types, RTAG, and APF, which allow assignments that modify how the parse progresses.

The greatest difference between the approaches is practical. Only half of the languages we have discussed have publicly available libraries to parse an input given a grammar. Those languages are APC, ASN.1, and XML.

Chapter 3

The Network Protocol Grammar

In Chapter 2, we saw grammars used to describe protocols, and an evaluation of network monitoring tools. The tools that we saw were, almost exclusively, datagram oriented. In this chapter we present the Network Protocol Grammar, a new language for succinctly describing stream oriented application protocols in a machine-readable manner. In Chapter 4 we use this language to build a network monitoring library called `qcap`.[1]

Before embarking on an explanation of the NPG, we explain why the NPG is necessary, and derive requirements for the language. With the requirements explained, we outline the language, identifying its novel features. Finally, we justify the design decisions that we made.

3.1 Motivating the NPG

We wish to know who is using the network, for what purpose, and how. "Who" could follow any axis of identity, from specific humans, to accounts, to hosts. Our definition of "purpose" is similarly open ended, we are interested in the reason why any event

[1] `qcap` is available at `http://qcap.sourceforge.net`.

occurs, either due to protocol specification, user behaviour, or processes running on a host. "How" encompasses the text that is sent over the network, and the nature of the host interactions that caused it to be sent.

The tools discussed in Section 2.1.2 provide some degree of network awareness, but in a limited manner. The network debugging and visualisation tools are datagram oriented, providing little information about the content of streams. The forensic tools provide access to stream content, but in an ad hoc manner: they only provide access to a subset of the fields in the stream, and they do not provide a programmatic interface.

In order to build network awareness tools, we need access to all levels of the network stack. Limiting our view to an individual layer blinds us to the big picture of overall network activity. We need to be able to observe all aspects of network communications, in context.

The problem we find ourselves facing is that there are no tools that provide consistent access to every field of application-level protocols. In building such a tool (see Chapter 4), we discovered that defining each protocol in code was time consuming, error prone, and unpleasant. We decided that there must be a better method to delve into stream content.

3.2 Problem Analysis

We wish to build a stream parser that is able to rend a stream down from a long sequence of bytes to a series of discrete fields, each tagged with their significance in the protocol. We wish to be able to define parsers in a manner that is concise, easy to read, easy to write, and minimises errors. The resulting parsers will be used by `qcap` for monitoring stream oriented protocols.

It is possible to create a parser in at least three ways. The most basic approach, taken by WireShark, is to implement a parser for each protocol in a general purpose programming language[2]. Alternatively, we could partially define a protocol as code, and annotate that code with textual parsing rules,[3] in the same manner as the parser generator bison[23]. Or, we can take the idea of parsing rules one step further, and use a context free grammar to define the entire grammar, along the lines of Packet Types[10]. If textual parsing rules are part of the language, a parser generator must be used to transform the rules into a parser.

3.2.1 Finding Fields in Protocols

A stream can be thought of as an ordered sequence of fields, following some kind of specification (i.e. a protocol). In order to understand the stream, we must be able to locate and understand the meaning of each field.

Finding meaningful text in a protocol is easy, regardless of approach. If the protocol is defined in code, the programmer only needs to identify when areas of meaningful text begin and end, and allow library to inform the application. With a partial or full context-free grammar, the rule names of the grammar can be used to specify each field, and the parser generator can provide the appropriate notification to the library.

3.2.2 Fast Execution

Network data can be very high volume. We want our parsers to be efficient enough to handle large volumes of data with as low a cost as possible.

[2]For the sake of brevity, we will refer to any procedural or object-oriented programming language as a *general purpose programming language*. Any program, or fragment of a program, written in a general purpose programming language shall be referred to as *code*.

[3]Textual parsing rules refer to productions defined in a context free grammar, or some other specification language such as BNF or ABNF.

Any of the approaches can produce a fast parser. It is possible to write fast code, just as it is possible for the parser generator to create fast code. However, it may be difficult for a programmer to consistently produce fast code for every parser they implement. A properly tuned parser generator could produce fast execution for every parser it generates.

It makes sense to allow the parser generator to handle speed concerns, as it only needs to be optimised once. If the parser generator is to handle speed critical sections, then those sections must be implemented using grammar rules.

3.2.3 Correct Parsers

A "correct" parser behaves in the manner the programmer expects.

Any parser written by a human is likely to have bugs, and may behave unexpectedly. The more code a person writes, the more likely that the codebase will contain errors[47]. Machine generated code, however, is likely to be more correct. Because it is automatically generated, the parser will be less prone to book-keeping errors that a programmer may fall prey to.

The declarative nature of context free grammars allows an explicit explanation of the structure of the protocol, rather than the implicit specification found in code; making deviations from specification easier to detect.

Another advantage of parser generators is that their compile-time syntax checking allows for the detection of certain classes of error. This allows the programmer to detect some errors at compile time, instead of explicitly having to generate test cases to show internal consistency. Hand-written code is a stark contrast, requiring that every possible code path be tested for some level of assurance that the code does not contain (severe) bugs.

3.2.4 Parsers Must Be Easy to Write

There are hundreds of streaming protocols in existence. We wish to simplify the task of writing parsers for each protocol.

"Simple" is a question of preference. Some programmers may find a special purpose protocol specification language attractive, because it allows them to ignore the complexities imposed by general purpose languages. Others may find the use of a protocol specification language limiting, because it cannot perform tasks that would be straight-forward in a general purpose language. Regardless of preference, writing stream parsers by hand is complex and error prone, as discussed in Section 2.1.3.

A context free grammar to describe a protocol should be much more compact than general purpose code describing the same protocol.

Ideally, the grammar should be derivable directly from the protocol specification document (assuming that the specification defines the protocol in a well-known grammar), possibly reducing the task of parser specification to a cut and paste.

3.3 Description of the NPG

Given the requirements in the previous Section, the author decided that a declarative grammar was the best approach to defining application protocols for network monitoring tools. As such, he has created the Network Protocol Grammar.

In this section, we describe the syntax of the NPG. An implementation of a parser generator for languages specified in NPG will be discussed in Chapter 4.

We have chosen to base the Network Protocol Grammar on ABNF[14], primarily for backward compatibility with existing protocol definitions. Since ABNF is already used in many protocol specifications, we can specify parsers for those protocols simply by copying the ABNF definition out of the specification, and into a file. As we saw in

Section 2.2.5, there are a number of languages that can elegantly describe dynamic elements. NPG borrows syntax from the Packet Types[10] specification language. Packet Types contains a structure that expresses relationships between elements in the input text; which we can easily repurpose. To facilitate grammar definition, we also add a few syntactic structures that simplify a specification author's life.

Section 3.3.1 describes the ABNF basis of the NPG. Section 3.3.2 describes the additions to ABNF that have been derived from Packet Types, and the additional syntactic structures that ease the lot of NPG users. We justify these decisions in Section 3.4.

3.3.1 Review of ABNF

Rules

The basic element of the ABNF grammar is a rule, which is identical to the rules of phrase structure grammar. In ABNF, a production is specified with a rule name, followed by an equals sign, followed by a set of elements:

$$name = element_1 \ ... \ element_n$$

Elements are anything that can appear on the right hand side of a rule. They consist of terminal values and nonterminals.

Constants

Constants are terminal values. Terminals are sequences of characters, specified as a case insensitive string (in quotes, as most programmers would expect), or as numeric constants.

Numeric constants can be specified as a series of alternatives or ranges. A constant

is specified with the prefix `%b`, `%d`, `%h`, or `%o`. Each prefix specifies a different base for the following number, either binary, decimal, hexadecimal, or octal. The values following can either be:

- a single number in the base specified, e.g. `%b1010`, `%d10`, `%h0a`, `%o12`, all of which match an ASCII newline.
- a set of values, each specified by a number, separated by dots, e.g. `%d50.51.53.55`, `%h32.33.35.37`, `%o62.63.65.67`, which match any of the ASCII characters 2, 3, 5, and 7.
- a range of values, whose upper bound and lower bound are separated by a "-", e.g. `%d97-122`, `%h61-7a`, `%o141-172`, which match any ASCII character between "a" and "z".

Constants match when the input text contains a character with the given value. Numeric constants match against characters with the same number, while strings match against an ordered set of characters in the input (regardless of case).

Figure 3.1 provides an example of rules and constants in ABNF.

```
rule1 = "abc"
rule2 = %d65 %h61-63 %o130.132
```

Figure 3.1: An example of ABNF. The example contains two rules. The first, <`rule1`> matches the text "*abc*", "*abC*", "*aBc*", "*aBC*", "*Abc*", "*AbC*", "*ABc*", or "*ABC*". The second rule, <`rule2`> matches "*AaX*", "*AbX*", "*AcX*", "*AaZ*", "*AbZ*", or "*AcZ*".

Rule References

The elements in the right hand side of a rule can refer to other rules. Unsurprisingly, a rule is referenced through the use of its name. Figure 3.2 contains an example of a

rule reference.

```
rule1 = "(" rule2 ")" %d33
rule2 = "abc"
```

Figure 3.2: A rule reference in ABNF. The reference is the second element of the production <rule1>. Assuming that <rule1> is the starting rule, the grammar will match "*(abc)!*", "*(abC)!*", "*(aBc)!*", "*(aBC)!*", "*(Abc)!*", "*(AbC)!*", "*(ABc)!*", or "*(ABC)!*".

Sequences

A sequence is a series of symbols in the right hand side of a rule that must match consecutively for the rule to match. Like the phrase structure grammar, ABNF uses white space as the sequence operator, meaning that a list of symbols separated by spaces defines a sequence. Figures 3.1 and 3.2 contain sequences.

In order for the sequence to match, every symbol in it must match against the input, in order.

To explicitly create a sequence (to control alternation or repetition, for example), any number of symbols may be wrapped in round brackets.

Alternation

Alternatives provide a set of possible values that an input may contain. In order to for the alternative to match, any one of the possible values must match. Each possibility is separated by a "/". Alternation has lower precedence than sequences, meaning that individual alternatives can contain multiple elements, as seen in Figure 3.3.

We can use recursion and alternation to define grammars with balanced start and stop features, as seen in Figure 3.4.

```
rule1 = "a" "b" "c" / "xyz" / rule2
rule2 = "123"
```

Figure 3.3: A rule reference in ABNF. Assuming that <`rule1`> is the starting rule, the grammar will match any case permutation of: "*abc*", "*xyz*", or "*123*".

```
brackets = arguments / "(" brackets ")"
arguments = "arg, " arguments / "lastArg"
```

Figure 3.4: A recursive grammar. Assuming the grammar starts with <`brackets`>, we will match a balanced number of brackets around an argument list consisting of one or more instances of the text `arg,` and finishing with a `lastArg`. Examples include: "*(lastArg)*", "*(((arg, arg, lastArg)))*", or "*((arg, lastArg))*".

Repetition Operators

Every structure of ABNF we have seen so far is a mapping from the phrase structure grammar described in Section 2.2.2. ABNF possesses two operators that we have not seen: the repetition operator, and the optional sequence operator.[4] These operators do not increase the expressiveness of the language, but they do make ABNF grammars easier to to read.

The *repetition operator* indicates that the associated element should repeat a specific number of times. It is a prefix operator denoted by a star (`*`). The operator takes two operands: a prefix minimum, and a postfix maximum. All counts dealing with the operator are inclusive.

To specify that a character `r` should occur between three and five times, the following block of NPG could be used: `3*5"r"`, which would match "*rrr*", "*rrrr*", or "*rrrrr*". Note that the more verbose, and less readable syntax `"rrr" / "rrrr" / "rrrrr"` could also have been used.

[4]These operators are the intellectual offspring of the *Kleene star*[22] operator. Operators styled on Kleene star exist in many languages.

Either operand may be omitted. If the minimum is discarded, it is an implicit zero. If the maximum is omitted, it is an implicit infinity. If both minimum and maximum are omitted, the entity may occur zero or more times.

ABNF also provides an *optional sequence* operator. Like the repetition operator, the optional sequence operator changes how often an element can occur, in this case, between zero and one times. The optional sequence operator is `[]`, with the element list between the square brackets. The optional sequence operator is functionally identical to `0*1()`. In order to match zero or one occurrences of the string "kittens", we could use either `["kittens"]` or `*1"kittens"`.

3.3.2 The Network Protocol Grammar

Our goal with this thesis is to provide a tool for monitoring the application layer of network interactions. We wish to extend network monitoring out of the network and physical layers, and into the application layer. To analyse the application layer, we need a succinct way of describing application protocols that contain dynamic elements. The Network Protocol Grammar is a context free grammar to do just that.

We derive most of the NPG's structures from ABNF. We use ABNF's rules, constant definitions, sequences, alternatives, and repetition operators without change. We support dynamic elements by borrowing Packet Types' `where` clause, and expressions.

The Packet Types `where` clause is used to express relationships and constraints, limiting the datagrams that a Packet Types grammar will recognise. Packet Types is designed in this manner because it is used for datagram recognition. When writing a Packet Types grammar, the goal is be able to detect syntactically correct instances of protocols. Our goal is different: we wish to parse streams containing dynamic elements.

Logical Model of the NPG

The NPG provides a model for grammars. A grammar consists of a series of rules. Each rule has two attributes, its minimum cardinality, and its maximum cardinality. If a rule is a terminal, it has an additional attribute: the text it matches. This model is identical to that of ABNF.

NPG differs from ABNF by making attributes mutable. For terminals, all three attributes are mutable. For nonterminals, only the minimum and maximum cardinality is mutable.

In turn, zero or more nonterminals are annotated with assignment operations that can modify attributes. When an annotated nonterminal matches the input, it changes one or more of the attributes of the rules. The modification made to the attribute(s) is a function of the text that the nonterminal matched.

Introducing the `set` Clause

To express the relationship between nonterminals and the attributes they change, we introduce the `set` clause to the NPG. Any rule can be annotated with a `set` clause. The `set` clause contains zero or more statements that define which rules' attribute is changed, the nonterminal the change depends on, and the function that determines the change.

Syntax of the `set` Clause

The `set` clause exists at the end of a standard NPG rule. It begins with the string `set`, and contains the list of statements between braces.[5] Each statement consists of a left hand side and a right hand side.

[5]In order to promote consistency, we also allow the elements of the rule to be wrapped in braces, but that is a purely cosmetic change.

The left hand side identifies the rule and attribute that is modified by the statement. The rule is identified by its name. Attributes are identified by the constant strings `text`, `repetition_min`, or `repetition_max`. A hash (`#`) is used to join the rule name and the attribute string.

The `text` attribute is used to modify the text of rules that contain a terminal. It may not be used with rules that contain nonterminals.

The `repetition_min` and `repetition_max` attributes change the cardinality of rules. The change is global across all uses of the the rule. `repetition_min` is used to change the minimum number of times that a rule may match, while `repetition_max` is used to change the maximum number of times that a rule may match. In situations where grammar authors may wish to change both attributes simultaneously, they may use the pseudoattribute `repetition`, causing both `repetition_min` and `repetition_max` to be modified at the same time.

```
text = length value
set {
     value#repetition_min = int(length);
     value#repetition_max = int(length);
}

length = 1*%d48-57
value = %d97-122
```

Figure 3.5: Grammar G_{set} provides a sample use of the `set` clause. The grammar, rooted at <`text`> will match any input text that has some number n, followed by n lower case ASCII characters. For example, it will match "*4four*", "*2ab*", "*20internationalization*", or "*0*".

