Department of Computer Science

Technical Report

Undergraduate Dissertation: Visualisation of IPTABLES

Florian Schanda

Technical Report 2004-10 **July 2004**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Visualisation of IPTABLES
BSc in Computer Software Theory

Florian Schanda

July 6, 2004

Abstract

Netfilter/iptables is the incredibly powerful packet filtering mechanism used in Linux 2.4 and 2.6. Due to its potential complexity there exist two common attempts at graphical user interfaces: wizards and front ends to the `iptables` program. Both of these have fundamental problems: wizards (task based solutions which oversimplify the problem) loose too much of the power and flexibility, and front ends have the same problem the command line inteface has: Non-trivial firewalls become very confusing due to the lack of an adequate representation.

A new approach using a graphical representation of the firewall and its rules is developed. Various possible visualisation methods are researched and one concept is implemented to show what the firewall does and how its rules are connected.

To further ease the task of designing an effective firewall, a simulation method is developed allowing the user to see how any given IP packet is dealt with.

Acknowledgements

My Supervisor, Dr. Russel J Bradford
For being a very helpful project supervisor with lots of good advice and ideas

My Father, Dr. Friedrich Schanda
For initial proofreading

My Brother, Johannes C Schanda
For more proofreading

Contents

Chapter 1

Introduction

1.1 Netfilter/iptables

Linux versions 2.4 and 2.6 primarily use netfilter as their packet filtering system. `iptables` is the user-space command line tool to manipulate the firewall rules in the kernel. The Netfilter/iptables system is very powerful and flexible, and firewalls are not always easy to understand.

The netfilter system is based on relatively simple rules. Rules have a number of conditions and an optional target. If all conditions match, the target is executed. Example targets can be simple things such as to DROP or ACCEPT a packet, or more complicated operations such as to LOG a packet or to modify its contents. Rules themselves are arranged in chains. For instance there exists a chain called INPUT, which is used to decide the fate of incoming[1] packets.

To define complex behavior (such as to first LOG a packet and then drop it) an arbitrary number of custom chains can be defined by the user. These are very similar to functions (or subroutines) in procedural programming languages.

Chains themselves are collected in a structure called a table. A typical netfilter system has three tables: filter, mangle and nat. These will be discussed in detail later.

1.2 The Problem

The problem becomes obvious, once one has seen a firewall of even moderate complexity[2]. Relations between rules, flow of control and the general firewall behavior become very hard to see. Consequently, bugs in the firewall can sometimes be very hard to spot. Similarly, testing a firewall is very hard at the moment: You have to be confident you perfectly understand all the rules you have used, and all types of packets you could encounter.

Additionally, using the command line tool `iptables` to modify a firewall can be a time consuming task, due to the potential number of rules and chains involved and the way `iptables` operates.

1.3 Current Solutions

There currently exist two very different attempts to solve this problem, wizards and pure front ends.

1.3.1 Wizards

One type of solution tries to oversimplify and abstract the problem away: These programs (usually called wizards) provide simple task-based interfaces to construct a corresponding firewall. Examples of these are found in many Linux distributions[3]. The problem with this approach is that power is sacrificed for simplicity.

1.3.2 Front-Ends

This class of programs provides a graphical user interface to iptables. They deal with data exactly the same way iptables itself does, and thus loose none of the power and flexibility of netfilter. However,

[1] Packets designated for local processes, not for routing
[2] About 40 rules and 5 custom chains
[3] Such as in YaST2 in SuSE Linux for instance

in the end they still have the same problems iptables itself has, unless they were to use a novel representation of the firewall: larger firewalls will be very confusing to look at, and thus hard to understand.

1.4 A new approach

The aim of this project is to investigate a new approach of dealing with this problem. A solution which does not loose any of the flexibility netfilter offers, but highlights flow of control and relations between rules. A visualisation of the rules and chains, along with an editor to modify and create firewalls and a system to test the resulting firewall will be researched.

The resulting system could also feature wizards as an alternative[4] way of creating new firewalls, but this is not a priority.

[4] Easier for novice users

Chapter 2

Literature Review

2.1 Introduction

Netfilter is the packet filtering, routing and mangling system used in recent Linux kernels. It is used to build complex firewalls or routers with NAT and IP masquerading. It is a fairly low-level system, consisting of chains of rules. As usual with low level systems, they are very powerful and flexible, but can be hard to use. In the case of `iptables` (the userspace front end to netfilter) it is especially hard to see relations between rules and chains if there are many of them.

This literature review will look at some aspects of netfilter/iptables firewalls and their visualization, the problems with current approaches and why a new solution is necessary.

The official packet filtering howto[5] and NAT howto[6] were very helpful in understanding how to use to use the user-space part of the netfilter/iptables system.

2.2 Visualisation

2.2.1 Aim

The general idea is to visualize the rules and chains of iptables. The emphasis is placed on the ability to see connections between related rules and chains, and not to just provide a front-end to iptables. The idea is that a good visualisation will also make it easier to edit existing (or create new) rulesets. I was unable to find any previous work done in this area, and indeed academic work done on the iptables is very scarce and only deals with firewalls as security measures, and not with visualisation.

This is the main reason all references cited are essentially weblinks; most of the work and research done in this area is published on the web, in the form of RFC's, documentation, howto's and manuals, and not in papers.

2.2.2 Current Tools

There currently exist a number of GUI's and wizards for iptables. They however have one thing in common: they do not do what we want to do. The wizards tend to hide the underlying rules from the user, and usually just provide simple forms to the user.[1] The GUI programs take a different approach, showing all the rules to the user, however have one thing in common: they are a front-end to iptables; meaning they produce a nicer version of `iptables -L`[2] The way these tables are presented varies from tool to tool.

The only project found with the title "iptable visualisation tool"[11] was a Perl script of about 25 lines posted to the Linux Networking Usenet Group, and said script only produced a tabular representation and thus did not try to do what we want to do.

There do, however, exist some good GUIs which are a bit closer to the aim. Bifrost[7] for instance is a web-based tool providing a kind of hybrid form between a pure frontend and a wizard: all rules are there if the user wants them, but most of the time you will only see the rules relevant to the current task. They provide a useful demonstration[8] site. An advantage of using a web-based solution is that remote administration of such a system is very simple. From the visualisation point of view this

[1] This is the usual way Linux distributions handle firewalls, an example would be the firewall component of SuSE's YaST2 tool.

[2] This command prints out all iptables rules active on a system.

software is not interesting since it restricts itself to tables of raw data, and as with all web applications not using Java, it has limits to its interactivity.

Firewall Builder[9] takes a different approach by being a "real" application, using the GTK tookit. The advantage of being a real application is twofold: performance and responsiveness is higher and complex graphics can be expressed easier (and the application does not depend upon a web-browser to render it correctly). However this tool again only provides a front end to iptables and all its data is displayed in tables.

2.2.3 A new direction

Each of these tools have one thing in common: they display the data as `iptables` sees it (tabular), and do not take into account the possibility of displaying the rules and chains in an alternative format with emphasis on relations. *No currently existing* tool was found which takes advantage of a more graphical representation of a firewall.

What is needed is an entirely new approach, and one of the main goals of this project is to reseach if and how such a visualisation can be done. As virtually *no work has been done before in this particular area* (visualisation of iptables), this will be the main aspect of the project: Find an adequate and helpful visual representation of netfilter's rules, chains and tables.

2.3 Interface

2.3.1 Toolkit

The user interface of the tool will be as simple as possible, since most of the space on screen will be taken up by the visualisation of the rules and chains. The author speculates that it is unlikely for the UI to even include a toolbar - all we need is a menu. It would, however, be unwise to restrict ourselves to just a basic UI toolkit.

The best solution is to use one of the many open source toolkits, which is capable of providing all the simple UI elements, and allows the creation of custom UI elements. Such a toolkit needs to fulfill three requirements: it must be free both in cost and usage, it must be powerful enough provide a way to draw a picture (or provide a pixbuf like control), and must be available in the implementation language (C or C++). Two very popular toolkits which fulfill these requirements are GTK+ and Qt, however there are also others like wxWidgets (formerly known as wxWindows).

The GTK+[12] tookit was originally designed for the GIMP and thus provides support to draw pictures. It is implemented in C, but a C++ binding called gtkmm[13] exists. GTK is popular since it is fairly well documented and widely used, for instance in the GNOME desktop environment. Although it is written in C, it was designed with objects in mind; thus making it easy to combine and extend current widgets. GTK has also some interesting accessibility features in newer versions.

The Qt[14] toolkit is a C++ toolkit which is also widely used, the most popular example being the KDE desktop environment. Its documentation is both indepth and easy to use, and the library itself comes with many example programs. Since it is written in C++ it is very object orientated and easy to handle. Another nice aspect is that a Qt application blends perfectly in to a KDE desktop, but does not require KDE to run. One of Qt's strongest points is its very consistent look and feel and that it is maintained very well by its developers. It includes several ways of presenting pictures, such as the **QCanvas** or the **QImage** class. One quirk with Qt is that it uses its own special C++ preprocessor to implement polymorphism, the designers of Qt choose this approach over C++ templates because not all compilers implemented them properly at the time. Additional properties of Qt relevant to the project are highlighted in appendix A.

The wxWidgets[19] toolkit is another C++ toolkit (which uses GTK+ or Motif to render its widgets on UNIX) which is used by some applications (such as xMule.) It seems to have a solid design, and like GTK+ does not require any preprocessing on its files. It should also be noted that it is multi platform like all of the other toolkits examined.

Although no final choice has been made, the author is already very familiar with Qt, and Qt fulfills all requirements mentioned above, so the project will most likely use it. It should finally be noted that the user interface around the visualisation is not as important as the visualisation itself.

2.3.2 Design

The interface of an application should be consistent throughout the application, and possibly adhere to some guidelines. Since the toolkit will most likely be Qt, and Qt is expected to blend in well with

KDE the project will follow the sensible standards and guidelines[20] the KDE project uses. Most of it is pretty much accepted rules and common sense; both documents draw upon other good approaches (like the MacOS interface guidelines).

2.4 Other Aspects

2.4.1 Extensions

The netfilter system is very extensible, and a number of interesting extensions exist. A howto[27] on the netfilter website explains how netfilter and iptables deals with extensions. The technical side of adding extensions[28] to the netfilter system is also explained on the main website.

Clearly, extensions are an important part of the iptables system, so our tool must take them into account as well. The reviewed documentation explains what kind of extensions one might come up with, and what ways exist at the moment to describe them. There are conceptually three ways to deal with extensions: Patches, plugins and descriptions.

An obvious way to deal with new, unexpected functionality netfilter provides is to write a patch to the tool and recompile - this is probably not the best solution for a number of reasons, such as maintainability and complexity.

A very commonly used way to provide new functionality to a program without actually patching its source is to provide a ABI[3] and support plugins. A widely known example of plugins are web-browsers such as Mozilla, which use a number of plugins to display multimedia and other non HTML content.

The third and probably most appropriate method is using "description" files which, if parsed by the program provide it with enough information to implement the new functionality. Such files could be of any format, for instance XML as a purely descriptive language, or perhaps a scripting language. One example of an application using XML to extend its functionality is dia[23]: It uses XML and a subset of the SVG language to add new shapes and functionality to the program. In the context of this project such a file could specify where a new rule fits into the netfilter system, what it does, how it relates to other rules, et cetera...

2.4.2 Packet Simulation

The system is required to be able to simulate a packet entering (or leaving) the system, highlighting (or otherwise identify) the rules which match. It should be noted that no netfilter firewall program available can do this yet. Finding a sensible solution to this problem would be a first.

In order to begin constructing a solution to this problem, an indepth knowledge of how networking and the protocols used work is required, along with how the netfilter system operates. A good staring point is provided by another helpful howto[29], the Linux Networking-concepts HOWTO, on the netfilter website. A more indepth resource on how general IP packets look is RFC 791[2]. The three most used protocols TCP[4], ICMP[3] and UDP[1] are specified in RFCs as well (along with all other published IP protocols).

From this reading it is clear that neither the "packet creator" nor the act of simulating a packet traversing a netfilter ruleset will be a trivial piece of work, and requires careful consideration as to which protocols should be supported directly and if there is a generic way to support any possible protocol (perhaps by limiting the tool to be able to generate or simulate packets for the most commonly used protocols only).

An alternative to hand crafted packets are of course real packets. Several programs exist to capture them, such as the ancient (but very popular) `tcpdump` program.

Some programs dealing with packets directly such as `ethereal`[24] manage to display packets in a list and by clicking on them additional information is revealed. Although this representation is useful for many packets, it does not provide editing facilities as such. It does, however, give a general idea of how one might go about describing packets in a simple way.

Either program can save sniffed packets into a file for later use, our tool could use the same file format for convenience. The tcpdump source code (also available on the tcpdump website) explains how this (binary) file format works.