Meanwhile, the right hand side references one or more nonterminals used in the rule. The nonterminals may have functions applied to them, or they may remain bare. Figure 3.5 shows a grammar G_{set} that uses a `set` clause.

Rules referenced in the right hand side of a `set` statement must occur exactly

once in the associated rule.

Semantics of the `set` Clause

G_{set} from Figure 3.5 creates two annotations in the logical NPG model. Both annotations are attached to the <`length`> rule reference in <`text`>. The target of the annotations are the repetition attributes of the <`value`> rule.

When G_{set} is matching an input string i, there are four events of interest that occur:

- The <`length`> rule is accepted, meaning that it has matched one or more ASCII characters representing integers.
- The string that <`length`> matched is fed to the function `int()`, which produces an output *out* compatible with the `repetition_min` and `repetition_max` attributes of <`value`>.
- The `repetition_min` and `repetition_max` attributes of <`value`> are updated to reflect *out*.
- A character starting <`value`> is received. Because the cardinality of <`value`> has been set to *out*, <`value`> will not match until exactly *out* characters have been received.

Scoping the `set` Clause

Nonterminals referenced in the right hand side of the `set` clause are scoped to the rule in question. This allows nonterminals to be used in multiple rules, without unpredictable effects.

Consider the grammar G_{multi} showing in Figure 3.6. In the example, G_{multi} the grammar matches some number, followed by zero or more space separated numbers, and then a string. The length of the string is specified by the first number.

```
multi = len multiLen value
set {
     value#repetition_min = int(len);
     value#repetition_max = int(len);
}

multiLen = *(" " len)

len = 1*%d48-57
value = %d97-122
```

Figure 3.6: Grammar G_{multi}. The grammar matches "*20internationalization*", "*20 1 2 3internationalization*", "*0 10 10 10*", "*0 10 10 10*", or "*2 50 1ab*".

G_{multi} uses <len> to describe all of the numbers in the input text, but the only instance that sets <value>'s attributes is the one in <multi>.

The same scope limit does not apply to the attribute target. When a rule/attribute pair is specified on the left hand side of a statement, every time the rule is used, it will match in a modified form.

Syntax Changes

During our experience using the Network Protocol Grammar, we discovered a number of usability issues with the ABNF-derived portions of the grammar. Notably, case-sensitive text is hard to define, and there is no mechanism for supplying a starting rule for the grammar. We make two additions to NPG to handle these problems.

ABNF forces users to define case sensitive text using character ranges. So, to define the case sensitive string "kitten", we must specify the ASCII character code for each byte individually, yielding: `%d107 %d105 %d116 %d116 %d101 %d110`. Such

constructs are difficult to read. We add a new string type: a *case sensitive terminal*, which consists of the string, surrounded by double quotes. To differentiate a case sensitive string from a case insensitive string, we append the character `c` onto the last quotation mark. The string "kitten", using our case sensitive terminal is a much more readable `"kitten"c`. Note that this change does not effect NPG's ability to read standard ABNF.

Although the ABNF standard indicates the structure of a grammar, it does not supply a method to indicate which is the root of the grammar. NPG sets the first rule in the grammar to be the topmost rule.

3.4 Design Decisions

The NPG did not spring from its creator's head fully formed. Instead, it was built iteratively. At various points the author considered different paths. This section describes possible paths, and why the chosen path was selected.

3.4.1 Divorcing Behaviour and Grammar

Some parser generators, such as `bison`[23] and `yacc`[34] force grammar authors to annotate the rules in their grammars with code that specifies activities that should occur when the given input is received. Other grammar description languages, such as XML, eschew that approach, forcing programmers to subscribe to rules in the programming code that uses the grammar.

The NPG is intended to be used as a general specification language, that is not bound to individual applications, or programming languages. Including code in a grammar definition would limit NPG's portability.

3.4.2 Choosing the Base Grammar

Since we did not want to create the NPG from scratch, we considered existing protocol definition languages as a starting point. Our intent was to extend an existing context free grammar. We considered three possible grammars to use as a base. The first, ABNF, is a standard context free grammar, meaning that it suffered the failings described in Section 2.2.4. The other two grammars are packet filter description languages that already contain syntactic structures addressing the problems raised in Section 2.2.4.

ABNF is a popular context free grammar, used to define protocols in many RFCs. It is close to the "textbook" context free grammars that most programmers encounter in school, meaning that most programmers can read and understand ABNF with little or no effort.

Our second candidate was Packet Types, described in Section 2.2.5. Unlike ABNF, we were unable to find references to the language outside of academia. It is also based on a context free grammar, but that grammar associates a type with each symbol used in a production; producing grammars that bear a resemblence to C-style structs.

Packet Types is notable because it provides syntax to handle dynamic elements, meaning that if the NPG uses it as a base grammar, we would be able to use the dynamic element handling portion verbatim. However, Packet Types is a filter specification language, its primary goal is winnowing packets. Its alternation syntax is limited: it does not allow inline alternatives in a rule, instead, it introduces a special class of rule that may match any one of a set of alternatives.

Our third candidate was APF. APF provides a `bison`-like syntax. Although APF is also a filter specification language, it is much closer to a standard phrase structure grammar. Its lack of alternation syntax could easily be remedied. APF does not appear to be used outside of Lambright's original paper, meaning that any protocol

definition in an APF-based language would have to be ported from ABNF.

When deciding which grammar to use as a template, ABNF's popularity won the day. Many protocols are defined in ABNF, meaning that very little work should have to be done to produce NPG definitions. For protocols that don't require NPG's advanced features, the specification can be used without modification. Protocols that require NPG's more complex field terminators would only require minor additions.

3.4.3 Choosing a Syntax for Describing Dynamic Elements

Protocol recognition literature provides two examples of ways to handle dynamic text: the Packet Types `where` clause, or the embedded expressions used in APF.

In order to get a rough idea of the appearance of these two idioms, we compare how they represent an IP datagram. The APF representation is shown in Figure 3.7, and the Packet Types representation is shown in Figure 3.8. The examples are adapted from Lambright's APF paper[38], and the Packet Types papers[10, 40].

APF's expressions are less structured, able to appear anywhere in the production. Packet Types has a more formal approach, forcing the assignment of dynamic attributes into a clause added to the production. Of the two approaches, the author prefers Packet Types, as it enforces more structure.

Within structures, APF references productions by index. In Figure 3.7, the <`ihl`> production is referenced with a $2 and the <`totallength`> production is indexed with $4. Packet Types uses the much more human readable approach of referring to symbols by name.

The author chose to use the Packet Types the approach to dynamic elements, as it is more readable and structured.

```
IP_PDU     :  version ihl {optLen = ptoint($2)*4 - 20;} tos totallength
                {payloadLen = ptoint($4) - ptoint($2)*4;}
                id morefrags dontfrag unused offset
                ttl proto checksum
                src
                dst
                options
                payload
;

version         :  bit*4 ;
ihl             :  bit*4 ;
tos             :  byte ;
totallength     :  byte*2 ;
id              :  byte*2 ;
morefrags       :  bit ;
dontfrag        :  bit ;
unused          :  bit ;
offset          :  bit*13 ;
ttl             :  byte ;
proto           :  byte ;
checksum        :  byte*2 ;

src             :  byte*4 ;
dst             :  byte*4 ;

options         :  byte*optLen;

payload         :  byte*payloadLen;
```

Figure 3.7: A specification an IP packet using APF. Text highlighted in bold denotes syntactic structures specifying dynamic length. The $2 and $4 in the expressions are references to symbols in the <IP_PDU>. The indexes start at 1, and refer to <ihl> and <totallength>, respectively.

3.4.4 Designing the NPG Logical Model

The NPG logical model, described in Section 3.3.2, was derived from the Packet Types grammar. The entities in the logical model map directly to entities in the grammar. Because the model exhibits the behaviour we wish NPG to express, possesses no

```
nybble := bit[4];
short := bit[16];
long := bit[32];

IP_PDU := {
     nybble    version;
     nybble    ihl;
     byte      tos;
     short     totallength;

     short     id;
     bit       morefrags;
     bit       dontfrag;
     bit       unused;
     bit       offset[13];

     byte      ttl;
     byte      proto;
     short     checksum ;

     long      src;

     long      dest;

     ip_option options[];
     payload   payload[];
} where {
     options#numbytes = ihl#value * 4 - 20;
     payload#numbytes = totallength#value - ihl#value * 4;
}
```

Figure 3.8: The specification of an IP packet using Packet Types. Text highlighted in bold denote syntactic structures providing dynamic length.

discernible drawbacks, and no alternatives made themselves clear, we did not examine any other approaches.

`value` The natural number formed by concatenating all the bits of the field in network order.

`numbits` The total number of bits occupied by the field.

`numbytes` The total number of bytes occupied by the field.

`numelems` The number of elements in an array-type field.

`alt` For an alternative-type field, a collection of booleans indicating which alternative was chosen.

Figure 3.9: Packet Types attributes, originally presented in [40].

3.4.5 Choosing Rule Attributes

Packet Types provides five attributes for rules, shown in Figure 3.9. We chose to ignore those attributes for a number of reasons.

Packet Types Attributes

The `numbits`, and `numbytes`, fields are not appropriate for the NPG, as the NPG is byte oriented. Similarly, `numelems` has no direct application to the Network Protocol Grammar because our language has no concept of arrays. Instead, we provide the same functionality with the `repetition_min` and `repetition_max` attributes. These fields are more general, able to describe ranges of valid repetitions, instead of a specific number of repetitions.[6]

The `value` attribute is sensible when querying a rule for the text it matches, as Packet Types does in its assertions. But the NPG exists in a different space: we wish to assign a value to a rule, instead of verifying it. Furthermore, we can only assign to terminals. We wish to make the relationship between assignment and the type of rule clear, so we use the name `text`. To the author, `text` implies the only type of

[6]Such ranges could be useful for cryptographic protocols that require a minimum and maximum key length, but whose key length may be variable due to compression, or other vagaries.

symbol that possesses a text value: terminals.

The `alt` attribute in Packet Types is used to determine which branch of an alternation matched. Packet Types contains syntactic structures to layer protocols on top of one another. Those structures use the `alt` attribute to set the type of a datagram's payload.[7] NPG does not currently provide features for protocols to be layered on top of one another, and therefore does not use this feature.

Selected Attributes

The author chose NPG's attributes based on their mapping to concepts within context free grammars. Cardinality attributes are derived directly from ABNF's repetition operators, which allow for the specification of upper and lower bounds for cardinality. Similarly, the `text` attribute was chosen to rewrite a single terminal.

Scale of Grammar Modification

A more general mechanism for exposing the grammar to modification could modify a grammar arbitrarily at parse time, adding or removing entire alternations and sequences, or adding new rules. We did not follow this route of redefinition because we wanted to limit the scope of modification from what is possible, to what is useful and achievable. In the protocols we have surveyed so far, the selected attributes are sufficient to allow parsing.

3.4.6 Choosing Scoping Rules

The Packet Types papers do not specify scoping rules. Instead, the author has created the two scoping rules. We address each scoping rule separately

[7] If other roles are intended for this feature, the author was unable to find them in [10, 40].

Scoping Attribute Assignments From the `set` Clause

Recall that nonterminals in the right hand side of `set` statement must occur in the associated rule. If the nonterminal occurs in any other rule, it is ignored for the purposes of the attribute assignment. We can say that the expressions in the `set` clause have local scope.

An alternative to locally scoping attribute assignment is global scoping. A global scope would cause *any* reference of the given nonterminal to modify an attribute in a `set` clause. The author feels that global scoping is unintuitive and error prone for attribute assignments. Consider Figure 3.10.

```
message = {
     "Sentinel:  " AlphaNum NL
     *1Headers NL
     Payload NL
     End
} set {
     End#text = string(AlphaNum);
}

AlphaNum = 1*(%d65-90 / %d97-122 / %d48-57 / "-")

Headers =      "From:  " AlphaNum "@" AlphaNum NL
               / "To:  " AlphaNum "@" AlphaNum NL
               / "X-" AlphaNum ":  " AlphaNum NL

Payload = AlphaNum / " " / NL

NL = %d10

End = "ignore"
```

Figure 3.10: Grammar G_{scope}. G_{scope} misbehaves when attribute assignment is globally scoped.

G_{scope} will do two very different things when run under local or global scope. Un-

der local scope, it will use the first <AlphaNum> in <message> to define a terminator for <Payload>. Under global scope, the terminator <End> will be modified whenever <AlphaNum> matches input; meaning that matches in <Header> will cause the terminator at the end of <Payload> to change.

Although this example is trivial, it highlights the drawback of globally scoped statements: a rule that modifies an attribute through a `set` statement must not "accidentally" be used anywhere else in the grammar. For this reason, the scope of attribute assignment is limited to the rule where it is defined.

Scoping the Use of Modified Rules

When a rule is modified with the `text` attribute, every instance of that rule throughout the grammar will match the new value. The modified rule has global scope. Local scope would only allow the modified rule to appear in the rule where it was modified, all other uses of the rule would match the original definition in the grammar.

Global scope is superior for rule use because the attribute assignment may be buried in part of the grammar that cannot appear in the same production as the reference to the modified rule. Consider a scenario where one of many headers, which may appear in any order, sets the length of a field following the headers. Under local scoping, both the header and the field would have to appear in the same rule. Under global scoping, the header and the field can occur in different rules, improving modularity of the grammar.

3.4.7 Layering Grammars

One of the features of Packet Types that we chose not to implement was protocol layering. Packet Types allows grammar specifications to be built on top of each other. So, for example, the specification for a TCP datagram can be layered onto the

payload portion of an IP datagram. This practice occurs at the application layer as well, some application layer protocols are built on other application layer protocols.

For example, the Internet Printing Protocol[29] can use HTTP to transport information. When IPP runs over HTTP, it sets some fields in the HTTP protocol in a specific manner, and transfers data in the HTTP payload.

We chose not to provide grammatical hooks for protocol layering for due to complexity. The line between the transport layer protocol and the higher level protocol may be fuzzy: data may be carried outside of what one considers the payload-bearing portion of the protocol. Instead, we opt to leave that problem to be addressed in somebody else's thesis.

3.4.8 The Case Sensitive Terminal

The case sensitive terminal, although not necessary, makes grammars much easier to read. When defining the case sensitive terminal, the author decided that the string had to be shown verbatim in the grammar. The only decision left was to decide on the delimiters to use for the string.

We chose not to use special characters, such as backticks (`) or apostrophes (') as delimiters, as ABNF already uses many punctuation characters. We feel that using yet another punctuation character would limit the scope of future modifications. Instead, we chose to use the existing double quote (") delimiter, with a suffixed argument "`c`". This follows the Perl idiom of appending modifiers to regular expressions. For example, the Perl regular expression `/search/` can be made case insensitive by appending an `i` to the terminating slash: `/search/i`.

This approach is extensible, allowing us to add other arguments in future. It maintains consistency with ABNF. It does not consume any more delimiters.[8] The

[8]The author was disappointed to discover that square brackets are used by ABNF to delimit

approach should also be familiar to anyone who has written more than a few lines of Perl.

3.4.9 Selecting a Starting Rule

We chose to use the first rule of an NPG grammar as the starting rule because it should be the first rule visible when opening the grammar file. We easily could have given the rule a special name, added a special delimiter to the rule, or chosen some other distinguished position.

Although a special name would be an acceptable, it is not necessarily easy for grammar authors to deal with. If the special name is somehow related to the file name (similar to Java's class naming approach), then the grammar author is forced to modify the grammar whenever the grammar's filename changes. If the special name is constant across all grammars (similar to C's `main()` function), then we make life difficult for anyone who wishes to combine grammars, as there is guaranteed to be at least one rule naming conflict.