[3] Application Binary Interface

2.5 Conclusion

Networking is a complex area and a significant chunk of the time spent working on this project will be to research and understand the relevant elements of this topic. Although no academic papers as such contain the information needed, the internet RFC's provide an excellent and exhaustive source of information on the matter.

This project is interesting because several aspects have never been done before, and have little or no literature to review available. The two key aspects (and simultaneously the ones that have never been done before) of the project are:

- Finding a sensible graphical representation of a netfilter firewall, which allows the user to spot relations between rules and also supports easy editing (or creation) of firewalls.
- Finding a way to simulate packets going through a firewall, highlighting the path taken and the final fate of a packet.

Overall there is a lot of information available, since all widely used protocols are open standards. There are also a lot of other firewall tools of varying degree of usefulness and usability. However, none of them solve either one of the two key aspects this project tries to tackle. The project is most definitely worth doing because it would potentially solve two unsolved problems, and produce a useful tool in the process.

Chapter 3

Dissertation

3.1 Introduction

3.1.1 Conventions

In this document the following conventions to indicate objects shall be used:

Font	Meaning
QFile	A C++ class called QFile
`/etc/protocols`	A file
`iptables`	The program iptables
foo()	A function or member function called foo

3.1.2 Background and Terminology

Firewall In the context of this document *firewall* means any netfilter/iptables firewall. Such a firewall can contains a number of tables.

Tables Netfilter usually features three separate tables: filter, mangle and nat. The filter table deals with packet filtering, i.e. accepting or dropping certain packets. It contains three chains which deal with incoming packets destined for this machine, incoming packets to be forwarded to other machines and packets generated on the local machine.

The nat table deals with modifications to packets related to network address translation (known as DNAT, SNAT and MASQUERADING in Netfilter) and port forwarding.

The mangle table deal with other modifications to packets. This can be used for things like user-space packet queues and packet scheduling.

As of Linux Version 2.6.6 a new, optional table called raw is available. It can be used to bypass all other tables. At the time of writing Linux version 2.6.6 was not released yet, and thus this table will not be supported in the program. (See section 3.8.6 for more information.)

Chains Each of the standard chains has a so called policy. It is the action to perform if none of the rules in it decide the definite fate of a packet. Per default it is ACCEPT, i.e. any packet not dropped during that chain is accepted.

User defined chains do not have a policy, they return control to the calling rule if none of the rules decide the fate of the packet. The name of any user defined chain is also a valid target for any rule.

Note that even if a rule matches in a chain, the packet can still continue to traverse it depending on the target of the matching rule. For instance the LOG target only logs the target, but does nothing else. The only standard targets which decide the fate of a packet are ACCEPT, DROP, REJECT and QUEUE.

Rules Rules themselves have two main features: a target and conditions (both of which are optional). The target specifies what happens to a packet if all conditions match, and can be things like to ACCEPT or DROP a packet, LOG a packet or call a user defined chain.

The conditions can cover a wide range of possible options, from simple things like source and destination ip addresses to the specifics of the used protocol (such as TCP connection states or port numbers).

If we cannot determine if a condition matches or not[1], we assume it does not match.

3.1.3 Goals

The overall goal of this project is to develop and implement a graphical visualization of any given netfilter firewall along with a complete firewall editor. Also a method of highlighting matching rules, given an IP packet, in order to simplify testing of the created firewall needs to be found. The high level functional requirements of the software (subsequently also referred to as 'the tool') are:

1. The tool must be able to save and load a firewall.
2. The tool should be able to import a firewall from some standard format. (Such as the output of the `iptables-save` program.)
3. The tool must be able to export a firewall to a shellscript which can be used in system startup.
4. The tool must be able to create or modify any (netfilter based) firewall.
5. The tool should be able to deal with the numerous extensions to netfilter in a sane way.
6. The tool must use a graphical representation of the firewall and not be a mere front-end to iptables.
7. The tool itself should not require iptables to be installed[2].
8. The tool must be able to allow the editing of any given rule in a firewall.
9. The tool should be implemented in a way to allow IPv6 in future versions.
10. The tool should provide a way to simulate a packet going through the firewall, possibly highlighting matching rules.

3.1.4 Tools and Technologies

The toolkit

As the main component of the software will be its user interface the toolkit used to create this interface should fulfill the following requirements:

- Have bindings in one of the major[3] languages.
- Be fully documented.
- Be reasonably portable, i.e. run on Linux and other popular UNIX flavors.
- Have an open source license. This is important, as it guarantees the future free availability of the chosen toolkit, and also gives the ability to study and modify its source in case bugs are found during the implementation of the project.

The following were examined: GTK+ and Qt.

GTK+ This toolkit was originally designed for an image manipulation program called gimp. It has come a long way since then, and is now used in a wide range of applications, including the gnome desktop. It is written in C, but has bindings for many languages including C++. It is reasonably well documented. It is portable across a number of platforms including Linux and Solaris. Being an open source effort, its LGPL license is no problem.

Qt This toolkit, although open source, is primarily maintained by the company Trolltech and was designed as a general user interface toolkit. It is written in C++, but also has some bindings for other languages, such as Python. It is portable across all major platforms, including Linux and Solaris. It is dual licensed under a commercial license and the GPL. Its documentation is excellent.

[1] For instance we can filter for the source port in the first fragment of a TCP/ip packet, but not in the second or further fragments.

[2] In order to make it more portable, and to allow for "offline" firewall creations.

[3] C, C++, Python, Perl, Java, CommonLISP

In the end both options were worthy candidates, but Qt was chosen for the following reasons:

- The author was vaguely familiar with both Qt and Gtk+, but liked the consequent and clear object oriented approach of Qt
- The documentation of Qt was found to be easier to use
- The authors desktop is running KDE (which is built on Qt), and thus properly learning Qt was deemed to be useful for the future
- Qt includes not only user interface classes, but many other utility classes (such as an excellet string class)
- Qt includes a fully fledged XML parser and DOM[4] facilities
- The program Qt Designer which comes with the Qt library is a useful tool to generate user interfaces

Appendix A provides a brief introduction to Qt and explains some key concepts and terminology used throughout the project and this dissertation.

Programming Language

Since Qt is written in C++ (and the author likes C++), the obvious choice is to use the same language to implement the project. Bindings do exist, however, for instance PyQt is a Python binding for Qt.

XML

The general file format throughout the project will be XML for the following reasons:

- Building parsers for XML is very simple, and Qt already includes one
- XML is in the spirit of cross-platformness since the format is well specified and understood
- Well designed XML files can be easily created by hand.[5]
- XML is generally regarded as a good idea, and many existing applications successfully use it for many different purposes: Dia[23] uses it to describe shapes, the editor Kate[21] uses it to define syntax highlighting rules, the web-browser konqueror[22] uses it to define its user interface.

Indeed even during the building process of the project XML will be used: Qt Designer creates XML descriptions of user interfaces which will then be transformed into C++ code by the Qt utility `uic`.

[4] Document Object Model

[5] This is useful especially for testing and in the very early implementation phase.

3.2 Shortcomings of current solutions

3.2.1 Wizards

Firewall wizards create standard rulesets (of varying quality) with specific properties. They usually come with a nice user interface and are aimed at users who do not have much experience with netfilter or iptables, and just want to create a basic firewall.

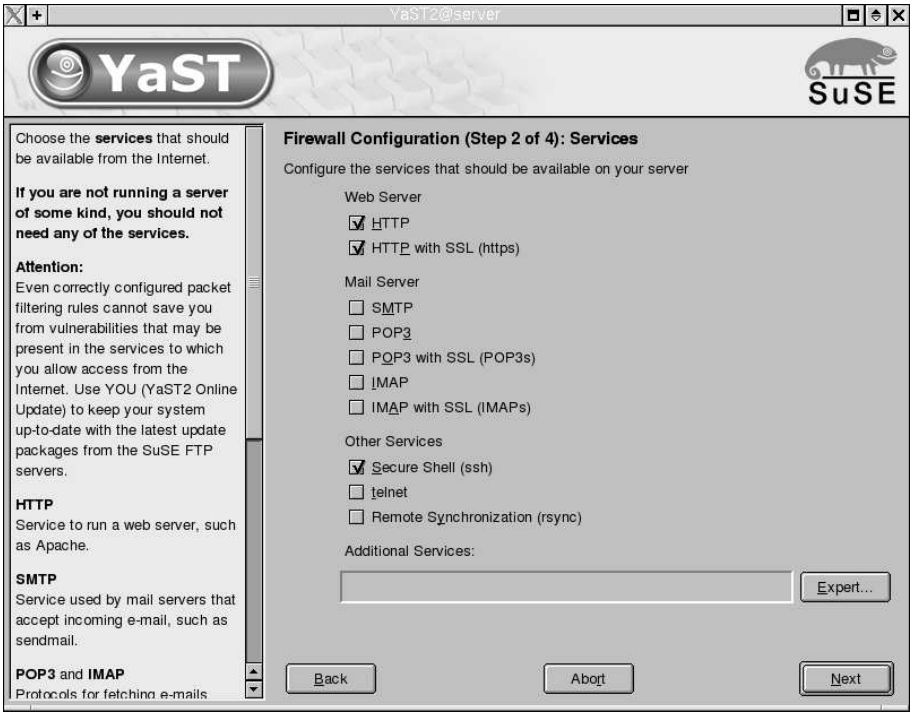

Figure 3.1: YaST2 Firewall Configuration

Most Linux distributions include some such wizard in their configuration tools; figure 3.1 shows YaST2[10] as a representative example. Typical tasks they can perform include creating basic firewall from scratch, forwarding or opening specific ports, network address translation setup, basic routing and blocking specific services.

Although the potential power of netfilter is much greater, wizards do not aim to be replacements for `iptables`. They exist only to simplify some tasks, and thus don't solve our problem of visualising and creating any firewall. Indeed, some wizards (such as YaST2) abstract so far, that there is no longer the notion of a *rule*, but of some high level emergent *properties* of a firewall.

Even though wizards on their own are pretty useless for our purposes, they can be useful to make a front-end program more user friendly. One such program for instance is Firewall Builder[9] which offers a wizard in addition to its base functionality in order to make it easier for novice users to get started. The approach of an optional wizard is generally regarded as a good idea, since it is entirely optional and an advanced user can just ignore it completely.

As a wizard is not part of our basic requirements, this idea will be revisited in section 3.8.5.

3.2.2 Pure Front Ends

Front ends to iptables approach the problem from a different way: They provide an user interface around the functionality of the `iptables` program. One such program is Firewall Builder[9], figure 3.2 demonstrates that is is basically a graphical version of `iptables -L`[6].

The main goal of such front ends is to provide a graphical user interface which maps more or less one to one to iptables. They do not try to hide or simplify information as wizards do, but to wrap up the functionality of the command line tool using common GUI elements[7]. This usually makes the program slightly easier to use, as such GUIs tend to present all possible options at once, from which the user can then choose the ones desired.

[6] This command lists all rules in any given firewall table

[7] Such as tables (or grids)

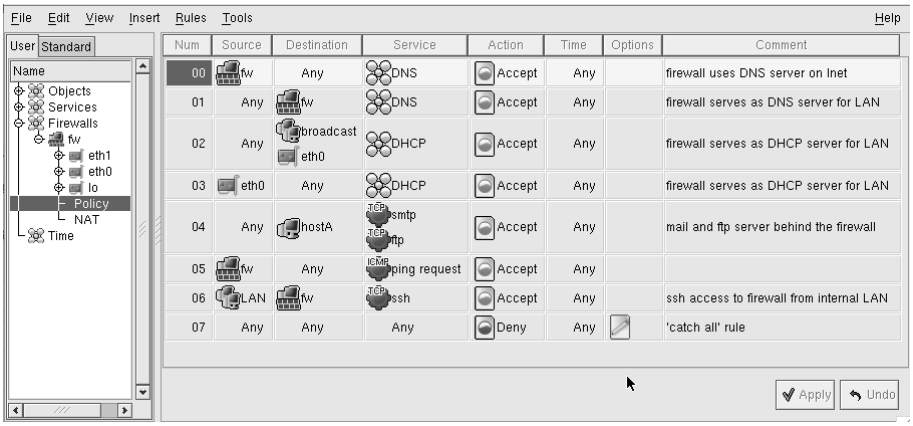

Figure 3.2: Firewall Builder

This is a different philosophy compared to command line utilities such as `iptables`, because the latter requires the user to know the options it supports. Although `iptables` does provide information on possible values, this step (of retrieving the possible values) is separate from creating content (the firewall).

Even though these concepts are very helpful in creating and modifying firewalls, the problem of presenting the firewall is still not solved. If one refers to figure 3.2 it can be seen that the information presented is nearly identical to `iptables -L`. Being presented in a graphical table and using small pictures to make the information presented clearer does not change this fact. In other words a big and complicated firewall will still be very hard to understand by just looking at the table.