We chose not to use a special delimiter for the start rule because that would add complexity to the grammar of the NPG. Instead, we would rather modify the semantics of the language.

The only other distinguished position the author considered was the last rule in the grammar. We chose to avoid the bottom because it would require the user to scroll to the end of the file to find it.

optional regions. Perl, and other regular expression languages use square brackets to delimit ranges of legal characters verbatim.

Chapter 4

The qcap Library

Chapter 3 presented a grammar for specifying network protocols in a machine-readable manner. This chapter builds on that work, presenting a tool for network analysis called `qcap`. `qcap` is a network monitoring library that provides applications with access to each level of the TCP/IP protocol stack, as well as streaming application protocols running on TCP. Stream parsers are generated from NPG specifications of protocols.

In this chapter, we describe `qcap`, starting from basic requirements, and working our way into a description of the library, as well as design and implementation concerns. We will begin with an explanation why `qcap` is necessary.

4.1 Motivating qcap

A quick survey of existing libraries (netdude[36], pcap[56], and CoralReef[9]) shows network analysis libraries concentrate solely on datagram oriented protocols. Newer libraries (netdude and CoralReef) provide hooks for protocol-specific datagram anal-

ysis, but provide no stream-level functionality.[1]

The lack of stream reconstruction libraries means that any researcher who wishes to explore application streams must build their analysis tools from the IP layer to the application layer. Their tools must defragment IP streams, reconstruct TCP sessions, and parse the application stream in question. TCP/IP reconstruction is not trivial, and differs depending on the endpoints in question[54], adding a further layer of complexity.

qcap is designed to be a foundation for tools that need access to some level of the network stack. It emulates the network, transport, and application portions of a network stack, and provides access to all parts of the reconstruction process.

4.2 Requirements for qcap

When designing the Network Protocol Grammar, we wanted to be able to describe protocols in a way that is concise, minimises errors, and easy to use. For qcap we map those requirements in the context of software design.

qcap must be fast. Fast network reconstruction means that large volumes of network data can be rebuilt quickly. The exact definition of "large" and "quickly" will depend, of course. The author wishes qcap top be able to reconstruct a gigabyte in one to two minutes on commodity hardware.

qcap must expose high level data. The purpose of the qcap library is to reveal information contained in high-level protocols to applications.

qcap must be easy to use. We want qcap to be used by other researchers. In order

[1] libnids[59] is a partial exception: it reconstructs byte-oriented data streams from TCP/IP datagrams. In doing so, however, it discards the relationship between the higher level TCP payload, and the datagrams that carried it, making it less useful for network analysis.

for that to happen, the library must be easy for programmers to use. Defining "easy to use" is difficult, so we shall simply state that we want to qcap API to be simple, comprising of a small number of functions, whose purpose is clear.

qcap must reconstruct all network traffic. As we stated above, a valuable reconstruction tool would rebuild data from every level of the network stack, starting at the IP layer, and working up to the application layer.

qcap's reconstruction process must be transparent. When a datagram is received by a host, it will not necessarily be passed out of the kernel. It may be discarded for a number of reasons. If the datagram is discarded, we wish qcap to inform the application that a discard event occurred, and why.

Our analysis of the qcap problem domain provided many fewer options than our analysis of the NPG domain. In addition to the aforementioned requirements, we wished to get a system up and running as quickly as possible. As such, we avoided exotic approaches to the problem of creating a full network stack, and used a simple layered approach. The layers correspond to the layers of the network stack.

We delay further discussion of design issues until after we have described the qcap API.

4.3 Description of the qcap Library

The qcap life cycle is simple, falling into four steps: creation, configuration, execution, and cleanup. During creation, the application creates a qcap instance, and specifies a source for datagrams to be read from. Configuration follows, allowing the application to subscribe to specific network events, and specify pcap filters for winnowing datagrams. During execution, the application passes control to qcap allowing qcap to

trigger callbacks installed during the configuration phase. Once the datagram source has been exhausted, or the application requests that execution stop, the application tells `qcap` to clean up after itself.

4.3.1 Callbacks in `qcap`

`qcap` provides four callbacks for network events, each providing a slightly different view on the payload of network transactions. Each callback is parameterized, allowing it to match a subset of events. The application can deregister callbacks at any time.

Datagram-Oriented Callbacks

Datagram-oriented callbacks are triggered for network events involving individual datagrams, such as initial receipt, discard, or acceptance by a higher level protocol. The application can choose to be notified of datagrams as they pass through specific portions of the network stack.

During IP defragmentation, many datagrams can be merged into one larger datagram. In this situation, a *pseudopacket* is created, carrying the total data of the fragments, as well as references to the constituent packets. The pseudopacket is treated as a normal datagram for the remainder of the reconstruction process.

Segment-Oriented Callbacks

A segment is a unit of data sent across a TCP connection, equivalent to the payload of a single TCP datagram[51]. Segment-oriented callbacks operate at the stream level. They are triggered when a segment is accepted or discarded. The application may choose to ignore some classes of segment, such as empty segments resulting from empty TCP acknowledgement datagrams.

Stream Creation Callbacks

To minimise the costs of marshalling datagram payload into segments, `qcap` includes a stream creation callback. The callback is triggered whenever a stream is erected (from one side of a conversation), and allows the application to decide if it wishes to receive segment callbacks for the data in that stream.

Application-Oriented Callbacks

Application oriented callbacks are triggered whenever specific portions of TCP streams are detected by the application parser generated from an NPG definition. The application parameterises each application-oriented callback with the type of stream `qcap` should listen on, and the specific field it wishes to receive notifications about (with names derived from rules in the NPG specification).

4.4 Implementation Notes

`qcap` is built in a manner that hides the complexity of the network stack from the application. However, part of `qcap`'s internals will determine how it behaves for tests in Chapter 5, so we will provide an overview of the system.

4.4.1 Handling IP and TCP Reconstruction

`qcap` uses the open source `libnids`[59] library to perform IP defragmentation, and TCP stream reconstruction. According to the `libnids` website, it "emulates the IP stack of a 2.0.x linux kernel." A number of `libnids` internal data structures had to be modified to carry datagram information from the physical layer to the application layer of the simulation.

4.4.2 Parser

A bulk of the original work in `qcap` is found in the application layer parser generator. The parser generator is responsible for producing parsers from protocol specifications written in NPG.

`qcap` transforms grammars into nondeterministic finite automata. Each automata consists of a set of *states*, connected by *transitions*. Every transition corresponds with a legal input symbol. Two states in the automata are distinguished: one as the *start state*, and one as the *accept state*. For every protocol there need only be one automata. Many parsers can read simultaneously from one automata.

Automata in `qcap` have a number of additional features designed specifically for handling dynamic elements. First, each transition has an unordered set of *vetoes* that execute after the parser has matched a symbol on the transition. As their name suggests, vetoes prevent the parser from matching the transition, effectively "turning off" transitions. Second, each transition maintains an ordered set of *actions*. Each action acts as an instruction that is executed when the transition is matched and followed. Actions may have arbitrary behaviour, but are usually used to perform book-keeping activities, such as managing the the parser's stack. Vetoes are tested before actions execute.

Parsers are responsible for receiving input, determining if the input conforms to the grammar defined by their associated automata, and parsing the input into parts, as defined by the grammar. Each parser has a *state pointer*, that records its position in the automata. Parsers have zero or more *registers* that store parameters for dynamic elements; and zero or more *buffers* for recording input to be processed into parameters (and eventual storage in a register). In order to properly maintain the location in an automata, parsers also have two stacks: a *rule stack*, which records the grammar rules that the parser has entered but not yet left; and a *counter stack* which is used

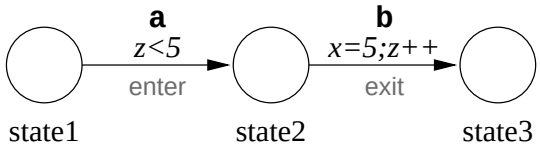


Figure 4.1: Graphical notation used to describe an automata
Notation used to describe automata. Assuming the automata is in *state1*, an input of "a", while z is less than five, will cause the automata to progress to *state2* across enter. The only valid escape from *state2* is to *state3*, on an input of "b". When the automata crosses exit, the value x will be set to 5, and z will be incremented.

when making decisions about cardinality.

Initially, the state pointer is set to the start state. When a character of input is received, the parser examines the transitions leaving the current state. If there are no transitions that match the input character, the parse fails and the input is rejected. If there is one transition that matches the input (that the vetoes do not prevent), then the parser updates its state pointer to point to the destination of that transition, and waits for the next input. If there is more than one transition that matches the input character, the parser has reached an ambiguous subgraph of the grammar – that case will be examined in Section 4.5.4. If, after the parser has followed a transition, it finds itself in the accept state, the parser reports that it has fully parsed the stream and terminates.

Some transitions are special. They represent recursive rule references in the corresponding NPG definition. When the parser crosses a transition that corresponds to entering rule *dst* from rule *src*, it pushes *dst* onto a *rule stack* (which already has *src* at the top). When it comes time to exit the rule reference, the parser checks its rule stack to ensure that *dst* is on the top of the stack. If *dst* is present, the parser pops it off, and checks that *src* is on the stack before returning to *src* if *dst* is not present, the parser tests the other outgoing transitions. In this manner, `qcap` ensures that parsers leaving rule references exit to the correct rule.

The generated parsers are able to perform as ordinary pushdown parsers and manage dynamic elements by using a combination of actions and vetoes.

The format we will use to render automata in this document is shown in Figure 4.1.

Note that the increment operations indicated in the Figure are performed by actions. Similarly, the tests are performed by vetoes. States in an automata are represented by circles. Transitions are represented by directed arrows. The character associated with a transition is rendered above the transition in a `monospace` font. If a transition has one or more vetoes associated with it, those are shown as boolean statements in *italics* over the transition. If a transition has actions associated with it, those are shown as assignments over the transition. We use C-style short hands to represent increment and decrement operations. When we need to label transitions, they are tagged with a sans serif font.

4.4.3 Handling Dynamic Elements

Dynamic elements are preceded by parameters. The parameter modifies the parser in some way, changing how the dynamic element will be parsed. All of the syntactic structures that act as parameters are defined explicitly in the right-hand-side of assignments in `set` clause of the NPG.

After the grammar has been compiled into an automata, the life cycle of a parameter/element pair is broken into four parts:

1. **detect** the parameter that specifies an syntactic feature,
2. **read** the parameter from the input stream,
3. **convert** the parameter into a form meaningful to the parser,
4. **update** the parser to reflect the parameter.

Each step is performed with a combination of actions and vetoes, stored at strategic locations in the automata.

Each parameter is explicitly linked to a rule reference in the grammar, which means that **detection** is the simple matter of annotating the transitions leading into the rule reference, and out of the rule reference. The entry transition tells the parser to start record subsequent input into buffer b. The exit transition tells the parser to stop recording into b, and to **read** the parameter out of the buffer. The **conversion** step causes the parser to execute the NPG expression on the right-hand-side of the assignment. The parser then **updates** itself by storing the result in a register associated with the left hand side of the assignment.

The exact nature of the register depends on the type of dynamic element in question. If the element is a dynamic cardinality, then the register is pair of integers, one representing the lower bound, and the other representing the upper bound. If the element is dynamic text, then the register is a variable length string. Every dynamic element has a different register, allowing an arbitrary number of parameters to modify the parse.

4.4.4 Parsing Dynamic Cardinalities

After a parameter has been stored into a register, it is ready for use by the parser. The upper and lower limits of cardinalities are stored into integer registers and consulted by vetoes and actions when the parser deals with dynamic cardinalities. The actions maintain a counter stack by performing push, pop, and increment operations. Vetoes use the topmost value on the counter stack, and the value of registers to decide which transitions should be active.

Parsing dynamic cardinalities has two components: parsing minimum cardinalities and parsing maximum cardinalities. `qcap`'s behaviour for each is analogous, so we

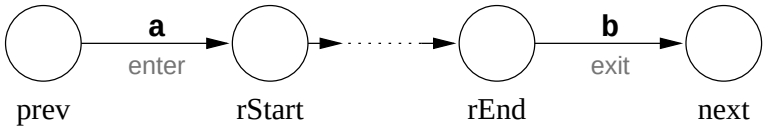


Figure 4.2: An automata that does not handle dynamic cardinality
An automata that we will add dynamic cardinality to. *rStart* and *rEnd* denote the first and last node we wish to repeat.

will start by discussing how `qcap` handles maximum cardinalities.

Before we consider a maximum cardinality, we examine the original automata, shown in Figure 4.2. The automata consists of the region we wish to repeat, delimited by *rStart* and *rEnd*, and the states before and after the region: *prev* and *next*, respectively. *rStart* and *rEnd* could be one node or many. In this example, *rStart* and *rEnd* represent start and end of a rule r.

Handling Maximum Cardinality

We want to assign an upper limit of u for repetitions of r, meaning that r can occur at most u times. But we do not know the value of u when we generate the automata, so we must consider every possible value of u that will change the structure of the automata. There are two cases of interest.

If u is zero, then *rStart* and *rEnd* will be skipped entirely. We add a transition named skip leading from *prev* to *next*, and give it the same symbol used to exit *rEnd*, in this case b. We add a veto to skip, to ensure that the value of u is zero. In order to ensure that enter does not inadvertently match, we add a veto to enter stating that u must be greater than zero.

If u is some value greater than one, then it will be necessary to cycle back from *rEnd* to *rStart*. In order to be able to count the number of repetitions, we create a counter *cu*, and initialise it in enter. To cause r to repeat, we add a transition named repeat to the grammar, and give it the same symbol as the enter transition:

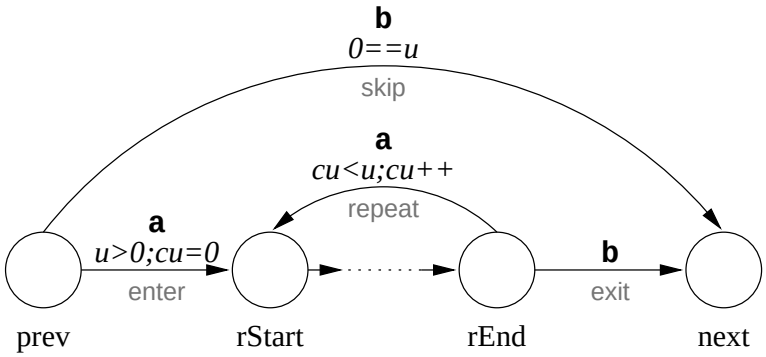


Figure 4.3: An automata modified to handle a maximum dynamic cardinality
The structure of an automata that handles dynamic maximum cardinality.

in this case, a. To prevent repeat from causing the region to match more often than it should, we add a veto limiting cu to be less than u.

After these changes, we arrive at the automata shown in Figure 4.3. An example parse of a maximum-respecting automata is shown in Figure 4.5.

Handling Minimum Cardinality

Now that we have an automata that can handle maximum cardinality, we consider minimum cardinality. We want to set a lower bound l for our rule r to repeat. If r matches less than l times, we will not allow the parser to leave the portion of the automata that maps to r. After r has matched l or more times, we will permit the parser to leave the region mapping to r.

The only way out of the region is through the transition exit. We prevent the parser from following exit at inappropriate times by adding a veto that checks if the repetition counter cu is greater than or equal to l.

Figure 4.4 shows an automata modified to handle an upper and lower bound for a region.

Figure 4.5 shows an example of dynamic cardinality in a simple automata.

Figure 4.6 shows the same parse, but with an invalid input string.

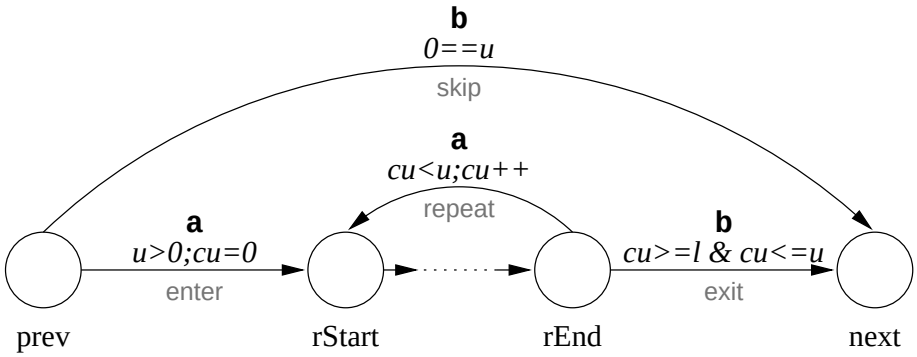
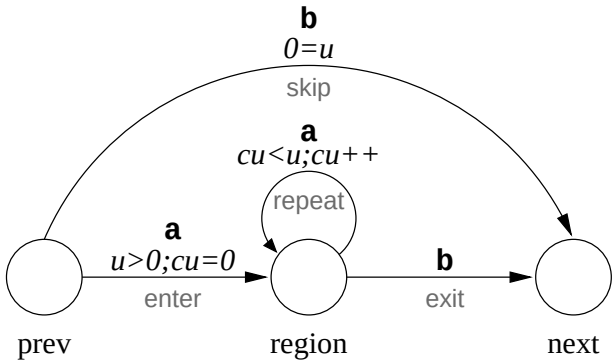


Figure 4.4: An automata modified to handle a minimum and maximum dynamic cardinality
The structure of an automata that handles both dynamic minimum and maximum cardinality.

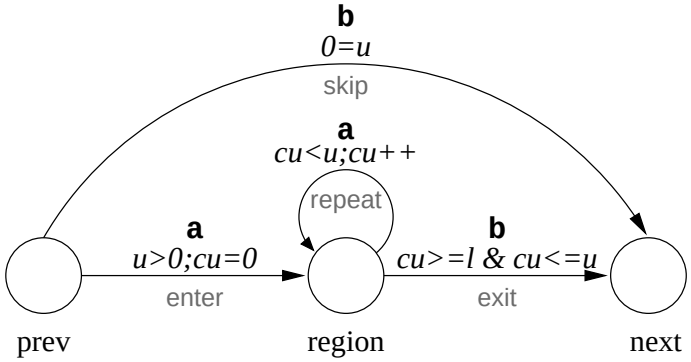


Register	Input	Current State	Events
$u = 2$ $c_u = 0$	a	*prev*	P follows enter. Increments c_u. Sets current state to *region*.
$u = 2$ $c_u = 1$	a	*region*	P follows repeat. Increments c_u. Sets current state to *region*.
$u = 2$ $c_u = 2$	b	*region*	P follows exit. Sets current state to *next*.

Figure 4.5: Example of parsing a dynamic cardinality. Register u is the upper bound on the cardinality for the only region. Register c_u is the counter used by the dynamic element. The input string is "*aab*", and the automata being parsed is shown above.

Maintaining the Counter Stack

Each counter resides on the counter stack. Counters are pushed onto the counter stack when the parser enters a dynamic region. Once in the region, all vetoes and



Register	Input	Current State	Events
$u = 3$ $l = 3$ $c_u = 0$	a	*prev*	P follows enter. Assigns 0 to c_u. Sets current state to *region*.
$u = 3$ $l = 3$ $c_u = 1$	a	*region*	P follows repeat. Increments c_u. Sets current state to *region*.
$u = 3$ $l = 3$ $c_u = 2$	b	*region*	P cannot find an exit from *region*. repeat will not match because the input b does not match a. exit will not match because *cu* is less than l. P rejects the input.

Figure 4.6: Example of the parser rejecting an invalid input string on a dynamic cardinality. The configuration of the parser is similar to that of Figure 4.5, but it has a lower bound of three repetitions in the region,. The input string is "*aab*".

actions operate on the topmost counter value. When the parser leaves the region, it pops the counter off the top of the counter stack.

So far, we have left the stack maintenance actions out of our automata for space reasons. Figure 4.7 displays the push and pop operations. All reads from the counter *cu* should be considered peek operations onto the top of the counter stack.

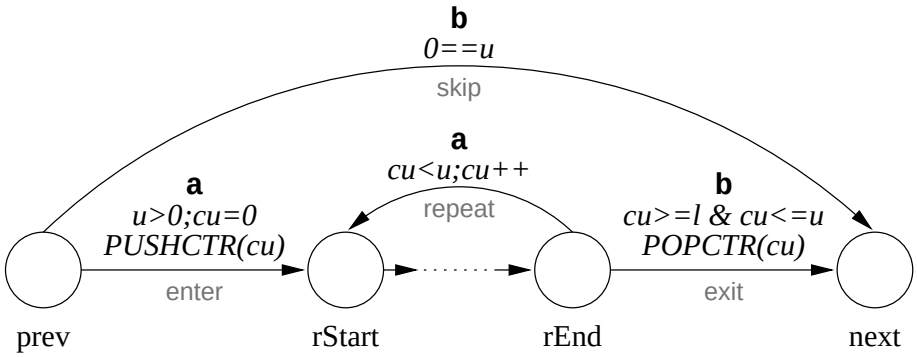


Figure 4.7: An automata modified to handle a dynamic cardinality, and maintain the counter stack
The complete structure of an automata that handles dynamic cardinality, and counter stack maintenance.

4.4.5 Dynamically Changing Text

Handling dynamic text is easier than handling dynamic cardinality. Every parser P is given two modes: a "normal" mode for parsing input using the automata, and a "register" mode for parsing input using the contents of a special text register. When in normal mode, P behaves exactly as we have discussed. When in register mode, P parses from a string stored in a special register.

Register mode exists entirely outside of the automata. It causes the parser to stop in the middle of a transition, and attempt to read the dynamic text. As subsequent inputs are read, the dynamic text is checked, bypassing the automata. If the text matches, then the parser returns to the transition it stopped in, and follows the remainder of the transition to the next state. The parser stops in the middle of the transition because there may be actions that need to be executed between the acceptance of the text in the register, and the parser's entry into the subsequent state.

Similar to dynamic cardinalities, we have to make modifications to the automata. We must consider two cases when handing dynamic text: that the text is empty, or that it contains more than one character. If the text is empty, we provide a transition

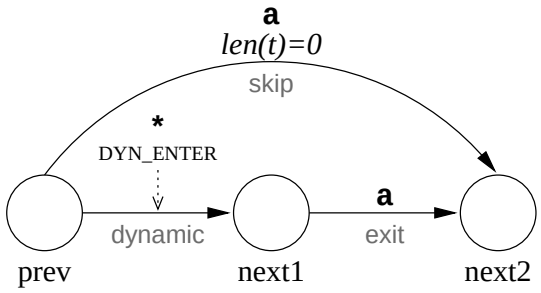


Figure 4.8: An automata that parses dynamic text
An automata that parses dynamic text. The asterisk (*) on the dynamic transition indicates that transition will match any input, deferring the decision to the DYN_ENTER action.

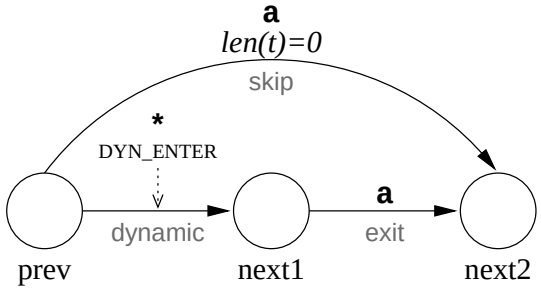
named skip that allows us to avoid the dynamic text, which may only be followed if the text length is zero. We also consider the case where the text contains one or more characters. In that case, we annotate a transition named dynamic with the special DYN_ENTER action, and connect it to the next state as if it were a normal transition.

The DYN_ENTER action switches P from normal mode into register mode. In register mode, P maintains a pointer to the register containing the dynamic text, and a counter for its progress through the text. It also tracks the transition that contains the DYN_ENTER annotation. When the counter reaches the end of the text, the parser drops out of register mode, sets its current state to be the destination of the DYN_ENTER transition, and continues executing.

Figure 4.9 illustrates a successful parse.

4.5 Design Decisions

The overall architecture of qcap is simple. It is designed as a layered network stack. In designing and building the library, an emphasis was put on creating a useful application level parser, rather than concentrating on design of lower level layers.



Register	Input	Current State	Events
$t =$ "xy" $pos_t = 0$	x	*prev*	P follows dynamic. DYN_ENTER halts execution, and switches to register mode. P increments pos_t, and tests to see if $pos_t = len(t)$.
$t =$ "xy" $pos_t = 1$	y	dynamic	P tests $t[pos_t] = y$. Since it does, P stays in register mode. P increments pos_t, and tests to see if $pos_t = len(t)$. Since it does P drops back to normal mode, sets $pos_t = 0$, and sets the current state to $next1$.
$t =$ "xy" $pos_t = 0$	a	*next*	P follows dynamic. Increments c_u. Sets current state to $next2$.

Figure 4.9: Example of a parser P accepting "xya". The text in the dynamic text register t is "xy".

4.5.1 Discarding the Tokenizer

Some parsers have preprocessors that consume input called *tokenizers*, that join symbols in the input text into *tokens*. A token is an element of the input that is known to have special meaning, such as keywords in a programming language. Tokens can be annotated with the input text, to give further meaning to the parser.

For example, a simple programming language may turn the input "*if a then b*" into a sequence of four tokens: IF, VAR(a), THEN, and VAR(b). The IF and THEN tokens denote keywords. The VAR token is parameterized with the name of the variable that it corresponds to.

Tokens are typically used to simplify the life of a human writing a parser. Tools such as `flex`[24] are used to tokenize input before the tokens are passed on to the parser, which may be specified using a language like `bison`[23]. If the parser P for some language L uses a tokenizer, then P consumes a slightly different alphabet from L.

`qcap` is not a human parser writer. The `qcap` parser generator can generate an automata from a grammar easily. The automata does not require input text to be preparsed and simplified. In fact, such a "simplification" would make the creation of a parser much more difficult, as the NPG specification for a protocol would have to be preparsed to generate tokens.

4.5.2 Choosing Callbacks for Data Return

Developers that use the `qcap` API will immediately notice that data is only made available to the application via callbacks. This approach was attractive for a number of reasons:

- It allows `qcap` to perform its own cleanup after the application has processed an event.
- It allows `qcap` to keep state on the stack between events. Pointers to stack data can even be handed to the application. This minimises the work necessary to copy data into heap data structures (saving time and programming effort), and allows for some simplifications in memory management.
- `qcap` produces four different kinds of events. It is unclear how different event types could be returned through a single function, without overloading a return type.

- The connection between subscriptions and the resulting events is very explicit.

Of course, there is a cost associated with using this style of subscription interface. It requires the application to be structured in a nonintuitive manner that may force the use of threading for interactive applications. However, given that a request for new data from `qcap` may block for a long period of time, interactive applications would have to be multithreaded anyway.

An alternative approach would see the application repeatedly requesting the next event, until the event stream is exhausted. There are three drawbacks to this approach: the application must deallocate the returned structures, four different types of data must be returned through the same interface, and it forces `qcap` to use the heap to store state between calls (instead of using the stack). In the author's mind, the latter two costs outweigh the benefit of allowing the application to request events.

4.5.3 Choosing a Network Stack

The basis of the `qcap` IP and TCP network stack is a modified version of `libnids`[59]. We chose to use `libnids` because large portions of its code base are lifted from the Linux 2.0.36 kernel[2], leading us to believe that `libnids` should be relatively fast and correct. During integration, however, we discovered that the `libnids` architecture does not make small change easy.

Two alternatives also presented themselves: implementing the network stack by hand, or using another network stack.

Writing the network stack by hand was not an attractive proposition, as the Internet protocol stack is complex. We did not wish to spend development time working on a problem that had already been solved.

[2]See the file CREDITS in the `libnids` distribution.

We could have used another protocol stack instead of `libnids`. In retrospect, this may have been a better approach, as modifying `libnids` proved to be more time consuming than originally intended. The other stack that the author considered was the x-kernel[49], which is a network stack simulator from the University of Arizona. Before implementation, the author feared that the xkernel may not be fast enough to handle large volumes of data. After implementation, the author was willing to review his hypothesis.

If we were to reimplement the network stack, we would not use `libnids`. Instead, we would likely use the x-kernel, or implement the stack ourselves.

4.5.4 Choosing a Parser Architecture

There are at least three classes of parser that we may employ. The classic parsers, the ones that get the most play in books on parsing[4, 19, 26], are LL and LR parsers. Both classes of parser share the same foundation: they are based on deterministic finite automata, they read every character in an input stream exactly once, and they usually have some amount of look ahead. `qcap` uses an entirely different class of parser: a generalized parser.

We chose not to use an LL or LR parser because of the way they handle *ambiguity*. An ambiguity in a grammar occurs when the parser receives an input, and there are two or more valid interpretations of the grammar. LL and LR parsers deal with ambiguity by using *lookahead*. With lookahead, parsers may delay a decision for a fixed amount of input.

```
in-a-house = "kitten" / "kitchen"
```

Figure 4.10: A trivial example of an ambiguous grammar in NPG.

A trivial example of an ambiguous grammar is shown in Figure 4.10. The first three characters of the rule <in-a-house> could belong to either half of the alternation. A parser is unable to ascertain which path of the alternation to take until the fourth character (either a t or a c) resolves the ambiguity. A standard LL or LR parser would require a lookahead of three characters to determine which path should be followed. Lookahead is decided when the parser is being built, and is compiled statically into the executable.

However, qcap deals with grammars that change at runtime. Figure 4.11 shows a grammar whose lookahead can only be ascertained at run time.

```
message = blob-template "!" (blob / 1*%d65-90 "!")
set {
     blob#text = blob-template;
}

blob-template = 1*%d65-90;
blob =
```

Figure 4.11: An example of an infinitely ambiguous grammar using NPG.

Unlike LL and LR parsers, *generalized parsers* run over automata that are much less optimised. A generalized parser has an automata and a set of subparsers. Outside of ambiguous sections, there is one subparser per parser. When a generalized parser hits an ambiguous section of the grammar, it spawns a subparser for each possible path. Each of the subparsers receive every character of input and progress down their own path. Whenever a subparser rejects an input, it is discarded. If the parser only has one subparser, and it rejects, then the input is rejected. If any of the subparsers accept, then the input is accepted.

We selected a generalized parsing algorithm because of its flexibility. It can be slower than other types of parsers and it performs more comparisons than are neces-

sary with other techniques, but it handles dynamic elements properly.

Chapter 5

Evaluation

To show that the NPG and `qcap` operate as expected, we demonstrate three attributes of NPG and `qcap`: (1) that the grammar and implementation create "correct" parsers, (2) that the implementation is at least as expressive and efficient as existing tools, and (3) that the parser and grammar are easy to use. This chapter provides results of our tests of NPG and `qcap`. The language is, in the opinion of the author, very easy to use. The library proved to create correct parsers. However, even though `qcap` is able to parse inputs properly, it is unusably slow.