As one of the main aims of this project is to research a good visualisation of a firewall, which enables one to easily discern the flow of control and relations of the rules, this approach (of creating a GUI version of `iptables -L`) cannot be chosen. There is, after all, little point in doing something that has been done before unless we can do better.

As mentioned earlier in the literature review, there exists much prior art in both wizards and pure front ends, however nobody set out to do what we are about to do: implement a new approach at creating, modifying and most importantly visualising firewalls.

3.3 Data storage and retrieval

3.3.1 Internal Format

The internal format used to save and load firewalls will be an XML based format. This approach was chosen rather than some binary format (or even iptables shellscripts) because:

- There already exists an XML parser in Qt
- Possible to read and write with just a text editor (useful for debugging purposes)
- In case other non-iptables information has to be stored in the file (such as simulation data or virtual hardware) that is not supported in other file formats. Since an XML DTD is easy to extend this is possible without too much trouble.
- A lot of other aspects of the project also use XML; having a XML save file seemed only consequent

3.3.2 Generating Shellscripts

To generate a shellscript which, when run, will load the netfilter firewall into the Linux kernel, the tree-walk approach seems the easiest. Each rule object, chain object and table object in the system will have a member function which converts itself to `iptables` syntax.

Section 3.8.2 deals with possible further work regarding the generation of `iptables` shell scripts.

3.3.3 Importing Firewalls

Initially, efforts were made to parse the output of `iptables -nvL` to import a firewall from a dump file or from the running kernel. This approach soon showed several major problems:

- The output of **iptables -nvL** is not regular at all, and a whole lot of extra information would have to be added to the extension XML files.
- A rather complicated lexer would have been necessary to pick apart the string describing the extensions.
- Three files (or three invocations if **iptables -nvL** are needed to obtain the complete firewall: one for each table.

Since the problem seemed so hard, investigation was started to look at how other programs solved this problem. After consulting sections of the `iptables` man page again, it seemed `iptables-save` provides a much nicer output to parse compared to `iptables -nvL`. Thus the decision was made to use `iptables-save` instead of plain `iptables -nvL` for the following reasons:

- It is far easier to parse
- One file (and one command) obtains the entire firewall, instead of just one table.

3.3.4 Implementation notes

The only challenging bit was to import firewalls from `iptables-save` output; several issues had to be taken care of: initially a class **TrivialParser** was used to segment the input into sub strings. It was a essentially wrapper around the member function *section* of the **QString** class.

Then an exceptionally irritating bug turned up where an imported firewall had an extension option argument such as `"foo "` (note the space and the quotes). This old parser did not recognize strings like that and the result were two arguments, `"foo` and `"`. The class was then rewritten to support more complicated tokens like that.

3.4 Extensions

3.4.1 Introduction

Before we can begin to develop any useful ideas for visualisation, we will take a closer look at iptables itself and the datastructures used to represent rules, chains and tables. Any given netfilter rule has seven basic properties: protocol, fragmentation, source, destination, input interface, output interface and target.

Protocol This (invertible) condition can match IP protocols by name (such as tcp or udp) or number (such as 1 for icmp). A full list of protocols and their number[8] is usually found in the file `/etc/protocols` on standard UNIX systems. Section 3.8.10 will look at how this file could be used in the project.

Fragmentation This flag will match any second or subsequent fragments of an ip packet. If specified invertied, only the first fragment of a fragmented packet is matched. If the flag is not specified at all, any packet (including a fragment) matches. Usually it is not necessary to specify this flag at all, only is special circumstances. The reason for this is if a match in Netfilter cannot be determined (such as UDP source port on a TCP packet) it defaults to false. Furthermore, if the first fragment of a fragmented packet is dropped, so is the rest of the packet due to the inevitable reassembly error. (It would be unecessary to drop the fragments as well.)

Source and Destination These (invertible) fields match source and destination ip addresses with an optional bitmask to specify entire networks. Alternatively the `iptables` program also accepts names (which are converted to IP addresses for the internal kernel representation).

Input and Output Interface These (invertible) fields can be used to test for the network interface the packet was received on and will be sent out on. Some matches obviously don't make sense, such as matching for the output interface in the INPUT chain.

These six options all deal with the IP protocol only, and are thus universal to any packet netfilter can filter. So far we cannot even filter out specific ports; ports are a TCP or UDP concept and are not part of IP. To deal with these more advanced matches netfilter uses *extensions*. Any extension can be one of two kinds: target or match.

Target Extensions

Target extensions can be specified as a target to a rule. (The three standard builtin targets which do not require extensions are ACCEPT, DROP and QUEUE.) Target extensions can decide on the fate of a packet (such as the REJECT extension, which drops and rejects a target properly with an ICMP error message), perform some action (such as the LOG extension, which logs information of the packet to the system log) or modify a packet (such as the SNAT or DNAT extension which performs source or destination network address translation respectively).

A target extension can also have a number of additional options (such as the log level in case of the LOG extension).

Target extensions to netfilter, like all netfilter extensions, are part of the Linux kernel, and must be compiled in or be available as a module in order to be usable. Additionally, `iptables` must also know about them in order for the project to use them in any meaningful way. To find out which extensions a Linux system supports one can consult the file in the proc filesystem `/proc/net/ip_tables_targets` on both 2.4 and 2.6 kernels.

Match Extensions

Match extensions can be specified as additional conditions to a rule. The general format is to specify an extension and then some (optional) conditions. Some match extensions do not have or need any options, an example would be the *unclean* extension to match strange packets. The majority of the match extensions however do require options.

In particular, we use match extensions to filter for ports. Both the *tcp* and *udp* match extension provide an option to match any of the source or destination port of a packet.

[8] for instance the icmp protocol = 1, tcp = 6 and udp = 17

Match extensions are not restricted to protocols above IP, some match extensions operate on a lower level. For instance the *mac* match extensions can be used to filter for ethernet MAC addresses.

3.4.2 The Problem

The problem here is that netfilter is extensible in virtually any way. For instance a bored hacker could write a match extension to match only packets that contain the byte sequence `mummification of happy vikings` and whose IP header checksum ends with `11001`. Although this is an rather extreme example, it illustrates the point: anything is possible, and there is simply no generic way of describing all extensions.

Thus hardcoding the existing extensions into the project will be a clumsy approach at best, and that doesn't even solve the major problem of simulating any random packet going through the firewall. To summarize, there are two aspects to the problem:

- Capturing extensions and their options, so that rules containing them can be created, edited and displayed on screen.
- Applying the rules to simulated IP packets for the purpose of simulation.

3.4.3 Representing extensions

The first problem is a relatively easy one: all that is needed is some kind of sensible description file format and one file per extension. The obvious choice for this kind of thing is an XML file, as it is both easily created and easily parsed. A simple example of such a file is the extension file dealing with the *udp* match extension:

```
<extension type="match" name="udp" simulation="ext_udp.so">
        <option name="source-port"
        type="portrange" caninvert="true" postinvert="true">
                <alias>sport</alias>
        </option>

        <option name="destination-port"
        type="portrange" caninvert="true" postinvert="true">
                <alias>dport</alias>
        </option>
</extension>
```

The outermost tag *extension* declares an extension. Its attribute 'type' can be either match or target depending on the nature of the extension. Its attribute 'name' is the value one would provide the `iptables` program with in order use it. The optional 'simulation' attribute specifies the name of the plugin to load that enables the system to simulate a packet being matched by a rule using this extension. The optional *option* tags declare options for this extension.

The (optional) *alias* tag inside declares other names this extension option is known as. (The only extensions requiring this are the *tcp* and *udp* extensions.) A complete specification or DTD of this XML file format can be found in Appendix D.

The *type* attribute of the *option* tag deserves special attention: it can be one of many predefined value types (for a complete list see Appendix D) which enable the program to present appropriate editors for the different options.

The other two attributes, *caninvert* and *postinvert* deal with how and if an option can be specified as an inverse. There are three possible ways: No inverse possible, the inverse symbol (!) precedes the option (such as the `[!] --syn` option of the *tcp* extension) or the inverse symbol is specified after the extension option name (such as the `--dport [!] <portrange>` option of the *tcp* extension).

One file exists per extension (be it target or match), and even though the file name can be unrelated to the extension it contains, all extension files provided with the project are of the format `extensionname.ext`. In other words the tcp extension is contained in the file `tcp.ext`, although we could have put it into a file called `foobar` instead. A dedicated directory (called `extensions/`) exists which is scanned during the initialization phase of the program, and any extensions found in it are loaded.

If a new extension for netfilter is created, its author must (as usual) first modify `iptables` to support it. Finally an extension file describing the extension and its options must be created for our tool to know about it.

3.4.4 Simulating extensions

This problem can be potentially very hard: how should completely arbitrary extensions be represented? The core of the problem is how to encapsulate the functionality of these extensions in a sane way. How to *represent* them becomes obvious once a encapsulation has been chosen. Section 3.5.2 deals with a few possible ways to encapsulate and implement extension functionality for the purpose of simulation packets; and why the shared object plugin based approach was chosen.

The plugin based approach has one definite advantage in representing extension functionality: all the author of an extension has to do is to modify the extension XML file to point to the plugin file for this extension. This file is a ELF shared object, written in C++ using some of the classes used by this project defining an API. It implements all conditions of the extension in C++ functions which are then called by the main program if needed.

Nothing further has to be done with the XML file, the shared object file will contain a initialization function which will connect functions to their extensions at runtime.

3.4.5 Implementation notes

The extension system was one of the first things implemented once the the most basic of the visualisation was ready, since it seemed like a good idea to get the data structures right first.

Since the author did not have any prior experience with XML parsers, it took some time to get used to Qt's approach. The first parser implemented was the one to load the firewall from an XML file. It used only the pure XML parsing functions of Qt. However the extension XML file parser uses the DOM classes of Qt which greatly simplify the parsing, since the parse tree is constructed automatically. The XML parser was the first one to use the DOM approach, and it is very easy to add new keywords to the extension XML file format.

Perhaps the most significant amount of time was spent on determining all of the many possible ways in which options are passed to the `iptables` program. After reading the help for each possible extension, the present classes were constructed. One consequence of this was to leave arguments as unparsed as possible (an argument is always represented by a **QString**) in order to handle the case of unknown extensions or values. In particular this means that even numeric arguments are represented as strings by the **QString** class instead of classint data types. Although this allows for greater flexibility when editing firewalls, it also means a bit more work for the simulation module, as all values have to be parsed then.

Implementing an extension's functionality for the simulation proved to be rather easy as well: the plugins can use all of the main programs facilities, in particular the classes dealing with network frames and IP packets. Thus many extension options were implemented in about three lines of code, as it was merely a matter of comparing the result of a member function call of such as class to the option argument's value.

3.5 Simulation

The single most important feature of the program (for demonstrating how a firewall works) is to simulate any random IP packet (either hand-crafted by the user or captured of the wire[9]) going through the firewall, identifying matching rules and the final fate of the packet. This section deals with the approaches considered and the problems encountered designing and implementing this feature.

3.5.1 The problems

The entire task can be split up into three closely related problems:

- How do we know whether a rule (and possibly the extensions used) matches a packet, and how do we encapsulate this information?
- How are extensions dealt with which are meaningless/impossible to simulate? (Such as the TCP connection state, or the limit extension.)
- How do we visualize the packet going through the firewall?

As mentioned in section 3.4.4, the choice of how to decide if a rule matches or not will directly determine how to represent them.

3.5.2 Ideas and concepts

Hardcoding

A possible solution would be to create a number of functions (one for each extension option) in the program which, given an IP packet, return true or false. Each of these functions could be referenced in the extension XML file by name, or the connection could be made by the program itself based on the name of the extension (or its option). This approach has a few advantages:

- Very easy to implement any kind of extension, as the functions could make use of all the internals the program provides and C++ can cover absolutely anything that can ever be written in netfilter (since the latter is implemented in C).
- Very easy to integrate these functions into the program, as they are already part of it.
- Would be very fast, as there is no overhead of any kind to determine whether a rule matches or not. A C++ function call is all that is needed.

However the disadvantages of this approach are immense:

- Inflexible. Hard to maintain and to add new extension functions, as the entire program would have to be relinked (and certain portions recompiled).
- New extensions would be hard to distribute, as one would need to supply both the extension XML file and a patch against the program itself. It would be very easy to get patch conflicts between many such extensions, as these would all modify more or less the same area of the program.
- Hardcoding data is not very good design and makes the whole idea of the XML extension files pointless: To separate data from the program.

Scripting Language

Another idea would be to use some kind of scripting language (existing or hand crafted for this purpose), create bindings for that language and embed the functionality within the extension XML file. The advantages for this approach:

- Very portable. All one needs to do as an author of a new extension is to create the XML file describing the extensions and implement them in the scripting language. One file, human readable and machine independent.