The remainder of this chapter summerizes our experiences with NPG and `qcap`, and compares a definition of an HTTP parser in NPG to three definitions of an HTTP parser in general purpose code.

5.1 Correctness of Parser and Implementation

Our parsing approach is able to correctly parse protocols. We ran a parser generated from the NPG specification of HTTP requests over 3,598 requests that we gathered from the network connection at our lab. The parser correctly accepted valid requests,

while rejecting invalid requests.

The HTTP requests were from five hour-long samples taken daily at 14:00, starting on June 12, 2006. The samples were captured on the outside interface of the CCSL gateway. Many of the requests were from the outside, aimed at a webserver from the Math Department whose traffic we are party to; while others were from computers within our lab, sent to the outside world.

Since the volume of requests was so high, we were unable to validate that all requests that were accepted were proper HTTP. Instead, we took a random sampling of 50 requests, and inspected them ourselves. The author was satisfied that each file sampled was proper HTTP. Our assertion that the parser is correct is further supported by the requests that parser rejected, which shall be discussed below.

The parser rejected 53 streams, informing the author of the character that caused the rejection. In each case, the rejection was for a valid reason. The most common cause for rejection was an illegal character inserted into the <`Request-URI`>. The second most common cause of rejection was an illegal white space character included at the end of <`Request-Line`> by a nonconforming user agent.[1]

Interestingly, the parser also rejected a number of streams that were clearly not HTTP requests. During the initial test, the author accidentally ran the parser against HTTP responses, as well has HTTP requests; only to discover that the parser correctly rejected the HTTP responses.

Due to an artifact of the traffic capture process, some of the streams in our test were truncated, leaving binary data from the body of an HTTP response in the test. The HTTP parser correctly rejected the binary data.

[1]The agent string provided was `NSORION`. A web search fails to reveal any information about the browser.

5.2 Usability

Even though the `qcap` library is incomplete, we are able to evaluate and discuss the usability of the NPG and the `qcap` API.

We argue that the Network Protocol Grammar is a promising method for parsing protocols for three reasons:

- Declarative definitions of protocols are easy to write, understand, and debug. In our experience the NPG definition was easier to manipulate than imperative definitions.
- The hierarchical nature of the NPG allows a programmar to ignore portions of the grammar that are outside his or her scope of interest. The relationship between rules is immediately apparent, given that NPG only supports rule reference operators and the `set` clause.
- The code generated from the definition has the potential to be as efficient, and much less error prone than imperative definitions.
- A declarative definition of a protocol can be transformed to fulfil many tasks. Imperative definitions are more specialised, they are much more difficult to transform.

Even though the NPG is very usable, the implementation of the parser in `qcap` proved to be somewhat disappointing, due to speed problems. We discuss those problems in detail in the next section.

5.3 Speed of Implementation

Although the parsing algorithm produces correct results, it runs at rates far slower than desired. Ideally, the `qcap` parser would be able to process tens of megabytes per second on commodity hardware. That is not the case. The `qcap` parser currently processes data slightly faster than 5.3 kilobytes per second.

In order to evaluate the speed of the `qcap` parser, we ran the HTTP parser against a 249,060 byte series of HTTP requests, and tested the amount of time spent on the processor with the `time` command. Averaged from 10 runs, `qcap` took 46.594 seconds to load the HTTP definition, parse 249K, and exit; leading us to believe that `qcap` can parse HTTP requests at a rate of roughly 5345 bytes per second.

Through profiling and algorithmic analysis, we have found multiple causes for slow processing. They include delayed computation, our nonstandard implementation of a generalised parser, and interpretation. In the rest of this section, we detail the issues we discovered, as well as addressing possible solutions.

5.3.1 Slowdown Due to Generalised Parsing

Finite automata may be transformed from one form to another without changing important properties. The process of generating an automata that can be used with an LL or LALR parser from a nondeterministic finite automata is an example of such a transformation. The advantage of working with a generalised parser is that it can use an unmodified nondeterministic finite automata without modification. The drawback of using a generalised parser is that the underlying NFA is unoptimised, meaning that unnecessary computation may be done during a parse.

An example of this is the creation of new parsers on static ambiguous sections. Static ambiguous sections are areas of ambiguity that are expressed in the non-

dynamic portions of a grammar. Such ambiguous sections may be detected before input is received, and the NFA may be transformed to delay a parsing decision until after the ambiguous section is passed. However, by using a generalised parser we delay the detection of ambiguity until input is being processed, thereby slowing down the parse.

If we consider the slowdown due to generalised parsing on a per-token basis, the slowdown can be expressed as $O(p * t * c)$, where p is the number of active parsers, t is the number of transitions leaving the current state, and c is the cost of traversing a transition. Since there is a high degree of variability in the values of p, t, and c, we only consider their worst case values: the largest number of parsers that can be active due to a static ambiguity; the largest number of transitions leaving any state; and the largest number of actions and vetoes present on any transition.

We can improve the speed of the parse by detecting static ambiguities before the parse begins and deferring the emission of parse information until the input resolves the ambiguity. In doing so, we are converting the nondeterministic automata into a deterministic automata. The result would remove the p multiplier from the cost of executing each character: $O(t * c)$. The cost of the parse time speed improvement is an extra compilation step after the grammar has been loaded, but because we can serialise the grammar after it has been compiled, and store it in its compiled state, we do not consider this to be an overt burden. Aho et al provide the *subset construction* algorithm for detecting ambiguities in [4], that is sufficient to perform these tasks.

Note that such a change means the parser needs to be able to be able to delay actions until enough input has been received to resolve the ambiguity.

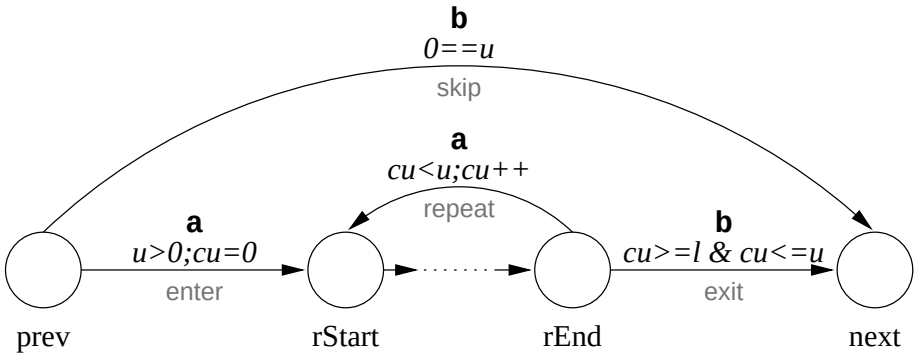


Figure 5.1: An automata modified to handle a minimum and maximum dynamic cardinality
The structure of an automata that handles both dynamic minimum and maximum cardinality.

5.3.2 Nonstandard Implementation of Generalised Parsing

The current parser generator builds nondeterministic finite automata in a manner that is more complex than necessary. Whenever there is a symbol that repeats, the last repetition is unrolled and annotated with an end-symbol action. This means that every pass through the repetition causes a new parser to be generated unnecessarily.

Logically, the parser acts in a manner consistent with the automata shown in Figure 5.1, but the internal representation is shown in Figure 5.2. The extra transition, named unrolled is necessary because it carries an end-symbol action. This approach was taken because actions immediately modify the state of the parser they run upon. We did not wish to allow actions to be delayed, because that would increase the complexity of the parser implementation.

This representation causes an extra parser to be generated every time the repetition occurs, as one parser heads back on repeat while the other follows unrolled. The next token causes one of the two parsers to be discarded. Generating and deleting a parser are unnecessary operations that occur every time a symbol repeats; adding a constant cost factor to every pass through a repeating portion of the finite automata.

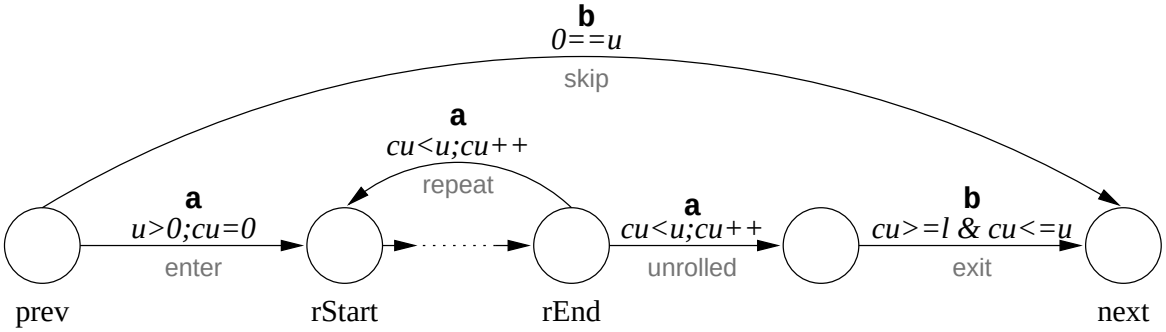


Figure 5.2: The internal representation of repeating symbols
Internally, `qcap` unrolls the last repetition of a region with a cardinality greater than one, resulting in the transition, unrolled being added to the finite automata.

We can improve efficiency be removing the unrolled transition, and moving the end-symbol action onto the exit transition.

5.3.3 Interpretation

The existing parser generator builds a nondeterministic finite automata annotated with vetoes and actions. At parse time, the parser traverses the finite automata. The author feels that such an approach is inherently slower than generating executable code that can be compiled and then run. Executable code would not necessarily store the finite automata explicitly, but would likely store the automata implicitly in its structure.

Running generated code should be faster than interpreting a finite automata, as the interpreter must loop over data contained in the automata (requiring both looping instructions and tests to see if the data in question has been exhausted). The author assumes that replacing the interpreter with a compiler would result in a speed improvement. The exact speedup is unclear, however.

In the next Section we compare the NPG definition of a protocol to multiple imperative definitions of the same protocols. Our comparison takes the form of a

case study.

5.4 Case Study

In order to provide perspective in the comparison of the NPG to other methods for writing protocol parsers and analysers, we compare the definition of an NPG parser to the definition of parsers in general purpose code. We target the client side[2] of HTTP[17] as our protocol of choice, because there many parsers are available for comparison.

We compare `thttpd`, Wireshark, and Apache to a parser defined in NPG. All three of the non-NPG parsers define HTTP implicitly in general purpose code, making our comparison artificial, as there are few commonalities between the declarative NPG definition and the parsers in question. We focus our comparison on the nature of the parsers, both qualitatively (in terms of their general structure, and coding idioms used), and quantitatively (in terms of lines). For each example, we will direct our attention to one representative part of the parser, to give the reader a feel for the code in question.

5.4.1 Review of HTTP

The server side of HTTP conversations consist of a series of requests. Each request has three parts: a request line, headers, and a body; as shown in Figure 5.3. The request line has a <`method`>, specifying the type of request; a <`url`> as a parameter; and a version, which implies the set of valid headers. Each <`header`> provides some extra information about the request, and the optional body provides parameters that may be required for the server to respond properly.

[2]By client side, we mean the portion of the stream that is sent by the client and received by the server.

```
Request         = request-line *header CRLF [body]

request-line    = method SP url SP version CRLF
method          = "POST"c / "GET"c / "HEAD"c
version         = "HTTP/"c 1*DIGIT "." 1*DIGIT

header          = ALPHA *(ALPHANUM / "-") ":"
                      *(ALPHANUM / WS / PUNCTUATION) CRLF

body            = *%x0-ff
```

Figure 5.3: A fragment of NPG describing an HTTP request. Note that the production <WS> is defined as "any white space", while <PUNCTUATION> is defined as "any punctuation character." <url> is an RFC 2396 specified Uniform Resource Locator. All other undefined productions are part of the core NPG grammar.

It should be noted that Figure 5.3 is a superset of HTTP, as defined in [17]. It will parse most requests, but it will not break headers down into the components that are necessary to perform more advanced parsing. However, it could be used to create a webserver capable of handling most requests. With a few small additions to the <header> rule, it could handle many, if not all, requests.

5.4.2 Implementations of HTTP Parsers

This section compares an HTTP parser defined in the network protocol grammar to three other parsers. In each case, we will compare the segment of code used to parse the HTTP <method> (shown in Figure 5.3). In each case, the <method> parser provides a reasonable impression of the style of parse used by each parser.

5.4.3 An HTTP Parser in NPG

The HTTP specification uses a nonstandard version of ABNF, meaning that creating the ABNF version of the protocol was more complex than expected. The changes are

fairly simple: HTTP changes the ABNF alternation operator from "/" to "|", allowing linear white space to occur between any elements in a sequence, and the addition of a list operator that implicitly allows commas in sequences. Most annoyingly, the RFC does not gather all of the ABNF together in one place, meaning that a human must dig through the document, copying tiny snippets of specification into one file.

Porting the changes from the HTTP RFC to canonical ABNF was trivial drudge work, taking roughly four hours. Adding the NPG construct (to handle the `Content-Type` header) took a few minutes. The most complex portion of the task was testing, which took an additional five hours. The resulting definition was 501 lines.

To get an idea of the nature of the changes necessary to port the HTTP specification to NPG, we consider Section 5.1 of the RFC, which defines the first line of an HTTP request. See Figure 5.4 for the text from the RFC.

In order to port this snippet of ABNF to NPG, we fix the syntactic errors, modify the grammar to meet the constraints supplied in the text, and then write the result to a file. The errors are trivial to fix, we simply substitute the ABNF alternation operator (/) for the nonstandard pipe (|) that was used in the text.

The prose places an additional constraint on the grammar: the <`Method`> text is to be case sensitive. In ABNF, strings are not case sensitive, meaning that the ABNF definition given contradicts the prose. We assume that the prose is canonical, and modify the definition, using the NPG case sensitivity operator described in Section 3.3.2. The resulting NPG file is very similar to the original ABNF shown in the RFC, as can be seen in Figure 5.5.

Even though we define the grammar, our task is not complete. We must also furnish a program with the ability to instantiate and run the NPG parser. An example of the code to start an NPG-generated HTTP parser is shown in Figure 5.6. We consider this code to be part of the background infrastructure for parsing, and will

```
5.1 Request-Line

   The Request-Line begins with a method token, followed by the
   Request-URI and the protocol version, and ending with CRLF. The
   elements are separated by SP characters.  No CR or LF is allowed
   except in the final CRLF sequence.


         Request-Line = Method SP Request-URI SP HTTP-Version CRLF

5.1.1 Method

   The Method token indicates the method to be performed
   on the resource identified by the Request-URI.
   The method is case-sensitive.


         Method    = "OPTIONS"    ; Section 9.2
                   | "GET"        ; Section 9.3
                   | "HEAD"       ; Section 9.4
                   | "POST"       ; Section 9.5
                   | "PUT"        ; Section 9.6
                   | "DELETE"     ; Section 9.7
                   | "TRACE"      ; Section 9.8
                   | "CONNECT"    ; Section 9.9
                   | extension-method
         extension-method = token
```

Figure 5.4: Snippet of RFC 2616[17] describing the first line of an HTTP request. Underlined portions of prose indicate constraints that modify the ABNF definition. Underlined portions of ABNF indicate syntactically invalid portions of the definition.

not go into detail to explain it. We include it to show that there is no hidden complexity to the NPG parser. The programmer does not pay for protocol simplicity by having to navigate a complex API.

```
; S5.1 - Request-Line
          Request-Line = Method SP Request-URI SP HTTP-Version CRLF

; S5.1.1 - Method
          Method    = "OPTIONS"c
                    / "GET"c
                    / "HEAD"c
                    / "POST"c
                    / "PUT"c
                    / "DELETE"c
                    / "TRACE"c
                    / "CONNECT"c
```

Figure 5.5: Snippet of the NPG grammar for HTTP, as mapped from Figure 5.4. Underlined portions denote text changed from the ABNF definition in the RFC.

```
qcap_t * qp;

qcap_npg_handler_add( qp, "http", "Method", method_callback, NULL);


/* The callback to be triggered whenever an HTTP <Method> is received.  */
qcap_handler_control_t
method_callback(
          qcap_t * qp, qcap_stream_half_t * stream,
          char * protocolName, char * ruleName,
          offset_t startOfMatch, offset_t endOfMatch, qcap_octet_t * matchText,
          void * userData
) {
     Handle the text requested by the callback.
}
```

Figure 5.6: Code to load and run an NPG specification. The function `qcap_npg_handler_add()` adds a callback to the given `qcap` session. The NPG file contains meta-information informing `qcap` which side of a stream the callback should be attached to, as well as the standard port numbers in use.

5.4.4 thttpd's HTTP Parser

thttpd[3] is a small webserver. According to its web page, it is a "simple, small, portable, fast, and secure HTTP server." We choose to compare thttpd's parser against NPG because it conveniently locates all of its parsing code in one place (unlike many other servers).

The thttpd parser is roughly 500 lines of C code. It uses the POSIX standard strpbrk(), strspn(), and strcmp() functions to recognise known strings. For the most part, it acts as a backtracking parser, repeatedly comparing input to expected values until it finds one that matches. When text fields of a specific format are expected, they are pulled apart, byte by byte.

The thttpd parser is longer (in lines of C) than the NPG specification for the server side of HTTP. Furthermore, thttpd parses a subset of the HTTP protocol.

A fragment of thttpd's code for parsing the HTTP <Method> is reproduced in Figure 5.7. The general approach of breaking the input line at each space can be seen for the first component, the method. Below the component decomposition, there is a fragment of code for determining which method was received. Clearly, the volume of code text is greater than that used in the NPG definition. From this example we see how thttpd backtracks until it finds a match: the parser compares method_str to each legal value, until it finds a match or runs out of values.

Aside from the slightly confusing use of strpbrk(), and the interleaving of parsing code with handling code, thttpd is easy enough to understand. It is not overly complex, although it is much more verbose than the NPG definition.

According to the thttpd release notes[4], there were only a handful of changes to the request parsing code, either in terms of bug fixes or new features added. However,

[3] http://www.acme.com/software/thttpd

[4] http://www.acme.com/software/thttpd/\#releasenotes

```
/* Get the end of the method */
char * method_str = bufgets( hc );
char * url = strpbrk( method_str, " \012\015" );
if ( url == (char*) 0 ) {
     return -1;
}
*url++ = '\0';
/* Repeat for URI and HTTP version */
               ...  Parse the rest of the request line ...

/* Determine which method the client has sent */
if ( strcasecmp( method_str, httpd_method_str( METHOD_GET ) ) == 0 )
     hc->method = METHOD_GET;
else if ( strcasecmp( method_str, httpd_method_str( METHOD_HEAD ) ) == 0 )
     hc->method = METHOD_HEAD;
else if ( strcasecmp( method_str, httpd_method_str( METHOD_POST ) ) == 0 )
     hc->method = METHOD_POST;
The following methods were added to give thttpd the same capabilities
as the NPG grammar
else if ( strcasecmp( method_str, httpd_method_str( METHOD_OPTIONS ) ) == 0 )
     hc->method = METHOD_OPTIONS;
else if ( strcasecmp( method_str, httpd_method_str( METHOD_PUT ) ) == 0 )
     hc->method = METHOD_PUT;
else if ( strcasecmp( method_str, httpd_method_str( METHOD_DELETE ) ) == 0 )
     hc->method = METHOD_DELETE;
else if ( strcasecmp( method_str, httpd_method_str( METHOD_TRACE ) ) == 0 )
     hc->method = METHOD_TRACE;
else if ( strcasecmp( method_str, httpd_method_str( METHOD_CONNECT ) ) == 0 )
     hc->method = METHOD_CONNECT;
else {
     return -1;
}
```

Figure 5.7: Code from `thttpd`(version 2.25b) for parsing the method of an HTTP request. Error handling code has been removed. The function `httpd_method_str()` does a lookup for the string representation of the requested method. Only the first three methods are present in the original code. The remainder were added to provide parity with the NPG specification.

a number of buffer overflows have been detected in `thttpd`'s parsing code.[5]

[5] `http://www.securityfocus.com/archive/1/342584`

5.4.5 Wireshark's HTTP Parser

Wireshark[6] is a datagram oriented protocol analyser. It has an HTTP parser, capable of reading HTTP request lines and headers from individual datagrams and sequences of datagrams. The HTTP parser is tightly coupled with the code for reporting the parse results.

Wireshark's HTTP parser is roughly 1027 lines long (including extensive comments and some reporting of parse results). It handles grammars outside of the server side HTTP specification, such as protocols that masquerade as HTTP, and client side responses. It is the author's estimation that the extra code for dealing with those other grammars numbers no more than 200 to 250 lines. Similar to `thttpd`, the parser backtracks, testing a set of known possibilities on the input until one matches.

The portion of the Wireshark HTTP parser responsible for parsing the HTTP method is shown in Figure 5.8. Wireshark uses a similar approach to `thttpd`, but instead of trying every legal HTTP method sequentially, it gets the length of the method, and only compares the received methods to expected methods of the same length. The approach should be faster than `thttpd`'s linear search, as it involves fewer string comparisons. Although it would be slower than the optimal speed provided by a non-backtracking parser (as could be generated from the NPG definition).

The author found Wireshark's code harder to understand than `thttpd`, due to the blurring of the line between the parser, and the reporting structure. It is unclear which calculations are necessary to parse incoming datagrams, and which calculations are necessary for providing output.

Wireshark's HTTP parser was noted as containing minor vulnerabilities[7]; parsers

`http://seclists.org/lists/vuln-dev/1999/Nov/0017.html`

[6]The HTTP parser described here is contained in the Ethereal Subversion repository. The version of the parser is 17769, and can be found at `http://anonsvn.ethereal.com/viewcvs/viewcvs.py/trunk/epan/dissectors/packet-http.c?rev=17769\&view=auto`.

[7]`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2361`

```
int linelen = tvb_find_line_end(tvb, offset,
     tvb_ensure_length_remaining(tvb, offset), &next_offset, FALSE);

/* OK, does it look like an HTTP request or response?  */
is_request_or_reply =
     is_http_request_or_reply((const gchar *)line,
     &http_type, &reqresp_dissector);
if (is_request_or_reply)
     goto is_http;
          ...
static int
is_http_request_or_reply(const gchar *data, http_type_t *type,
          ReqRespDissector *reqresp_dissector)
{
     /* Look for the space following the Method.  */
     The original code contained a loop searching for a space
     const guchar * ptr = strpbrk(data, " ");
     if (ptr == NULL)
          return 0;
     int len = ptr - data;

     switch (len) {
          case 3:
               if ( strncmp(data, "GET", len) == 0 ||
                    strncmp(data, "PUT", len) == 0) {
                    *type = HTTP_REQUEST;
               }
               break;

          case 4:
               if ( strncmp(data, "HEAD", len) == 0 ||
                    strncmp(data, "POST", len) == 0) {
                    *type = HTTP_REQUEST;
               }
               break;
```

Figure 5.8: Part of the Wireshark HTTP parser. Error handling code has been removed, as has code unrelated to parsing. There are more methods supported in the original code, but non-HTTP related methods were removed.

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1141

for other protocols have contained more serious vulnerabilities[15].

5.4.6 Apache's HTTP Parser

Apache is a popular open source webserver. According to NetCraft, it was used to host 61.25% of all websites on the Internet in July, 2006.[8] It is highly optimised and designed for easy extensibility through plugins. As such, its HTTP parsing code is a mix of deferral and optimised reading. Whereas Wireshark mixes a great deal of reporting code with its parser, Apache mixes a large number of memory allocation decisions and defensive coding practices with its parser.

The HTTP request parser operates in two passes. The first pass reads the request line and headers into in-memory structures. At the end of the first pass, the method, URL, and protocol are known, and the headers are stored in a general purpose header table. In the second pass, the headers are parsed as necessary. Because the second pass is incremental and spread throughout the codebase, we cannot make an accurate estimate of its size. The first parse, however, takes roughly 521 lines.

We continue our comparison of parsers by examining the <`Method`> parsing portion of the Apache parser. The Apache parser is not a backtracking parser, instead, the developers have clearly attempted to limit the number of comparisons made on each character, as can be seen in Figure 5.9. The parser consists of a series of nested comparisons forming an implicit trie, which was generated by the `shilka` tool from the `cocom`[9] open source project. This is the closest approximation to a formal grammar that the author was able to uncover during this survey.

Of the parsers surveyed, Apache's makes the clearest distinction between parsing code and application code. Despite that, Apache's source is still difficult to read for

[8] `http://news.netcraft.com/archives/2006/06/28/july\_2006\_web\_server\_survey.html`

[9] `http://cocom.sourceforge.net/`

a neophyte, as input reading and parsing are split between two files in different parts of the source tree, and much of the reading uses Apache-specific data structures.

```
Acquire the request line
rv = ap_rgetline(&(r->the_request),
     (apr_size_t)(r->server->limit_req_line + 2), &len, r, 0, bb);
if (rv != APR_SUCCESS)
     return 0;

Find the start of the method
r->method = ap_getword_white(r->pool, r->the_request);
r->method_number = ap_method_number_of(r->method);

Parse the method
int len = strlen(method);
switch (len) {
     case 3:
          switch (method[0]) {
               case 'P':
                    return (method[1] == 'U' && method[2] == 'T'
                         ?  M_PUT : UNKNOWN_METHOD);
               case 'G':
                    return (method[1] == 'E' && method[2] == 'T'
                         ?  M_GET : UNKNOWN_METHOD);
               default:
                    return UNKNOWN_METHOD;
          }
     case 4:
          switch (method[0]) {
               case 'H':
                    return (method[1] == 'E' && method[2] == 'A'
                         && method[3] == 'D'
                         ?  M_GET : UNKNOWN_METHOD);
               case 'P':
                         return (method[1] == 'O' && method[2] == 'S'
                         && method[3] == 'T'
                         ?  M_POST : UNKNOWN_METHOD);
```

Figure 5.9: The Apache 2.2.2 HTTP request method parser. Error handling has been removed, and code has been shifted from various files into a single listing, and comments have been added.

Apache's HTTP parser has been the source of a number of vulnerabilities[10]. Interestingly, the vulnerabilities found by the author seem to deal almost entirely with memory exhaustion. This may be a result of Apache's two pass parser: since the first pass is very simple, only responsible for reading untrusted data into memory, it does not appear as prone to assault as a full hand-coded parser.

5.4.7 Comparison of Approaches

The three parsers listed are very similar. They accept lines of text, and then attempt to break the line into smaller chunks. The parsers differ slightly in their approach to dealing with the smaller chunks: `thttpd` and Wireshark repeatedly test input against expected values, while Apache looks for individual characters that would uniquely identify the input.

In each case, the parser was longer than the complete ABNF specification for HTTP, while only parsing a subset of the entire protocol. The only exception to this rule is NPG, which is the same size as the original ABNF specification. NPG's succinct, declarative style provides a number of advantages over general purpose code: it is easier to maintain, generated parsers can be efficient and secure, and specifications allow for reuse.

In order to properly prove that NPG really has the following attributes, we would need to undertake a study comparing a number of parsers written in different languages, to one written in NPG. Ideally, such a study would involve a number of different teams of programmers, building software that would be used in some realistic environment. However, such a test is beyond the scope of this thesis. Instead

[10] URL:http://lists.grok.org.uk/pipermail/full-disclosure/2004-November/028248.html
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0786
http://lists.grok.org.uk/pipermail/full-disclosure/2004-June/023133.html

of basing our argument on empirical testing, we must base it on reasoning about the nature of NPG's declarative style.

NPG Is Easier to Maintain

NPG is easier to maintain than general purpose code because it forces the programmer to explicitly state the parsing rules for a protocol. The NPG keeps rules separate from the code that uses the result of the parse, as well as the code that performs the parse. In contrast, general purpose code forces programmers to mix parsing rules with bookkeeping code necessary for a proper parse such as updating a state machine, buffer maintenance, incrementing counters, or managing repeated passes over the input.

The `qcap` implementation of NPG not only makes parsing code easier to maintain, but it provides an explicit interface between the protocol parser and the application. By forcing the programmer to use callbacks to extract information from the parser, the boundary between the two domains is well defined. NPG enforces strong encapsulation, splitting the parser from the rest of the code base.

The size of NPG specifications makes the grammar superior to general purpose code. Because the NPG specification is small compared to hand-written parsers, it should be easier for programmers to explore, understand, and modify.

Generated parsers can be efficient and secure

Parsers built from an NPG definition can be both secure and efficient, assuming that the parser generator is well designed. The NPG parser generator provides developers with these attributes at minimal cost.

Generated parsers should be at least as secure as hand written parsers because the author of the parser generator is able to design the parser generator to avoid known

vulnerabilities. As vulnerabilities in generated parsers are discovered, the author is able to improve the generator to prevent those as well. All parsers generated from that point on, should avoid that vulnerability. In contrast, hand written parsers must be tested for vulnerabilities individually and must be improved individually.

Similarly, generated parsers can be efficient. The author of the parser generator is able to optimise the generated code for speed as well as security. As other developers use the parser generator and submit patches or suggestions for optimisations, the parser implementation can be improved.

In both cases, using a parser generator allows a developer to use the work of others to build a parser that is secure and efficient. Compared to the process of building a parser from scratch, profiling it, running security audits, and making improvements, there is much less work involved.

An NPG specification is fertile ground for reuse

The NPG's declarative style is easy for programs to transform into other representations, making the definition fertile ground for further development. For example, we could use an NPG definition to create parsers in many different languages, or we could use the definition to automatically build test suites for servers. For other possible uses, see Section 6.2.

It is difficult to transform parsers specified in general purpose code, as the parser must be disentangled from the rest of the code base, and the flow of control must be followed. The best hope for reusing the code in a general purpose parser is to tear it out and wrap it in an API so that other programmers may use it as a parser for their projects.

5.5 Experience Creating Grammars

Over the course of this project, the author has created NPG specifications for a number of existing protocols, including HTTP and SMTP. The experience of building the grammars leads the author to a number of conclusions.

5.5.1 RFC-defined Grammars Are Seldom Proper ABNF

Each of the ABNF-specified protocols examined contained syntax errors. Usually, the problems were small errors, such as using illegal characters in rule names, or typos in rule references (often dealing with the case of the first letter in the rule name).

Occasionally, as is the case with HTTP, the authors add new syntactic constructs to ABNF and modify existing ABNF constructs. Their changes are easy to port to correct ABNF, but the process is time consuming.

The headaches involved in this process should not be overestimated, however. In the case of both HTTP and SMTP, the time taken to create a specification from an existing ABNF-like grammar was only a few hours. Testing took only a handful of hours. Compared to building a parser from scratch, the time taken is negligible.