[9] by using a packet sniffer such as `tcpdump`

- Qt already features a scripting language called QSA[15] which is sufficiently powerful to deal with any of the existing extensions and possibly with any new ones. QSA can bind directly to any objects we choose to export to the scripter, and thus all internal classes can be used to make implementing those extensions easier.

As usual there is a flip side to the coin:

- Especially in case of a custom language (and to some extent using QSA) its power and flexibility is a major factor: if an extension option cannot be expressed in the language, the author must modify the language. In case of a custom language this is still relatively easy, in case of QSA one must either get the patch accepted by Trolltech or use a custom version of QSA.
- In case of QSA we would require the user to install an additional library, as QSA is not part of Qt itself.
- Since such a language would be interpreted, the speed impact might become noticeable with exceptionally large firewalls or when simulating on very slow computers.

UML

A very interesting approach would be to use User Mode Linux (UML) [26] to implement the simulation. User Mode Linux is what the name implies - a linux kernel running in userspace. The approach is conceptually simple: the firewall is constructed in a virtual environment, a minimal linux distribution running in user mode. The only difference is that the netfilter firewall in UML will be 'peppered' with logging instructions, indicating if a rule was matching or not. To illustrate this we examine a simple ruleset (in pseudo instructions for the sake of readability) such as this:

```
if (protocol == ICMP) then DROP packet
if (protocol == TCP) AND (port == 31337) then DROP packet
```

Would be transformed into this sequence:

```
LOG with log-prefix: simul0000
if (protocol == ICMP) then DROP packet
LOG with log-prefix: simul0001
if (protocol == TCP) AND (port == 31337) then DROP packet
LOG with log-prefix: simul0002
```

The log-prefix extension option to the LOG extension is used to identify which log instruction was triggered. For instance, if the last log entry begins with simul0001, the packet got dropped by the second rule. To inject a packet into the system, we simply use the virtual networking facilities UML provides. To modify the firewall, the program logs on by using a virtual console provided by UML and executes iptables in the virtual environment. To identify matching rules, it suffices to simply parse the system log file or the `dmesg`[10] output directly. The entire method is essentially a glorified *popen* (see UNIX manpages, section 3 for more details). This approach, obscure as it may seem, has some strong advantages:

- The extension XML file does not needs to contain any more information we already have, as this information is enough to call `iptables`. It does not need to know anything about the inner workings of extensions, saving the author of new extensions a lot of work. The extensions do not have to be reimplemented in user-space.
- The extension XML file describing the functionality (or any plugin or hardcoded code) could contain bugs, and thus not accurately reflect the behavior of the firewall. This problem is avoided entirely with this approach, as we only observe what actually happens, rather than trying to emulate what happens.
- Arguably this approach could save a lot of time, since all standard extensions do not need to be implemented in the initial release.
- This framework could also be used to check if any given netfilter rule construct is valid or not.

The disadvantages of this approach are equally strong however:

[10] This program prints out the kernel ring buffer which is used for logging.

- A suitable linux distribution for the UML environment must be created. Since the Linux kernel is highly customizable, not all kernels might run the distribution provided. Compiling a micro distribution along with the program would solve the problem with incompatible host and guest kernels[11] but on the other hand take far too much time to compile[12].
- The resulting system would not be very portable even across UNIX platforms: UML is only available for x86 and a few related platforms and currently need a Linux system itself to run. Thus it would neither run on Solaris x86 or on Linux on a SPARC.
- The UML environment needs to be 'booted' during application startup. On slow machines this could contribute very significantly to program startup times.
- It is only possible to determine if an entire rule matches or not. Which specific condition caused the match could not be determined.

It is especially the portability and compatibility problems which caused a different approach to be selected.

3.5.3 Final solution

The method selected to simulate extensions and their options is an ELF shared object approach. The XML file describing an extension and its options cites the filename of an ELF shared object which can be loaded at runtime. When parsing the XML file, the program will load the shared object (if one is given, it is possible to quickly hack up an XML file describing an extension for debugging purposes without telling the program how to simulate them).

To associate the extensions and their options with the functions inside the shared object, an initialisation function named *init_extensionname* is called. For example the function called to initialise the tcp extension would be *init_tcp*. We cannot simply call functions called *match_optionname*, since some extension options are named identically for different extensions and some extension options have more than one alias. (The –dport and –sport options are examples of both.) This initialisation function will connect extensions and their options with functions provided by the shared object. All these functions are of the form:

```
bool foo_bar(IPPacket *p, ExtensionOptionUse *eou);
```

Thus, all the initialisation function has to do is to search for options inside an extension by name and set up a function pointer for each option. To determine if a specific option of a rule matches, the appropriate function pointer is called with two arguments: the packet we wish to simulate and the values of the extension's options (which are encapsulated in the **ExtensionOptionUse** class). This approach combines some of the advantages of the hardcoded functions approach and the scripting language approach:

- Execution time will be just as fast as hardcoded functions.
- Since these shared objects are written in C++, one can express any conceivable extension.
- Although two files (the shared object and the XML file) now describe an extension, this is still better than hardcoding anything.
- It is possible to use a precompiled shared object along with an XML file somebody else wrote (as long as no binary incompatible change has been made to the API headers).
- It saves time during implementation, as a separate scripting language does not have to be researched or implemented.

As with all approaches to this, there are a few disadvantages:

- Shared objects are not portable in their binary form. In practice one would have to supply a XML file and a C++ file to support an extension.
- There will be slight performance impact when starting up the program because the shared objects need to be loaded and the function pointers need to be set up.

[11] One such incompatibility would arise if the host kernel would be configured with a 2G/2G address space split and the guest kernel with a 3G/1G address space split.

[12] The author runs a self compiled Linux system which took more than a few hours to configure and compile for a system with the functionality equivalent to the one we would require.

- Two files are potentially harder to manage than one.

Even so, the three advantages speed, simplicity of implementation (as in implementing the framework for this approach) and simplicity of use (as in implementing extension options) caused this approach to be chosen over the others.

3.5.4 Implementation notes

Implementing the simulation logic

This proved to be fairly easy once the choice was made to use a plugin based system. To begin the six IP specific core matches were hard-coded into the system (as they never change and are always present) to get a basic simulation going. Member variables were added to the classes **Rule** and **Chain** to represent the various states a rule (or chain) could be in when simulating a packet. For instance a rule could not be consulted at all, a rule could not match the packet, a rule matched but did not decide on the fate of the packet and so on.

Once a packet is selected from the simulation dialog and the appropriate route is chosen, a new **Simulation** object gets created which encapsulates the simulation context (i.e. route and the packet). The simulation is then run (by calling a member function), which in turn updates all of the (simulation specific) visualisation variables in the rules and chains. Finally, the visualisation widget is redrawn to show the path of the packet.

One nice thing, which was very easy to implement because of this approach, is that the user can still manipulate the firewall while running a simulation and the changes in the packets path (if any) are immediately visible.

The function to check if a rule matches or not is rather short as it just performs the six core matches and then just loops over used extensions calling any simulation functions (if present). During the implementation of the simulation, a few new attributes were added to the XML extension file format specifying whether a target will decide on the fate of a packet (such as the REJECT extension) or do something and then move on (such as the LOG extension).

Implementing the plugin system

Although it was perceived to be simple at first, the author had no first hand experience with using *dlopen* and related functions. The first step was to write a proof of concept function. The choice was made to implement the udp match extension first, after that tcp and icmp to provide the most common functionality. Writing the file `ext_udp.cpp` was easy enough along with integrating it into the build system to produce an `ext_udp.so` file in the proper place.

Next, a new attribute had to be added to the extension XML file to specify the filename of the plugin to load. Since adding new attributes for the parser to understand had been done often enough, doing so was trivial.

Once all of the extension has been loaded a function is called to open the plugin file and connect the functions to the extension and its options. Using dlopen was easier than expected and it worked immediately. Since the plugin did not feature anything except an empty init function, it was attempted to map this function to a function pointer in order to call it. However at this point it was noticed that the functions are not stored under the same name in the shared object as they appear in the source. This is to be able to represent both functions foo(void) and foo(int): the symbol name takes into account the signature of a function.

The symbol naming scheme was found to be *_Z3foov* for a function declared as *void foo()*. However this signature is not the same as for the real init function, it was just used as first test. The actual init function of the udp extension *void init_udp(Extension *)* has a corresponding symbol name of *_Z9init_udpP9Extension*. Unfortunately the manpages of dlopen were not explaining this and the `objdump` program and some guesswork had to be used. Some reading of the KDE and GNU `libtool` source code (libltdl in particular) suggested that the naming scheme is specific to the C library or compiler used. However, it seemed to work in the universities Solaris environment as well as on the authors Linux machine.

Once the symbol for the init function has been retrieved, the init function is called by the main program, which in turn assigns the function pointers of the extension and its options to functions within the plugin to implement the functionality of the extension. However, once the first member function was used inside the plugin (getOptionByName in particular) the dlopen call failed due to unresolved symbols on the side of the plugin. After consulting the manuals of gcc and ld it was found

that the main program didn't export any of its symbols at runtime and thus the linking parameter `--export-dynamic` was used to resolve this problem. On Solaris this flag is not necessary.

Once these problems were solved, the rest was relatively easy. Connecting the function pointers to the proper functions proved to be rather simple after all. (`eo->simulation = &udp_source_port;` in the case of the udp source-port extension option.) The C++ file then contains a function named *udp_source_port* to implement the functionality of this particular extension option.

As a proof of concept, the extensions udp, tcp and icmp were implemented as three separate plugins. The rest shall be implemented later due to time constraits for the project. (There are a lot of extensions.)

3.6 Visualisation

This is the main research aspect of the project. Since there exist no tools to the authors knowledge at the time of writing which solve the problem in the way we want to, this is largely new work inspired by the shortcomings of previous and current solutions. This section describes the process of finding an efficient way to represent a firewall, highlighting relations between rules and chains.

3.6.1 Ideas and concepts

Tables

Tables, the representation used by most pure front ends to the `iptables` program, are exceptionally easy to implement (given a good toolkit) and indeed Qt provides a class called **QTable** for these purposes. However, this method will not be used, since it solves none of the problems we try to solve: packet simulation in particular has no obvious solution. Flow of control in the firewall, relations of rules to chains and other rules will not be obvious at all.

Graphical Abstraction

A purely graphical idea which was briefly considered was to abstract the firewall to a point where an unbroken line would represent a reject-all firewall, and gaps would represent open (or forwarded) ports. Connections between IP addresses and forwarded ports could be visualized by connecting an icon of a computer (which would represent an IP address) with the gaps.

However, this approach was quickly abandoned as the more and more documentation and specifications about netfilter and iptables was read (and thus discovering the power and flexibility of netfilter). This method has the same limitations wizards have, in particular not being able to construct or represent any possible netfilter firewall.

The Box Method

This method uses a big empty area, called a canvas, and boxes (drawn upon the canvas) to represent a firewall. Such a box represents a single rule and all its properties. Figure 3.3 shows an example of such a rule. The six core matches[13] are shown in the top section of the box (omitting the non-specified ones for the sake of readability). The extensions used (if any) are shown in the bottom section in an ordered fashion: first a header naming the extension used (state in this example) and below the options used. This information can be translated directly to an `iptables` command, in this case `--match state`

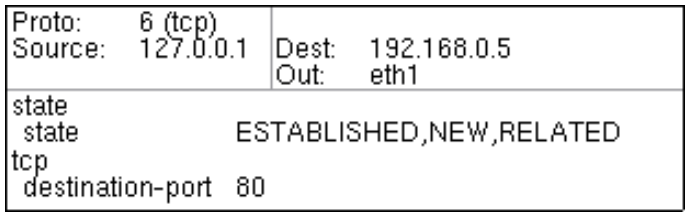

Figure 3.3: A rule represented by a box

`--state ESTABLISHED,NEW,RELATED` for the state section of the example. A number of such rules can then be arranged on the canvas to represent a chain. Additionally, a target can be shown either inside the box or on the outside (perhaps connected with an arrow). This box-method will form the basis of the visualisation used in this project, due to these useful properties of such a representation:

- Fairly easy to implement, only basic line drawing and type-setting functionality required.
- Object orientated, as we have one box per rule. This further simplifies the implementation as we only need to be concerned with this one rule when the box is drawn.
- Easy to arrange these boxes on the canvas in any way desired.
- They can completely represent all properties of a netfilter rule. (Given a method to show the target.)

[13] Source IP, Destination IP, Input Interface, Output Interface, IP Protocol and Fragmentation

3.6.2 Visualizing Tables, Chains and Rules

Representing Tables

There are a few ways to represent tables. One such method involves using the entire canvas and partition it up into sections representing tables as shown in figure 3.4.

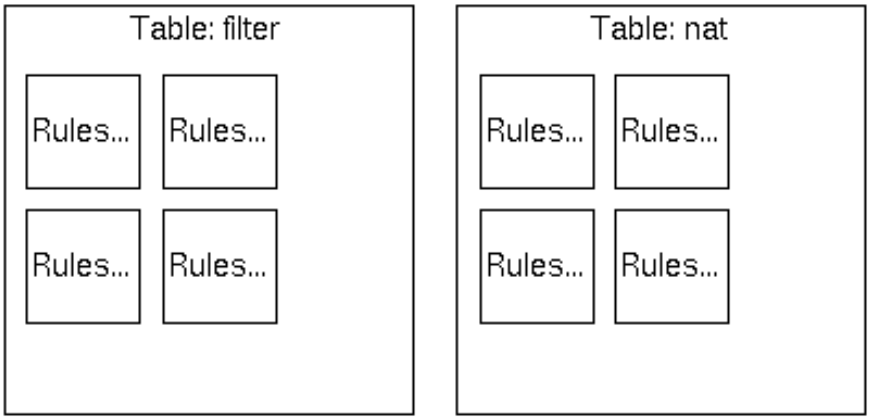

Figure 3.4: Inline Tables

This initial idea was abandoned quickly however, because of the following key disadvantages of this approach:

- Tables can get fairly big, which would lead to excessive scrolling around (which in turn makes it harder to locate specific chains or rules and this has a impact on usability).
- It is more complex to draw and layout than other methods.

A different approach uses one canvas per table. The user can then switch between the separate tables using some kind of control. Methods considered to change the current table were:

- A menu called *Table* which features an entry for each table present in the firewall. This approach is comparable to the *Window* (or *Document*) menu found in many applications to switch between the currently shown document. This approach was not chosen because the option is not immediately visible to the user and it would take two clicks (or two key combinations) to switch between tables.
- A combobox located above the visualisation widget which features one entry for each table. This would have the advantage of being immediately available to the user. Due to the nature of a combobox it would still take two clicks to change the table. Furthermore, mapping obvious shortcut keys to comboboxes is very hard. (Not from the implementation perspective, but from the usability perspective)
- A tab bar located above the visualisation widget (in the same spot the combobox would have been). Since all options are always visible to the user, it will only take one click (or one key press) to change the current table. Figure 3.5 illustrates this approach.

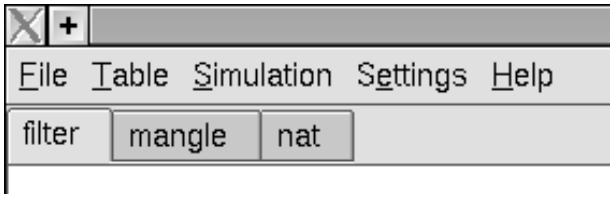

Figure 3.5: A **QTabBar** to change the current table

After implementing all of them, the tab widget seemed to be the most useful and practical one, especially since one can immediately see which tables are available and change tables with just one click (or key press): it is possible to change the active table with the left and right cursor keys.

Representing Chains

Using the box method, there are two possible ways to arrange them to chains: In a horizontal lineup or a vertical one. Initially the horizontal lineup was considered, for the following reason: english is read left-to-right so it felt natural to arrange the rules in such a way. Flow of control would go from left to right as shown in figure 3.6.

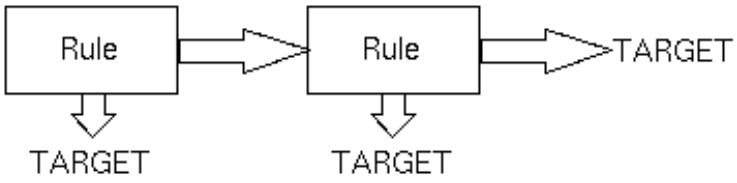

Figure 3.6: Chain rules arranged left to right

A more practical approach however is to arrange the rules from top to bottom for the following reasons:

- Some firewall chains tend to be very long, for instance firewalls which forward a lot of ports to different computers have very long chains in the nat table. In case not every rule fits onto the space the application window can provide, the user must scroll around to view the part of the firewall he / she is interested in. Since modern mice usually feature a mouse wheel, the most likely form of scrolling can be taken care of this way.
- Not all languages are read left to right. Arabic for instance is right to left and japanese from right to left (and top to bottom). A general top to bottom approach seems more sensible knowing this. A proper arabic localization of the program would also involve mirroring the direction of the rules in the horizontal approach. This would also result in more (unnecessary) implementation work. To demonstrate how confusing a left to right chain might appear to a user accustomed to right to left reading figure 3.7 should be consulted. It shows a web browser with its entire layout reversed.

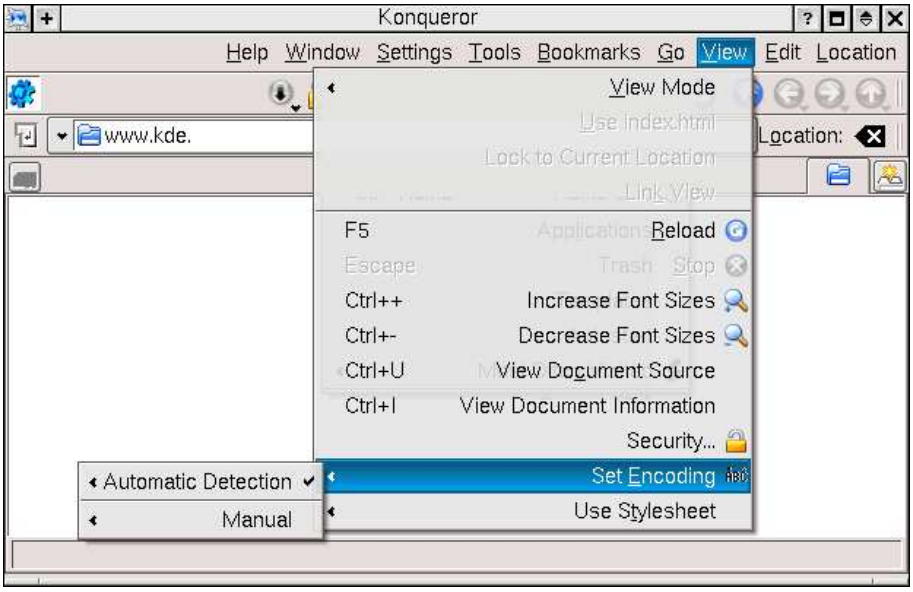

Figure 3.7: Reversed layouts can be confusing

Representing Rules and Targets

As mentioned earlier, rules will be represented with boxes connected by arrows. Figure 3.8 shows how a few rules in a fictional (useless) firewall are arranged. The three chains, INPUT, FORWARD and OUTPUT are arranged next to each other. Their policy is shown directly under their name and in case the chain has at least one rule the policy is also shown at the end of the chain. The target of each

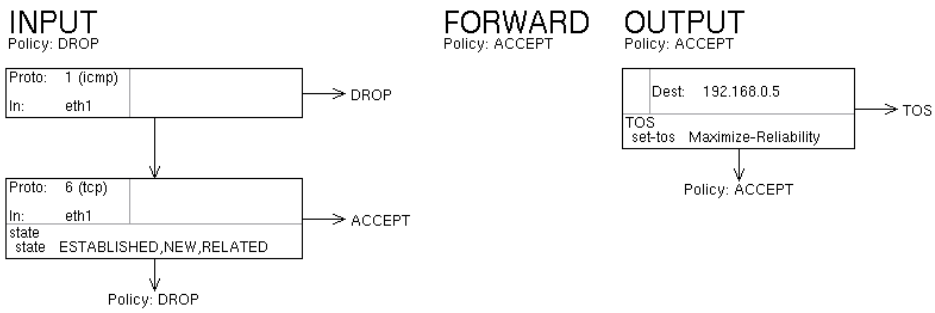

Figure 3.8: Complete representation of a table using top to bottom chains

rule is shown to the right of each rule, which still assumes left to right reading, but should not be as bad as having the entire chain go in the 'wrong' way.

3.6.3 Visualising Simulated Packets

The only remaining task is to visualise the path a test packet takes through a firewall. Figure 3.9 illustrates the basic idea by considering a simple example: how an udp packet, arriving on interface eth0, would be represented by the program passing through a very simple (and again useless) firewall.

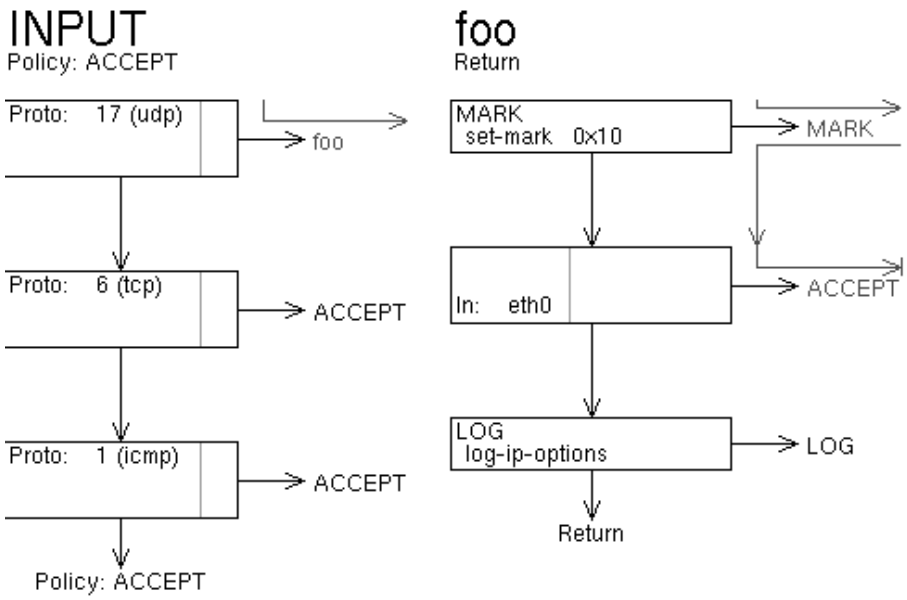

Figure 3.9: Simulating an udp packet from eth0

The packet gets matched by the first simple rule in INPUT which has a target of foo. The arrow demonstrates the packet being handed over to the user defined chain 'foo'. In chain foo, the first rule matches unconditionally and the packet is marked. As this rule only modifies the packet and does not decide upon its final fate, rule number two is considered. As the packet does indeed arrive on eth0, it is accepted and no further rules are considered. Optionally the program could highlight, in non-matching rules, the conditions which caused the rule to not match.

3.6.4 Implementation notes

The framework

Initially, the plan was to create a widget subclassed from **QWidget** and reimplement the drawing methods to display the firewall. However, it was soon discovered, that this could lead to a lot of extra work as scrolling and mouse input event had to be handled as well. Since there is already plenty of code like that inside Qt, other approaches were considered. In particular, the only other feasible option was

to use the **QCanvas** classes. Since this class is used primarily by applications handling document-like data[14] it seemed appropriate to use.

Then a visualisation view widget, called **VisView**, was created (again subclassed from **QWidget**, but for different reasons this time), serving as a kind of container widget for other widgets. The purpose of the **VisView** class is to handle the entire visualisation of a firewall which it has a pointer to. The widget itself arranges two child widgets in its body: a **QTabBar** and a **VisViewCanvas**. The tab bar widget features one tab for each table present in the firewall. The currently displayed firewall table can be changed using these tabs. The custom **VisViewCanvas** class is a subclass of **QCanvasView**. It fulfills two purposes: rendering the current firewall table and dealing with user interaction (such as dragging rules around or editing them).

Initially a **QCanvas** (which is rendered by a **QCanvasView**) is empty. It is then populated with **QCanvasItem** classes, which in turn implement some drawing functions to actually produce a visible image. The visualisation approach uses three custom canvas items which are all shown in figure 3.10. (Bounding boxes are shown around all the items in this figure to demonstrate the space of the canvas they paint upon.)