5.5.2 RFC-defined Grammars May Be Incomplete

Just because a protocol is defined with a context free grammar, we cannot assume that the protocol is fully defined in grammar. The original protocol authors may have omitted "fiddly" portions of the formal definition, opting to define those in text instead. One example of this behaviour is found in the SMTP grammar. The SMTP does not provide a definition of the text following the `DATA` verb[52].

Neither of the specifications that we ported provided a top level rule in the formal grammar. In other words, the RFCs did not include an overall formal definition

of a protocol session; instead, they specified individual interactions. As a result, the author had to determine the proper ordering of interactions and string them together.

Such omissions complicate the process of grammar porting, but are not especially arduous.

5.5.3 ABNF Is Incomplete

The ABNF specification omits at least two semantics that are all but necessary for parser generation.

First, ABNF does not define a method for indicating which rule is the root of the grammar. If a grammar contains two or more rules, a parser generator is left guessing as to which rule should be the topmost rule in the parser generator.

Second, ABNF does not specify an end of stream token, meaning that there is no way to indicate when a stream is supposed to end. For some protocols (such as SMTP), where there are strings that indicate an end of stream in the protocol itself, this did not pose a serious problem. For other protocols such as HTTP[11], the end of the stream is a syntactically significant event. For that reason, we extended NPG to include a stream termination terminal.

5.5.4 NPG Requires Better Parsing Direction

Both PacketTypes and GAPAL allow protocol specifications to contain assignments that direct parsing. In each case, the language allows protocol authors to insert a structure that changes the rule associated with a rule name. In other words, when some rule <`current`> matches, it is able to change the rule associated with the rule name <`future`>. We call such direction *dynamically directed parsing*.

[11]Specifically HTTP sessions that do not include a length field for the response body.

For example consider a protocol defined by G_{Set}, shown in Figure 5.10. It uses a rule named <control> to set the type of payload expected in <data>. If the <control> is binary, we wish to allow any bytes in <data>; but if <control> is text, we wish to allow only ASCII upper-case characters.

```
a = 1*control data

control = ("Type:" ("text" / "binary") / "Length:" 1*%d48-57) %d10

data = *%d0-255
```

Figure 5.10: Grammar G_{Set}. Although not shown in this grammar, we wish to use the Type branch of the <control> alternation to set the type of characters that are valid in <data>.

The standard manner of defining such a parse is seen in the grammar G_{SetCFG}, shown in Figure 5.11. As we can see, the <control> rule of G_{Set} must be reproduced in its entirety, in order to properly parse the input. So long as the list of Types we wish to support is small, then the repetition is not onerous. If we wish to expand our types to include binary, hex, octal, Unicode, and ebcdic; the grammar will become annoyingly verbose.

```
a = type1 / type2

type1 = 1*(("Type:  binary" / "Length:" 1*%d48-57) %d10) *%d0-255

type1 = 1*(("Type:  text" / "Length:" 1*%d48-57) %d10) *%d65-90
```

Figure 5.11: An explicit enumeration of types to emulate dynamically directed parsing.

Like dynamic cardinality and dynamic text, this is a problem of enumeration and verbosity. It is possible to use a context free grammar to represent this structure,

but it becomes onerous when there are too many possibilities to enumerate, or when there are too many productions that must be repeated for each enumeration. We call this problem *dynamically directed parsing*.

We can extend NPG to handle dynamically directed parsing with a trivial extension to the `set` clause. So far, we have supported two types of assignment: integer assignment for setting cardinality, and string assignment for modifying terminals. We add another type of assignment, rule assignment, to modify the symbols associated with a rule name.

To handle rule assignment, we allow a rule name to appear on the LHS of an expression. The RHS is allowed to contain the name of any other rule. When a rule <a> has <b> assigned to it, <a> will accept an input if and only if <b> will accept it. This allows us to rewrite G_{Set} as G_{SetNPG}, shown in Figure 5.12.

```
a = 1*control data

control = ("Type:" (cText / cBinary) / "Length:" 1*%d48-57) %d10

cText =   {"text"}       set  { data = dataText; }
cBinary = {"binary"}     set  { data = dataBinary; }

data =
dataText = d65-90
dataBinary = d0-255
```

Figure 5.12: We can avoid explicitly enumerating values in dynamically directed parsing by using rule assignment.

5.6 Summary of Evaluation

A complete evaluation of NPG and `qcap` requires empirical comparisons of `qcap`'s generated parsers to hand-written parsers; or, failing that, an evaluation of `qcap`'s

correctness. Although the author cannot furnish the first, he does provide the second; showing that `qcap` is able to parse correctly, if slowly.

We show that NPG's declarative nature makes it easy to use. A specification defined in NPG is shorter than an equivalent definition in general purpose code. Parser generators built for NPG can be optimised for speed and security, potentially lessening the amount of effort that must be expended by the programmer. Specifying a protocol with NPG creates reusable final result, as the definition is not tied to a programming language or purpose; it can be transformed and used in ways not intended by the protocol designer.

Chapter 6

Discussion

We have created a novel language, NPG, for specifying network protocols, that handles common idioms in network programming. Our grammar provides for the concise expression of dynamic regions, which cannot practically be expressed using context free grammars. The language is a mixture of ABNF, one of the standard languages for specifying streaming network protocols, and the Packet Types packet filtering language.

We have shown that the NPG can be used to specify a grammar that can be used to create a parser for network data. We have done this by building a parser generator into the `qcap` network analysis library.

The `qcap` library provides other novel functionality. It exposes all levels of the network stack to applications, cross referencing physical layer events to the transport and application layers. `qcap` is the only library that provides programmatic access to the higher layers of the network stack.

In doing so, we have achieved our initial goal, which was to create a grammar for parsing stream oriented network protocols.

6.1 Implications of Results

The `qcap` library is currently very slow, but the speed results for GAPA, provided by Borisov et al[8], indicate that the use of grammars to specify protocols for monitors is feasible.

It is worth noting that the GAPA framework interprets the grammar during the parse, just as `qcap` does. It seems likely that the GAPA parser could be sped up further if the grammar definition was compiled directly into machine code. Such speed gains could make the parser applicable to other applications.

6.2 Other Applications of the NPG

The ability to write a grammar that can completely define a network protocol provides many opportunities, some of which are more apparent than others. We will discuss these opportunities in order of effort: the easiest to achieve will be described first.

6.2.1 Validation of Existing Protocol Implementations

At the most basic level, NPG allows us to recognise a stream of network data, determining if a server should be able to parse it. Such recognition only needs to be done during debugging, but may be useful in scenarios where both a client and server are being developed in parallel.

Stream validation may also be useful in production environments. Consider a scenario where outgoing SMTP connections are allowed out of a network, but HTTP connections are not. A savvy user may set up an HTTP proxy operating on the standard SMTP port outside the network, and use that to perform any activity they desire. To detect such a scenario, we need to be able to examine a stream and

recognise if it is of a specific protocol.

6.2.2 Testing Protocol Syntax

Currently, protocols are defined in prose spiced up with BNF-like syntax to make the definition less ambiguous. A good example are RFCs discussed in previous chapters. Because the RFC is not machine-readable, it is not possible to directly test the definition. The definition may be syntactically incorrect (as was the case with many of the protocols converted to NPG), or it may not express what the authors intended.

Without machine readable protocol specifications, the closest thing to a true test is the reference implementation of a protocol. The reference implementation provides a "canonical" implementation that developers may use to test other implementations of the protocol. However, the reference implementation will likely contain bugs, meaning that it may fail to properly parse the protocol as specified.

With the NPG, it is possible to create a specification that can be converted directly into a parser. With that parser, it is possible to test the protocol syntax directly.

6.2.3 Trivial Protocol Analysers

In most situations, the software at each end of a network stream has to be tightly coupled with the protocol it is running. However, protocol analysers do not need such tight coupling, as they simply need to analyse and classify passing traffic (possibly recording some subset of the traffic for analysis). A network protocol grammar allows protocol analysers to be easily extended after deployment, by adding a new grammar definition for any new protocols that may appear.

6.2.4 Canonicalisation of Implementation

When creating a new implementation of a network server or client, developers must convert the human-readable specification into a machine-readable executable. Tools, such as `flex`[24] or `bison`[23] exist, but require that the programmer translate a protocol definition from a technical document into executable code. That process is boring, error-prone, and labour intensive. Even if the programmer creates an implementation that works for some set of test cases, there is no guarantee that the implementation is correct relative to the protocol standard, and there is no guarantee that the test cases provide sufficient coverage to ensure that all possible conversations will be parsed correctly.

Automatically building protocol parsers is an extremely attractive proposition: it speeds up the development cycle, allowing programmers to generate optimised parsing code; gives developers the opportunity to prevent well-known parsing bugs; and ensures that the final parser is correct relative to the protocol definition. Intriguingly, it also gives us the opportunity to write the protocol definition once and generate implementations for many programming languages.

If the NPG is to be used to this purpose, trivial extensions could be added that would provide other opportunities. Protocols often require common tasks to be performed, such as password validation, compression, or encryption. The NPG could be modified to allow decoding functions to be set in the grammar.

In such a scenario, encryption could be added to a protocol by adding an `encoding` clause to a rule, and specifying the type of encoding to use, and the fields that contain credentials. When the specification is compiled into a parser, a library of callbacks would be searched for the appropriate decoding function. The decoding function would be automatically incorporated into the parser, simplifying development.

Generated parsers could be used in new applications as the entire parsing frame-

work, providing data to application code through call-backs or some other mechanism, as proposed by Glenn Wurster[60].

They may also find a role as preprocessors for legacy applications. If a legacy application has a known security vulnerability or is otherwise unable to process certain types of input, a proxy can be built between the source of the input and the application itself. The proxy would be responsible for modifying the input (and possibly the output) to minimise the impact of the fault, without modifying the application itself. In such a scenario, a generated parser could provide a secure, standards-compliant front-end for the proxy. Such an approach is similar to Microsoft's Shield[58] defence. James Deverick is currently involved in using a network-based proxy to this end[16].

6.3 Limitations

The NPG is a concise rendering for certain classes of context free grammars. As such, the language is extremely expressive. From a protocol design standpoint, however, there are still areas to improve.

6.3.1 Expressing Encodings

The text of protocols is often transformed in some way, either to confound listeners, or to limit bandwidth consumption. In both cases, the net result is the same: NPG cannot describe the structure of the transformed text.

A simple example can be found in the base 64 encoding of HTTP authentication tokens[20]. When a client sends a username and password to a server with the `Authenticate` header, and specifies the `Basic` encoding type, it passes a base 64 encoding of a user name and password to the server, separated by a colon. In its current incarnation, the NPG would be able to locate the encoded text, but would

not be able to parse the user name field and password field.

6.3.2 Synchronising Client and Server

Every TCP connection has two sides. Currently, the NPG does not provide any mechanism for sharing data between the parser reading each side of the connection. However, it is conceivable that a protocol may require information sent from one participant to modify the syntax of the protocol coming from the other participant.

For example, if a client requests x bytes of data from a server, the server may oblige and provide exactly x bytes. If the server doesn't explicitly state x in its outgoing stream, a parser reading that stream has no way of telling the size of x, unless it was informed of x by the parser reading the other direction.

So far, we have not encountered a protocol that is so thrifty with its transmissions that it does not have recipients repeat parameters back to senders. However, we would not be surprised if such a protocol existed.

6.3.3 Selecting Rules Based on Previous Matches

NPG is based on Packet Types, a datagram filtering language. Packet Types provides a syntactic construct allowing a grammar to specify that "if some rule <`initial`> matched in the past, match some future input against <`following`>." This is useful for datagram parsing because fields that occur early in the datagram often specify the format of the payload, which follows another large volume of fields.

Again, we have not directly encountered a protocol that requires this functionality; however it may prove to be useful for other protocols in future.

6.4 Further Work

Although we have a fully functional, open source implementation of an NPG parser generator, there are many possibilities for further work.

6.4.1 Roadmap for Improving `qcap`

The `qcap` library is functional but in need of improvements. The following list is a roadmap for improvements that should be made to the library.

Replace the generalised parser with an LL(k) parser. The existing parser is functional, but it is unnecessarily slow. The goal of the project remains the construction of a fast parser for network traffic. Until the parser runs at acceptable speeds, `qcap` will not be ready for widespread use.

We select an LL(k) parser[1] for two reasons. First, it is easy to implement an LL parser and the transformation function (*subset construction*, as described in [4]) is well understood. Second, an LR parser may require that the entire stream be read before returning a parse result[8].

Reimplementation is a two-step process. Initially, the *subset construction* transformation must be implemented to turn the NPG nondeterministic finite automata into a deterministic finite automata. The finite automata is usable by an LL(k) parser. Since an LL(k) deterministic finite automata is also usable with the existing generalised parser, it can be tested with the existing parser. After the transformation has been debugged, the LL interpreter can be implemented.

Modify the NPG to support dynamically directed parsing. Although dynamically directed parsing is not strictly necessary to parse protocols it is convenient,

[1] Where k is chosen upon examining the grammar.

as it allows a protocol designer to store a parse decision at token k and defer its use until token $k + n$. It is the author's belief that the NPG would benefit from the addition of dynamically directed parsing.

Connect the `qcap` NPG parser generator to the `qcap` network analysis stack. Parsers built from NPG grammars only run on input text, they do not yet run on data parsed from network streams. In order for `qcap` to be a useful network analysis library, NPG parsers must be able to read network data.

Improve the `qcap` API. The `qcap` API is functional, but is not as easy to use as it could be. Small usability fixes are desirable.

6.4.2 NPG Enabled Network Monitoring Tools

`qcap` has been designed as a library for developing network monitoring tools. Because `qcap` uses the NPG to define parsers, such tools would be very easy to extend to handle new streaming protocols. `qcap` also provides a full view of the network stack, meaning that users of such a tool would be able to see the effects of the behaviour of high level application protocols on low level transport and network layer protocols.

6.4.3 Using NPG In Applications

Even though the Network Protocol Grammar is targeted at network monitoring tools, it could be useful to application developers. Existing parser generators do not use language specifications directly to create parsers, they rely upon a programmer to port the specification to an intermediate language before creating the parser. Because NPG is a superset of the existing specification language ABNF, application developers using NPG as an input to a parser generator would not need to port the language; they could use the specification directly.

Before NPG can be used to generate application code, a new parser generator must be written that would enable the creation of a parser in static code. However, a subtle optimisation could be added: the callbacks that the application is interested in are known at compile time, instead of run time (as is the case with `qcap`). Knowing which portions of the grammar matter at compile time allows the parser generator to avoid checks for callbacks at the start and completion of regions.

6.4.4 Using NPG to Send Data

So far, our discussion of the NPG has centred on using the language to recognise and parse incoming streams of data. It could also be used to generate outgoing streams of data from applications, similar to the process followed by ASN.1[32]. Since NPG only defines an on-the-wire format, without defining the data structures used to generate the protocol text, the value of such an approach is questionable. It would require an extra language (or additions to the NPG) that would map between data contained in the application and the one-the-wire specification.

Redefining Grammars as Obfuscation

Treating a protocol grammar as a first class object provides an interesting opportunity. It allows us to automatically transform the grammar into other forms[48]. If we assume that there is an ASN.1-style encoding and decoding layer shielding the application from changes in the protocol text, then we can modify the encoding in arbitrary ways without having to modify the application at all.

Such manipulation could be used for obfuscation at run time. If the parties in a communication session can agree on a transformation in a manner that is opaque to an eavesdropping party, a simple manipulation could be used to protect the session from automated analysis by outside parties.