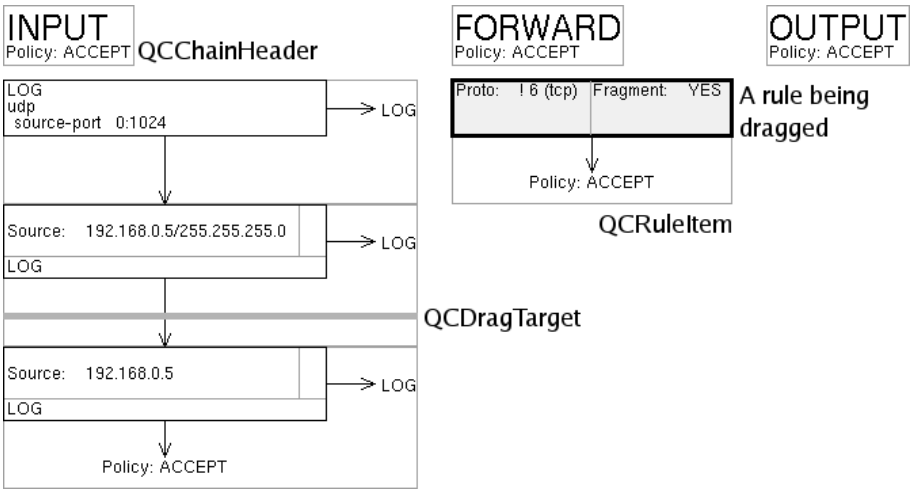

Figure 3.10: The three custom canvas items

The **QCRuleItem** class is the most important one of the three: it fully represents a single rule of the firewall. Drawing the rule is done in just one step in a single function, which produces different results depending on member variables or functions of the **Rule** and **QCRuleItem** classes. For instance, if it is the last rule in a chain, the arrow pointing to the next rule is not drawn as usual and the chain policy is shown.

To speed up drawing (and to make all rules in a chain equal size, since this is much nicer to look at then rules with varying widths) a certain amount of preprocessing is done whenever the state of a rule, chain or table changes. This includes precalculating the length (in pixel) of all the strings drawn (to determine the maximum width in a chain), so that all other rules can adjust. Also the positioning of the rules and chains is dealt with in this preprocessing step to further simplify the drawing function. To do so, the program calls the member function *reorderChains* of the current firewall table after updating any rule or chain. This function recursively calls the appropriate member functions in the rules and chains of the table to perform this preprocessing.

The auxiliary item **QCDragTarget** is just a simple line to represent the new position of a chain if the user would drop the currently dragged rule at the current cursor position. It is only used once and was very simple to implement.

Finally, the canvas item **QCChainHeader** just displays the name of the chain in a big font. Below its name the chain policy is shown.

Rules

The most time-consuming part in implementing the drawing function was finding a sensible way to present the rules in a uniform way, both easy to look at and still conserving valuable screen space. In

[14] For example the KWord word processor uses the **QCanvas** classes to render its documents.

the end all rules in a chain will have the same width, and the middle[15] should be the same for all rules as well. Initially, the light gray separators were not drawn, but the rules seemed to appear clearer and more structured with them, especially if some of the core matches were not specified.

This was the first time the author had to do low level drawing of widgets, but Qt uses a very useful and easy to use class, called **QPainter** to do all of the drawing. This made the drawing of the boxes rather simple.

Since a proportional font is used to render the properties of the rules, it is essential to calculate the pixel width of a given string. Fortunately, Qt provides a class called **QFontMetrics**, which can be used to calculate this length without having to resort to native X calls (which in turn would have an impact upon the portability of the system).

Chains

This was the easiest part of the visualisation problem: all that is needed is to draw two text strings on the canvas.

Simulation

The visualisation aspect was the easiest part of the simulation problem. Since code to draw the arrows to connect the boxes already existed, it was reused. Thus the code was moved into the auxiliary function collection, which also cleaned up the original drawing code a lot.

The rest was simply drawing the right arrows based upon the visualisation variables set up by the simulation object.

[15] The middle of a rule is between the left column of standard matches (containing the protocol, source and in conditions) and the right column of standard matches (containing the fragment, dest and out conditions).

3.7 Editing

Interacting with the visualisation should be possible in various ways: changing the position of rules in the firewall, editing the details of any rule, changing the name of a user defined chain, deleting any rule or user defined chain, adjusting policy of a standard chain and to add new chains and rules to a table. This section explains how these editing functions are provided to the user.

3.7.1 Changing the position of rules

The obvious answer to this problem is to implement drag and drop on the **VisView** class, in order to allow the user to pick up any rule (represented by a **QCRuleItem** and drop it into a new position.

There are quite a few variations on how to visualise drag and drop. For instance, most file managers show a small icon next to the cursor to symbolise what is being dragged; it is rather obvious that the files are dropped at the cursor position. Other applications, such as vector drawing applications, actually drag the entire object or a ghosted / outlined shape, which clearly shows where the object would be when dropped. All these approaches were found not to be particularly useful for this application, and thus a custom drag and drop model was developed.

When the user starts a drag action, the dragged rule is clearly marked with a bold outline and its backdrop changes from white (empty) to a light blue (this could be any other color as well). Additionally, a clearly visible line in the same color appears (which is represented by a **QCDragTarget**) to indicate the position the rule would be in if dropped at the current cursor position. Figure 3.10 (on page 28) shows such a drag action.

3.7.2 Editing chains

To edit chains, it seemed sensible to connect the editing functionality to their representation on the canvas. The usual way to do this is to use a rmb[16] popup menu. In fact, the entire canvas should use such a context sensitive menu to allow for editing of the object underneath the cursor.

For the rest of this document 'sec-clicking' will mean using the secondary mouse button instead of the primary one. Thus for a right handed mouse this means 'clicking' refers to the left button as usual and 'sec-clicking' refers to the right button.

By sec-clicking on a chain header a popup menu will appear, giving options concerning the chain: edit, delete and adding a new rule.

A rule can either be built in (making it possible to change its policy) or user defined (making it possible to change its name), thus the editing dialog for a chains properties should account for that. Similarly, it is impossible to delete a built-in chain. Adding a new rule to a chain is very similar to editing the properties of a rule, and will be discussed in the next section.

The first approach to implement this was to put the entire menus for chains, rules, and tables in the actual menu bar of the main application window. For tables it works fine, there can only be up to three different tables. However, the chain OUTPUT exists three times and would have to be prefixed by its table to be uniquely identifiable. It gets even worse for rules: They do not have an obvious name and there can also be a lot of them, making the menus far too obscure and too big to be used in a sensible way. This is why the popup menu approach was chosen and implemented in the end.

3.7.3 Editing rules

Sec-clicking on any rule in the firewall will present the user with the option to modify or delete it. Deleting a rule is simple, editing is a bit more complicated.

A rule has three distinct properties: a target, conditions and a position in a chain. Changing its position via a dialog would be awkward at best, and is already perfectly taken care of by drag and drop. Editing the target and conditions should be possible via a dialog, but the appearance of such a dialog and the presentation to the user with the vast number of extensions and their options is not obvious at all.

Two different approaches for editing a rule were considered:

[16] right mouse button, in practice however this is secondary mouse button. Thus for a mouse with a left handed configuration, this would refer to the left button.

Hierarchical approach

The hierarchical (or task based) approach is similar to a wizard: the user is first presented with a dialog to set up the most basic options, and on further pages of the dialog additional options are available. For instance, if the user selects the IP protocol to match tcp at the beginning, it is then possible to choose the tcp match extension on a later page, but the user will never see the udp or icmp extensions. The possible advantages of this approach are:

- Fewer options and thus a clearer, less cluttered interface
- By making it impossible to select meaningless extensions (such as icmp if proto = tcp), the chance for creating 'impossible' rules is reduced.
- This approach could be easier to use for a novice user

However, the reasons why this approach was not chosen are:

- By structuring the property dialog into many pages, it can potentially take quite a while to reach the option the user wants to modify
- The novice user is of no prime concern for this application, it is assumed the user already has knowledge about `iptables` and impossible or silly rules.

Flat dialog approach

This approach uses a single dialog to configure every possible aspect of a rule. The disadvantages of this approach are:

- The sheer number of options might confuse the novice user. The interface could also appear cluttered if not well designed.
- It is far easier to construct silly rules, as all options are presented at the same time.

However, this approach was chosen for the following reasons:

- Since the editing method for chains is also 'flat', having a similar method to edit rules would be consistent
- Having all options visible at all times, the advanced user will take less time to locate the option he / she wants to modify

The remaining problem is finding a clean way to represent the many extensions. After a bit of consideration, it was found to be helpful for presentation to make a distinction between the core options and the extensions. Thus the dialog is split in two sections:

- a couple of combo boxes or line edits to modify the seven core options protocol, fragment, source, destination, input interface, output interface and target
- some interface dealing with extensions and their options

One of the first problems was that extensions are not static. At any time a user can add an extension by creating a new XML file. To be able to use this extension, the user should absolutely not be required to edit the `.ui` file of the dialog in Qt Designer to let the system know about that extension. It should "just work", or in other words, a dynamic user interface is needed, that adopts to the extensions present in the system. Three different methods to represent the large number of extensions in a sane way were considered.

Scrollable area This approach attempts to present all extensions lined up in a long list. Since this list will never fit onto the screen as a whole, the user can scroll it using a scroll bar or the mouse wheel.

Figure 3.11 shows how this approach might look like, showing the options to the tcp match extension and half of the udp match extension's options. To view the rest of the udp match extension, the user can either resize the entire dialog window (to allocate more space for the extensions section) or use the scroll bar to position the viewport so that the entire udp options are visible.
Even though this might seem like a good idea, there are several severe disadvantages to this approach:

- Finding an extension can take some time as the list gets longer and the user has to scroll further distances.

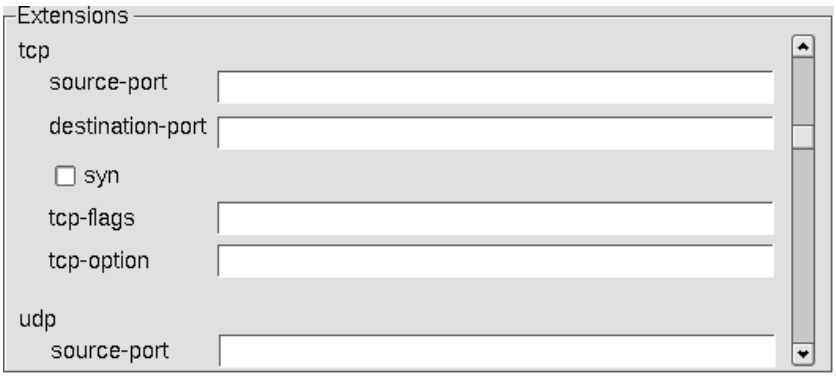

Figure 3.11: The 'flat-file' approach

- This option becomes especially hard to use for users without a mouse wheel.
- This method takes up quite a significant amount of space.
- It is harder to construct the ui in a font-independent way. We would be forced to arrange all widgets by absolute positions, which requires us to take the users font settings (otherwise overlapping widgets would be the result) into account. The reason why absolute positioning is needed, as opposed to the usual layouting approach, is that in Qt an enclosing element (the **QButtonGroup** in figure 3.11) will allways try to be big enough to show all child elements at once. (This only applies if a layout is used.)
- The user would have to scroll through the entire list to see which extensions are used.

Tabbed widget This approach uses a **QTabWidget** to split up all extension options into small and easy to use sections. Since it is easy to add tabs at run time, and also to populate the pages of the **QTabWidget** using layouts, this approach will definitely look more refined as well as being easier to implement as the 'flat-file' approach.

Figure 3.12 shows an example of this approach in which the udp match extension is selected. The page of the tab widget shows the extension's options in the same style we have seen for the flat file approach.

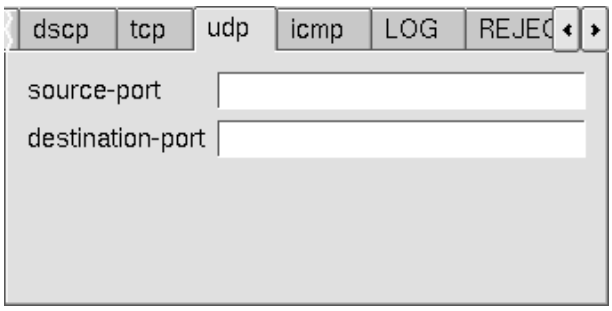

Figure 3.12: The 'tabs' approach

The major disadvantages of this approach are:

- The most obvious one an already be seen in figure 3.12: Given a lot of extensions, it becomes even harder to find the desired one in a **QTabWidget**, as the user would have to click many times on the scroll buttons. It is generally regarded as a bad idea to have tabbed dialogs with more than a few pages.
- It would, for the same reasons, take even longer to find out which extensions are used.

The property editor approach As the solution to this problem was by no means obvious, other applications were examined to research how this problem was solved. A particular one was used all along during this project and seemed to have a solution: Qt Designer. Widgets have a large number of properties which need to be presented and edited in a sensible way.

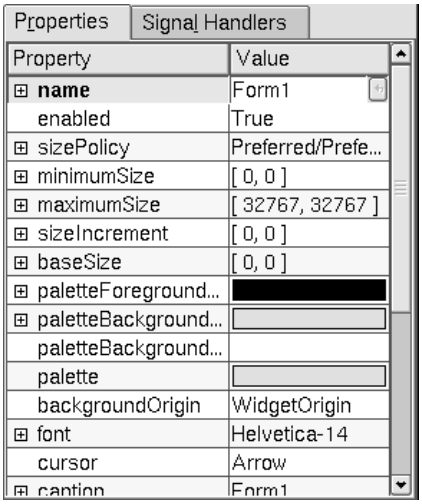

Figure 3.13: Qt Designer's property editor

Figure 3.13 shows an example of this property editor. It is basically a subclassed **QListView** which allows easy editing of the properties: if the user clicks on a property to edit, for example *backgroundOrigin*, the text (at the moment 'WidgetOrigin') changes into a **QComboBox** listing all possible values, as *backgroundOrigin* is of type enum.

Other properties have different mini-editors, for instance the *font* property will present the user with a font chooser dialog, or the *name* property editor will show a simple **QLineEdit**.

However, our problem is slightly different, as we have a more hierarchical structure: a number of extensions, each of which has a zero or more options. The ideal widget to represent this would be a customized list view which knows about this structure using the Qt Designer approach of presenting a mini-editor for the selected option. This can be achieved by subclassing the **QListView** and the **QListViewItem** classes.

An extension itself can be either used or not. If it is used, the user can specify any number of options, if it is not used, specifying options is not possible. This (obvious) fact can be used to further condense the information. Figure 3.14 demonstrates the concept of the **ExtensionList** widget.

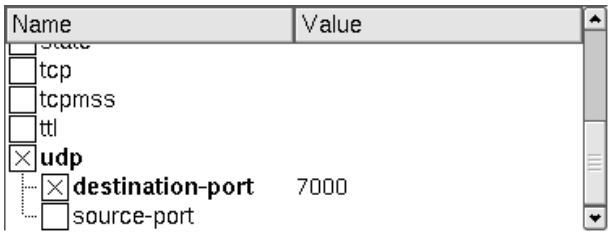

Figure 3.14: The **ExtensionList** class

The boxes to the left of the extensions and their options indicate whether the extension (or one of its options) is used. For instance, in figure 3.14 the tcp extension is not used, but the udp extension is used. Only used extensions show their options. Furthermore, used extensions or options are printed in a bold type face (just like in Qt Designer), as this immediately draws attention to specified, non-default entries.

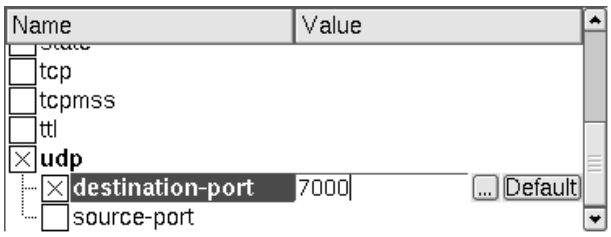

Figure 3.15: The basic mini-editor

Figure 3.15 shows the same situation, but the user has now selected the destination-port option of the udp extension. The user will be presented with three options:

To manually edit the value of the option in a **QLineEdit** widget; this approach is very useful for the advanced user who knows all options by heart. It is also useful if only minor changes are made to the value, such as changing the port from 7000 to 700.

The user can also reset the value to its default value by clicking the 'Default' button. For almost all options this will deactivate the option (since no option is specified per default) and clear its value.

Finally, the user can click the '...' button (called the edit button from now on) to bring up a more advanced editor. This editor depends on the type of the extension option. For instance figure 3.16 shows the "portrange" editor the user would get by clicking the edit button for the udp destination-port option.

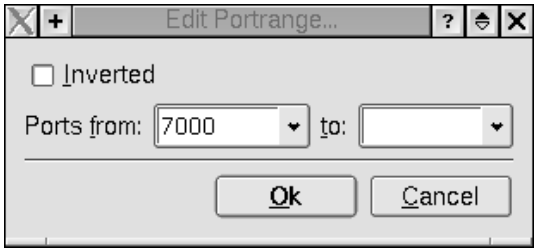

Figure 3.16: The 'portrange' property editor

However, sometimes it is sufficient to provide only the inline mini editor. Figure 3.17 shows the mini editor for the icmp-type extension option, which is a **QComboBox** containing all possible options.

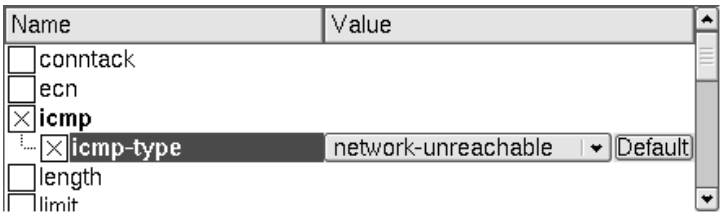

Figure 3.17: The combobox mini-editor

Most extension options do not follow a strict pattern similar to the icmp-type option (which is classified as an *enum* type in the program) or the destination port option (which is of type *portrange*). Since it would consume far too much time to implement separate mini editors for each one of them, the decision was made to introduce a generic type (called *very_ugly* internally) which provides a nice editor with a short help text describing the option.

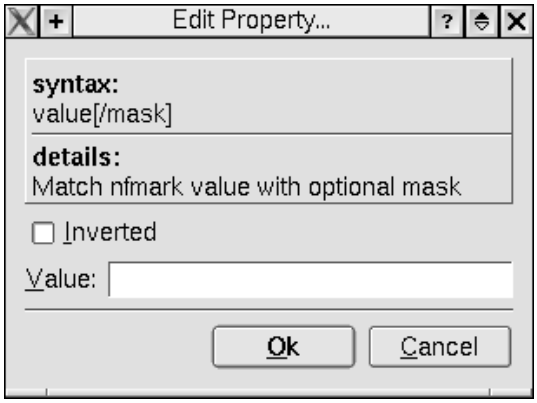

Figure 3.18: The last resort property editor

Figure 3.18 shows an example of this editor. The help text displayed at the top is taken from `iptables -h`, explaining the syntax expected for this option. A simple **QLineEdit** control is then used to specify the exact value. If the option can be inverted, a checkbox do this is also provided.

By reading through all help files of the extensions these types were deemed to be the most useful:

- **NONE**
  The option does not require any arguments. Used for options like syn from the tcm extension.
- **VERY_UGLY**
  Used as a last resort when no other types match. The goal is to have as little as possible extension options of this kind.
- **PORTRANGE**
  A port range which will result in a port[:port] argument.
- **PORTRANGEDASH**
  Similar to the above type, except that the result will look like this: port[-port]
- **NAT_VALUES**
  Used only for the SNAT and DNAT target extensions. It will produce arguments of the general form <ipaddr>[-<ipaddr>][:port-port].
- **ENUMLIST**
  This value can take a combination of predefined values. The prime example is the state option of the state match extension: the possible tokens are INVALID, ESTABLISHED, NEW, RELATED and UNTRACKED. Thus a possible valid argument to the state extension is NEW,ESTABLISHED.
- **ENUM**
  This is used by options like icmp-type of the icmp match extension. It can take one and only one of the predefined values.
- **PROTO**
  This identifies an IP protocol. Examples include icmp, tcp, udp or their numeric values.

As time permits, more types could be added to provide better editors for the various options. The argument type of each extension option is specified in the extension's XML file.

Since the target of a rule is already specified in the core section of the rule property dialog, the **ExtensionList** does not have to display all possible target extensions, only the one that is actually used. Since there are also a lot of target extensions, this further cuts down the space needed to show all extensions.

3.7.4 Implementation notes

Drag & Drop

The drag and drop functionality was implemented using the Qt Event system. This was fairly standard work; the system listens for mouse button press and mouse button release events and modifies its state accordingly.

If the canvas view is in the 'dragging' state, a pointer to the rule being dragged is stored and the **QCDragTarget** item is displayed. Whenever the mouse moves, the position the rule would be in if dropped at the current cursor position is recalculated, and the position (and size) of the **QCDragTarget** is adjusted accordingly. Once the mouse button is released, the chains get modified and the table redrawn. Again, figure 3.10 (on page 28) shows a drag and drop operation in progress.

Special care had to be taken to ensure the pointer is never dangling, otherwise the application would segfault in special corner cases[17]. Qt provides a class **QGuardedPtr** which would have solved this problem much easier, unfortunately this class was only discovered after implementation. Rewriting the code to use this class would have taken too much time, but it is planned in the future. (Also see section 3.8.4 on page 40 regarding this issue.)

The popup menus

The popup menus were very easy to implement, it is just a matter of displaying the correct menu depending on the object the user performed a sec-click. Figure 3.19 shows the table popup menu.

Figure 3.20 shows the chain popup menu. The decision was made to link a more specific popup menu to its parent. For instance the chain popup menu has an entry *Table* which will access the popup menu of the chain's table.

[17] One such corner case is the user, while dragging a rule, pressing CTRL + N to create a new firewall.

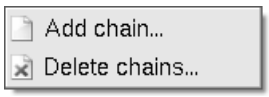

Figure 3.19: The table popup menu

This decision was made to enable the user to always have access to every popup menu (as they are the main form accessing the editing dialogs) without requiring the user to either locate a chain header or find a 'free spot' to sec-click on.

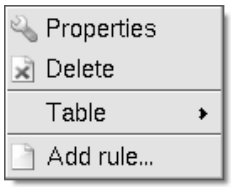

Figure 3.20: The chain popup menu

Figure 3.21 shows the popup menu of a rule. The entry *Chain* is similar to the *Table* entry of a chain's popup menu, giving access to the chain containing this rule.

The *Scroll to target* entry was added (thanks to an idea from my project supervisor) allowing the user to quickly view the target chain of a rule. In case a firewall has many rules, finding the right one can sometimes take some time. This options centres the view on it and highlights the chain header.

Figure 3.21: The rule popup menu

Editing chains

This was by far the easiest part of the editing problem. A simple dialog with two fields, name and policy, with the appropriate one disabled, is all that is needed to adjust chains.

To edit a chain, the user must select the *Properties* entry of the popup menu of a chain. Figure 3.22 shows both versions of the dialog, to the left the property dialog of a builtin chain and to the right the property dialog of a user defined chain. Adding a new chain to the system uses exactly the same dialog. In order to add a new chain to a table, the user selects the *Add chain...* option from the table's popup menu.

Editing rules

The dialog to edit a rule consists of two sections, one to modify the core options of a rule and one to deal with all the extensions.

Figure 3.23 shows an example of this dialog. The already familiar **ExtensionList** widget handles all extensions, whereas a few comboboxes deal with the core properties of a rule.

We observe that the LOG target extension is the only target extension displayed, since the user has selected LOG as the rule's target. If a different target is selected, this target's extension options are available instead of the LOG extension. In other words, only the used target extension is shown.

The most significant work for this dialog (aside from the **ExtensionList** widget) was to implement the correct initialisation for the widgets (in case an existing rule is edited), the rest was standard work.

The ExtensionList widget

The heart and soul of the rule property editor and one of the most complex custom widgets in the project is the custom widget **ExtensionList**. It is subclassed from the **QListView** class with the following key differences:

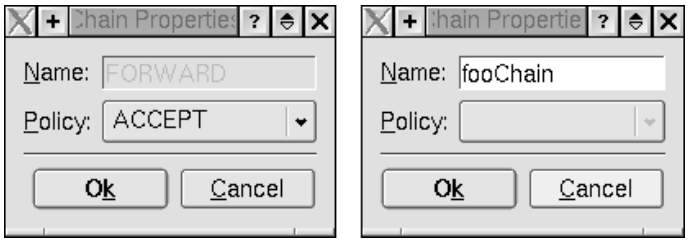

Figure 3.22: Chain Property Dialog

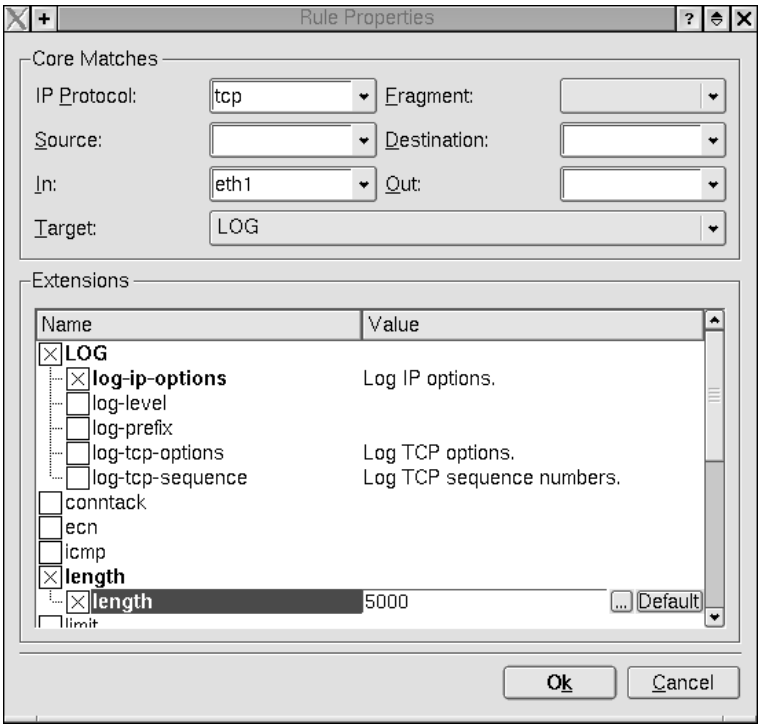

Figure 3.23: Rule Properties Dialog

- When constructed, it automatically queries the **ExtensionRegistry** class and populates itself with all extensions and their options.
- It uses the special classes **ExtensionItem** and **ExtensionOptionItem** instead of the usual **QListViewItem** class to represent items.
- It manages the positioning, showing and hiding of the mini editors and buttons.