6.4.5 Implementing Grammars That Modify Other Grammars

Many application protocols in the Internet suite have been modified since their initial specification. For example RFC 1870[35] extends SMTP to allow the client to specify the size of an email before it is sent to the server. The extension modifies two SMTP productions minimally: one on the client side and one on the server side. The modifications are sufficient to make RFC 1870 streams break SMTP parsers. Ideally, NPG would provide a mechanism to allow minimal changes such as these to be added without modifying the original grammar.

One approach to handling such a change would see a special NPG construct to allow one grammar to be declared as a modifier to another. In other words, if we have a grammar G_{base}, some other grammar G_{ext} can declare itself an *modifier* to G_{base}. The modifier would change some subset of G_{base}'s rules. If we take the SMTP client grammar as an example, we can say that G_{SMTP} is the base grammar, and $G_{RFC1870}$ modifies it.

Ideally, modifiers would be implemented in a manner that would allow multiple modifiers to change a single grammar.

6.4.6 Bit Oriented Parsers

Parsers are machines that consume symbols and change their internal state depending on that symbol. Previously in this document, we have implicitly assumed that the symbols our parsers consume are bytes. That need not be the case.

The NPG provides bit-oriented literals, meaning that a protocol may contain fields smaller than a byte. In order for a parser to consume such fields, it would need to shrink its symbol size to handle n bits, where n is the length of the smallest bit

oriented field. Assumedly, there would be a runtime cost to mask off the appropriate number of bits from the input, and then feed them into the parser.

`qcap` has a fixed symbol size of eight bits, meaning that any protocol using binary literals must ensure that the literals contain exactly eight bits. It could be extended to smaller symbols, but the value of such an extension is questionable: the author is not aware of any stream oriented protocols that do not respect byte boundaries.

6.5 Dangers of the Network Protocol Grammar

Just as NPG offers a number of promising technical opportunities, it presents moral risks. `qcap`'s express purpose is exploring network traffic, providing access to the full text of unencrypted TCP streams. That tool is available as an open source library, free to anyone with unencumbered access to the Internet.

6.5.1 Legal Concerns Surrounding the Network Protocol Grammar

Although network traffic monitoring may be illegal in some areas, we do not address any legal concerns surrounding this technology. It is the deployer's responsibility to ensure that their tools are in compliance with local laws. `qcap` is being developed and deployed in an academic environment with the goal of improving understanding of network traffic.

6.5.2 Moral Concerns Surrounding the Network Protocol Grammar

The inherent risk of `qcap` is that it will be used to violate the privacy of people using the Internet. So long as privacy violations lead to nothing more serious than salacious gossip and embarrassment for the unwary, the author is happy to see his work used in this way.[2] However, violation of privacy can have much more serious consequences. Consider a scenario where a repressive government is monitoring Internet traffic to prevent challenges to its authority: in such a situation, a breach of privacy may result in a jail term or death[3]. If the ideas presented in this thesis are used by a repressive government as a tool of repression or persecution, the author will have committed an immoral act.

The author's response to this risk is pragmatic. `qcap` is an improvement on the existing approach to building network parsers. It simply promises to make parser development easier. Programmers can already build fast, efficient parsers. The NPG speeds up development, and produces parsers with fewer bugs. The author feels it is unlikely that existing network-sniffing tools are so shoddy that any organisation with a large installed base will discard their existing network sniffer in favour of `qcap` in the short term. In the long term, organisations may start to use the NPG as a basis for their own sniffers.

It seems much more likely that `qcap` would be adopted by small businesses in open societies. Businesses could use `qcap` to monitor traffic flowing on their LANs and WANs for misuse. The categories of misuse are broad and arbitrary, but may include file sharing and instant messaging. The immediate effect would be the limiting of protocols running on "inappropriate" networks.

[2] Of course, if the author falls prey to sniffing, his opinion on this matter is likely to change.

[3] `http://web.amnesty.org/library/index/engasa170012004`

At this point, however, the author suspects that the second order effects of `qcap` will become apparent. The developers of illicit applications will wish to continue to use their applications. We can expect those developers to make their protocols harder to detect; either by tunnelling them over existing protocols, encrypting them, obfuscating them, or repurposing existing protocols. Regardless of the route followed, the net result will be protocols that are less vulnerable to full text analysis.

The author would like to believe that these developers will start a general push that will result in more protocols being secured. More secure protocols may, in turn, result in stronger privacy online. This is, however, solely speculation.

Although the NPG and `qcap` may provide incremental improvements in network sniffers and monitors, it should also provide improvements in network security and, most importantly, our understanding of networks. As described above, the NPG can be used to build parsers for existing network protocols. Those parsers should have fewer bugs, and fewer exploitable security holes than parsers designed by hand; meaning that servers implemented with NPG should be more secure.

From an academic standpoint, the most valuable contribution of `qcap` is its ability to fully parse large volumes of application data. This allows researchers to dig deeply into network data, and easily discover exactly what given network streams are doing. Given that researchers do not currently have a "turn key" approach for full stream analysis, the author feels that the academic value of `qcap` and the NPG outstrips any marginal improvements in network sniffing technology.

Glossary

ABNF Augmented BNF, a language for representing context free grammars, often used in RFCs, 33

accept If a grammar accepts an input, then the grammar recognises an input, meaning that the grammar can generate the input, 20

alphabet A set of tokens legal in a language, 19

APF A language for specifying the format of datagrams, proposed by Lambright and Debray in [38], 31

ASN.1 Abstract Syntax Notation One. A framework for exchanging data between hosts on a network. Provides data formatting and marshalling functionality, 33

dynamic length An idiom in protocol design. Indicates that the protocol specifies the length of some field in a preceding field, 28

dynamic text An idiom in protocol design. Indicates that the protocol specifies a terminator for some field in a preceding field, 28

dynamically directed parsing An idiom in protocol design. Indicates that the protocol may use input at token k to modify the set of grammar rules available at token $k + n$, 104

filter A tool used in network monitoring to filter out classes of datagram that are not of interest, 31

GAPA An engine for parsing network protocols, 33

HTTP Hyper-Text Transfer Protocol. Designed to transfer files required for rendering web pages, but now used to transfer information of many types. Runs on top of TCP, 14

NPG Network Protocol Grammar, a context free language devised by the author to describe streaming network protocols, 7

optional sequence operator A repetition operator that indicates that a symbol may repeat zero or one times, 45

Packet Types A language for specifying the format of datagrams, proposed by Chandra and McCann in [10, 40], 31

parse An action that can be performed on a grammar and an input string. Decomposes the input string into constituent parts, as defined in the grammar, 20

recognise An action that can be performed on a grammar and an input string. Tests to see if some string is a member of a language, providing a boolean response, 20

reject If a grammar rejects an input, then the grammar does not recognise the input, meaning that the grammar cannot generate the input, 20

repetition operator An operator that may appear in ABNF or NPG allowing some number of repetitions of a symbol, 44

RFC Request For Comment document. Specifications for protocols published by the Internet Engineering Task Force, 33

stream oriented An attribute of a network protocol. Stream oriented protocols exchange data as a reliable, well ordered sequence of bytes, sent from one host to others, 13

string A sequence of tokens from an alphabet, that may appear in a language, 19

TAP Timed Abstract Protocol. A language for describing behaviour of datagram-oriented protocols, 32

TCP Transmission Control Protocol. Protocol running on top of IP that allows hosts to exchange data in as a reliable stream of bytes, instead of an unreliable stream of packets, 13

XML eXtensible Markup Language. A popular data format language, 34

Bibliography

[1] Netwitness homepage, 2005. `http://www.netwitness.com` Accessed: May 3, 2005.

[2] snort homepage, 2005. `http://www.snort.org/` Accessed: May 10, 2006.

[3] Wireshark homepage, 2005. `http://www.wireshark.org` Accessed: July 31, 2006.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[5] D. P. Anderson and L. H. Landweber. A grammar-based methodology for protocol specification and implementation. In *SIGCOMM '85: Proceedings of the ninth symposium on Data communications*, pages 63–70, New York, NY, USA, 1985. ACM Press.

[6] R. Ball, G. A. Fink, and C. North. Home-centric visualization of network traffic for security administration. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 55–64, New York, NY, USA, 2004. ACM Press.

[7] R. A. Becker, S. G. Eick, and A. R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995.

[8] N. Borisov, D. J. Brumley, and H. J. Wang. Generic Application-Level Protocol Analyzer and its Language. URL: `http://research.microsoft.com/research/shield/papers/gapaFeb102006.pdf`.

[9] CAIDA. Coralreef homepage. `http://www.caida.org/tools/measurement/coralreef/` Accessed: June 26, 2006.

[10] S. Chandra and P. J. McCann. Packet types. In *Workshop Record of WCSSS '99: The 2nd ACM SIGPLAN Workshop on Compiler Support for Systems Software*, pages 94–102, Atlanta, Georgia, May 1999.

[11] Clearsight. Clearsight analyzer homepage, 2005. `http://www.clearsightnet.com/products-analyzer.jsp` Accessed: May 4, 2005.

[12] Colasoft. Colasoft capsa, 2005. `http://www.colasoft.com/products/capsa.php` Accessed: May 3, 2005.

[13] W. W. W. Consortium. eXtensible Markup Language (XML) Homepage. `http://www.w3.org/XML` Accessed: May 3, 2006.

[14] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 4234 (Draft Standard), Oct. 2005.

[15] Debian Project. Debian Security Advisory 1049-1. `http://www.debian.org/security/2006/dsa-1049` Accessed: June 7, 2006. Details Ethereal's vulnerability to CVE-2006-1932, CVE-2006-1933, CVE-2006-1934, CVE-2006-1935, CVE-2006-1936, CVE-2006-1937, CVE-2006-1938, CVE-2006-1939, CVE-2006-1940.

[16] J. Deverick. Personal communications, 2005. Discussion between the author and James Deverick at the Large Installation System Administration conference, held in San Diego.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[18] G. A. Fink, P. Muessig, and C. North. Visual correlation of host processes and network traffic. In *VizSEC*, page 2, 2005.

[19] C. N. Fischer and J. Richard J. LeBlanc. *Crafting a compiler with C.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[20] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[21] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard), Nov. 1996. Updated by RFCs 2646, 3798.

[22] D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages (2nd ed.).* Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[23] GNU Project. Bison homepage. `http://www.gnu.org/software/bison/` Accessed: January 30, 2006.

[24] GNU Project. Flex homepage. `http://www.gnu.org/software/flex/` Accessed: January 30, 2006.

[25] B. Gregg. Chaosreader homepage, 2004. `http://users.tpg.com.au/bdgcvb/chaosreader.html` Accessed: May 3, 2005.

[26] D. Grune and C. J. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood, Chichester, England, 1990.

[27] U. C. Guard. Team coordination training student guide. `http://www.cgaux.info/g_ocx/training/tct` Accessed: May 4, 2005.

[28] P. Herman. tcpstat homepage. `http://www.frenchfries.net/paul/tcpstat/` Accessed: May 1, 2005.

[29] R. Herriot, S. Butler, P. Moore, R. Turner, and J. Wenn. Internet Printing Protocol/1.1: Encoding and Transport. RFC 2910 (Proposed Standard), Sept. 2000. Updated by RFCs 3380, 3381, 3382, 3510, 3995.

[30] E. Hughes and A. Somayaji. Towards network awareness. In *Large Installation System Administration Conference (LISA'05)*, Dec 2005.

[31] N. Instruments. Observer homepage, 2005. `http://www.networkinstruments.com/products/observer.html` Accessed: May 4, 2005.

[32] ITU-T and ISO/IEC. Asn.1 homepage. `http://asn1.elibel.tm.fr/` Accessed: May 3, 2006.

[33] M. Jayaram and R. Cytron. Efficient demultiplexing of network packets by automatic parsing. In *Workshop Record of WCSSS '96: The Inaugural Workshop on Compiler Support for Systems Software*, Tuscon, Arizona, February 1996. ACM SIGPLAN.

[34] S. Johnson. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/` Accessed: February 13, 2006.

[35] J. Klensin, N. Freed, and K. Moore. SMTP Service Extension for Message Size Declaration. RFC 1870 (Standard), Nov. 1995.

[36] C. Kreibich. Netdude homepage. `http://netdude.sourceforge.net/` Accessed: June 26, 2006.

[37] K. Lakkaraju, W. Yurcik, and A. J. Lee. NVisionIP: netflow visualizations of system state for security situational awareness. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 65–72, New York, NY, USA, 2004. ACM Press.

[38] H. D. Lambright and S. K. Debray. Apf: A modular language for fast packet classification. URL: http://www.cs.arizona.edu/people/debray/papers/filter.ps.

[39] C. P. Lee, J. Trost, N. Gibbs, R. Beyah, and J. A. Copeland. Visual firewall: Real-time network security monitor. In *VizSEC*, page 16, 2005.

[40] P. J. McCann and S. Chandra. Packet types: abstract specification of network protocol messages. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[41] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Conference*, pages 259–270, 1993.

[42] T. McGuire. Austin protocol compiler homepage. `http://apcompiler.sourceforge.net/` Accessed: May 1, 2006.

[43] T. McGuire. *Austin Protocol Compiler Homepage*. PhD thesis, University of Texas at Austin, 2004.

[44] J. McPherson, K.-L. Ma, P. Krystosk, T. Bartoletti, and M. Christensen. Portvis: a tool for port-based detection of security events. In *VizSEC/DMSEC '04: Pro-*

ceedings of the 2004 ACM workshop on Visualization and data mining for computer security, pages 73–81, New York, NY, USA, 2004. ACM Press.

[45] A. Oline and D. Reiners. Exploring three-dimensional visualization for intrusion detection. In *VizSEC*, page 14, 2005.

[46] S. Ostermann. tcptrace homepage. `http://www.tcptrace.org` Accessed: May 3, 2005.

[47] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2002. ACM Press.

[48] G. Palidwor. Personal communications, 2006. Discussion between the author and Gareth Palidwor, regarding the possibilities for grammars that are first class objects.

[49] L. Peterson. x-kernel homepage, 2006. `http://www.cs.arizona.edu/xkernel/` Accessed: June 28, 2006.

[50] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[51] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFC 3168.

[52] J. Postel. Simple Mail Transfer Protocol. RFC 821 (Standard), Aug. 1982. Obsoleted by RFC 2821.

[53] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), Oct. 1985. Updated by RFCs 2228, 2640, 2773.

[54] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. `http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps` Accessed: June 26, 2006.

[55] C. Shannon and D. Moore. The spread of the witty worm. `http://www.caida.org/analysis/security/witty/` Accessed: June 7, 2006.

[56] tcpdump workers. Tcpdump public repository. `http://www.tcpdump.org` Accessed: May 12, 2006.

[57] D. Waitzman. Standard for the transmission of IP datagrams on avian carriers. RFC 1149 (Experimental), Apr. 1990. Updated by RFC 2549.

[58] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204, New York, NY, USA, 2004. ACM Press.

[59] R. Wojtczuk. libnids homepage, 2005. `http://libnids.sourceforge.net/` Accessed: May 5, 2005.

[60] G. Wurster. Personal communications, 2005. Glenn Wurster gave an in-house talk in the lab on using parsers for input processing on servers.

[61] X. Yin, W. Yurcik, M. Treaster, Y. Li, and K. Lakkaraju. Visflowconnect: netflow visualizations of link relationships for security situational awareness. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 26–34, New York, NY, USA, 2004. ACM Press.