Since a **QListViewItem** (and its subclasses) are not a subclass of **QWidget** slightly more work is needed to position **QPushButton** and the mini editor widgets correctly.

Whenever an item is selected, the previously selected item will hide its editing widgets. The newly selected item will, if necessary, create its editing widgets and show them. For positioning we cannot simply make them a child of the **ExtensionOptionItem**, since the latter is not a **QWidget**. However, it is still possible to position a widget by absolute coordinates, thus we simply put it on top of the **ExtensionList** and position it by querying the newly selected item's geometry.

This was not very hard to do at all, since the Qt Designer source code provides an excellent example of how to do so. The actual code is different however since we have more widgets to position.

Property Editors

These editors (which can be accessed by clicking the '...' button) are mostly of very trivial nature, and it was routine work to implement them. The only editor more complex and interesting than the others is the nat rules editor. Since the `--to-source` and `--from-source` options can be specified multiple times, a new convention was introduced: the pipe symbol in an argument value in the program separates two different arguments. The advantage of this approach is that we can easily group together extension options into just one. Figure 3.24 shows the nat rules property editor.

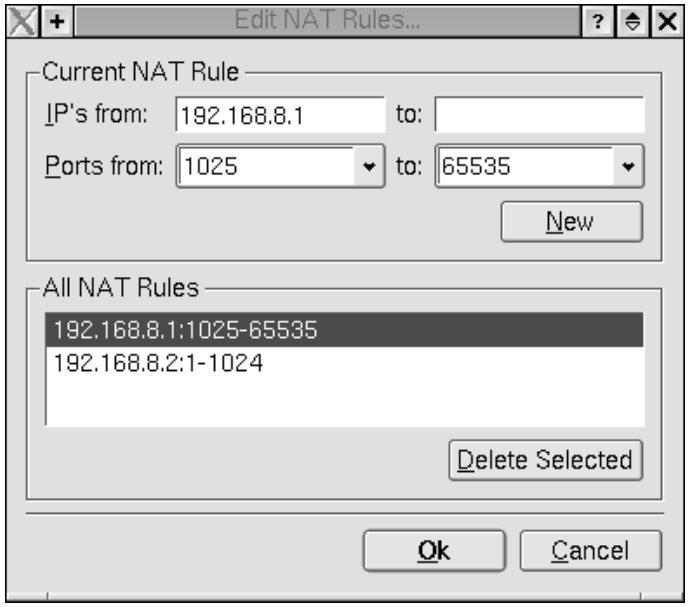

Figure 3.24: The nat rules property editor

This editor constructs the just-one-string representation of several nat rules, and can also be used to edit such a string in a comfortable way. The string constructed for the rules show in figure 3.24 would be:
`192.168.8.1:1025-65535|192.168.8.2:1-1024`

3.8 Limitations and Future Work

This section deals with three things: Ideas that were proposed initially but were not implemented due to time constraints as they were not as important as others, new ideas that came up during the implementation work and limitations of the system and their possible workarounds or solutions.

3.8.1 Make the program installable

Currently the build process is designed to create an executable which is meant to be executed in the build directory itself. This was done in order to make the edit-compile-test cycle as short as possible. Making it possible for the program to be installed into a system will involve at least these steps:

- Remove (or make configurable) all hardcoded relative paths (there are no hardcoded absolute paths, save to files like `/etc/protocols`). Relative paths are used for things like where to find the extension XML files. (at the moment `./extensions/xml/`)
- Possibly convert the project to the GNU autotools, so that one performs the usual `./configure && make && make install` dance to install the application.

3.8.2 A XML firewall to shellscript tool

The default save file format for firewalls is the XML file format. Although an executable shellscript can be exported very easily by the application it could be nice to have a little command line utility that will:

- Take a XML firewall and convert it to an `iptables-save` file which can be used by `iptables-restore`
- Take a XML firewall and convert it to an executable shellscript (for use in the runlevels.)
- Take a XML firewall and upload it into the local kernel using `iptables`

The program (perhaps called `fwtool`) could be used directly in a runlevel script to initialize the firewall from an XML file rather than a shell script.

It should not be too much trouble to quickly create such a tool by reusing much of the code from the main program. (Since Qt does not require us to use a GUI at all, all classes can be used.)

3.8.3 Support IPv6

Currently the tool only supports `iptables`. However it will be no trouble at all to use `iptables6` instead, only the simulation area requires a bit of work. The ideal solution would be to somehow integrate IPv6 into the tool so that the user can construct both firewall with just one program.

There is rudimentary support in the simulation classes to support both versions of IP, however only IPv4 is actually implemented at the time. Classes like **QHostAddress** were used to represent IP addresses rather than **unsigned int** in order to make this effort as painless as possible.

3.8.4 Beautification of code

Although care was taken at all times to produce readable code, sometimes a quick hack or an incomplete solution (allways marked by comments!) were used to quickly get basic functionality in order to implement the "more interesting" bits. Sometimes things were done one way when they could have been accomplished using Qt classes that were discovered afterwards. This section deals with the various things that can make the code more readable, maintainable and stable.

Resolve FIXME, HACK and TODO comments

Whenever incomplete solutions were used instead of a "proper" solution, mostly due to time constraints, these sections in the code were marked with comments. Removing them by taking some time to produce a more correct, more stable, faster or easier to maintain solution will be a good thing to do and increase the overall quality of the code.

QGuardedPtr

Sometimes special care had to be taken that a pointer is never 'dangling', i.e. pointing to an already deleted object. Only after implementing those special measures (especially with the pointer that points to the rule being dragged) I learned about the **QGuardedPtr** class which would have solved all these problems.

Due to time constraints the old (but functional) method was not converted to **QGuardedPtr** uses. It would greatly simplify some code sections and make the implementation of some new features far easier if the project would use **QGuardedPtr** objects.

Use QDomDocument to create and load XML files

The **QDomDocument** class is another class I learned about after it was "too late". The parser for the XML firewalls is especially nasty code which could have been implemented far nicer and quicker if the **QDomDocument** had been used.

Since this class can also be used to generate XML, the save function for XML firewalls would have been much easier and cleaner.

Support internationalization

All strings and messages have been hardcoded. Qt supports internationalization in a very nice way, and it should not be too much trouble to convert to it. Doing so has the obvious advantage of making it possible to provide localized versions of the program (if there exists demand that is).

Use doxygen to generate documentation

By inserting specially formatted comments into class declarations and implementations it is possible to create high quality class documentation in an automated way. An excellent example of this is the KDE documentation[18] which uses doxygen to generate it.

This would possibly make life easier for third party extension developers, as they would use some of the internal classes of the program in their extension code.

3.8.5 Editing improvements

This section deals with possible improvements to the editing capabilities and the general usability of the program.

Wizards

One idea that came up during the project proposal was to add some wizards as a complementary tool to create a new firewall from scratch. They could be integrated into the system and would then provide a convenient way to set up a basic working firewall, which can then be tweaked by the advanced user, or as a method to set up a basic working firewall for the novice user.

It could be also used as an educational tool for novice user, as in: "Oh, that's how you do it!".

Copy, Cut and Paste for rules

As the title says, adding copy, cut and past (also known as ccp) entries to the rule's popup menu may be of some use. Although cut and paste are already implemented by drag and drop, it could be used to move a rule from one table to another.

As an alternative to copy and paste, the drag and drop system could be modified to copy a rule if the CTRL (or some other user-definable key) is held down during the drag action. This functionality would then be similar to the behavior in some popular file managers like KDE's Konqueror.

Copy, Cut and Paste for chains

The ability to quickly duplicate entire chains or to move chains from table to table might be useful for certain scenarios. The menu entries to a chain's popup menu should definitely be added, especially if ccp is available for rules: it would also make the interface more consistent as a whole.

Dragging chains

Implementing drag and drop on chains could also be a nice feature, although its real use is purely cosmetic: it does not matter where a chain is located in a firewall, however the firewall designer might prefer to order them differently at a later time.

Notify the user when creating impossible rules

At the moment a user can create invalid netfilter rules: for instance it is possible to specify SNAT as a target in the filter table. The extension's XML file format could be extended to allow the extension author to specify dependencies or conflicts.

This feature would also greatly benefit novice users who do not have indepth experience with `iptables`.

Create more mini-editors

More mini editors for the rule property editor will also improve the usability of the program. For instance, editors for numeric values (with min / max validation) can be useful for the length or ttl extensions. Editors for the address/netmask argument type could also be useful.

Ideally, these mini editors also show a brief help text, such as the last-resort editor as seen in figure 3.18 on page 34. Explaining the option in such a way would make the system more user friendly for both novice and advanced users alike.

An undo function

Humans (and especially users) make mistakes. Although the system already provides some form of undo in the form of *Cancel* buttons, which discard any changes made to the object being edited, a global undo function could prove useful. Most users certainly would expect an undo function of some kind. Indeed the KDE style guideline[20] and several others also suggests an undo function to make a program 'user-friendly'.

3.8.6 Simulation improvements

This section examines several possible improvements to the packet simulation subsystem of the program.

Also visualise the global system

Currently the path of a packet through a table is visualized through arrows. However what happens on a global scope? It could be beneficial to show the entire netfilter system (this would be static anyway and thus easy to implement) in a separate window, and show the path of a packet though the chains and tables on this global view. Thus we could see a packet entering the POSTROUTING chain, then a routing decision is made and the arrows pass into the appropriate chain. (In this case either FORWARD chain or INPUT.)

This overview would include all standard chains and tables and abstract entities like 'routing decision' or 'local process' and network interfaces. (User defined chains are irrelevant in this case, as they are only 'sub-chains' and never stand on their own.)

Report the fate and state of the packet

This is closely related to the above feature, a brief indication whether the packet gets dropped or accepted would be useful. Currently the user has to follow the packet's path through the firewall and identify the end of the packet's path to figure out the fate of the packet.

This should be not too hard to implement, as the system already knows the fate of a packet. It just doesn't tell the user about it explicitly yet.

Use tables other than filter

At the moment the system only understands the three chains of the filter table. However, for a complete simulation the other tables should also be used, especially the mangle table to demonstrate how specific packets are modified.

The new raw table (as of Linux 2.6.6) could also be supported very easily. It contains two chains, PREROUTING and POSTROUTING, which are simply executed before (and after) any other respectively.

Implementing the rest of the extensions

Currently only three match extensions are implemented: tcp, udp and icmp. No target extensions are implemented. An obvious task in the future is to implement the rest of the standard extensions shipped with the Linux kernel to make the system complete.

Highlighting non-matching conditions in rules

A nice feature could be the ability to identify which condition caused a rule to not match. This could be achieved by color coding the offending condition.

Support awkward extensions

Some extensions are hard to simulate properly. Two examples are the state extension and the limit extension. They are hard to simulate, since they depend on context and the system deals only with one packet at the time. Thus there is no notion of number of packets per second or TCP connection states.

One possible solution to this problem is to introduce an entity where extensions can register the-user-must-decide values. Thus the state extension can register a value called *state*, and the limit extension can register a value named *match-limit*. This happens transparently to the user and changes nothing so far.

Once the user decides to simulate a packet, an additional settings field is shown in the simulation window, where the user can change any of these values. This compares to the meta-information the user has to specify already, such as the route or the input (or output) network interface. To be user friendly, only the values needed to determine the fate of a packet should be displayed. The system then uses this additional information to determine if a condition matches.

Resolve hostnames

A nice feature could be to automatically convert from and to hostnames when needed. Currently the system can compare two IP addresses, but it will be unable to match (or not match) www.google.com with 192.168.0.5 for example.

When importing a firewall from an `iptables-save` file it should also be possible to lookup any IP address and convert it to a human readable name if possible. Possible are additional entries in the popup menu of a rule with entries to convert all IP addresses to names and vice versa or facilities to do so in the rule property editor.

Packet editor

Currently the system can load `tcpdump` dump files in order to obtain packets for simulation. Work has already started on an editor which can be used to hand-craft any possible packet in a user-friendly way. However time constrains forced this work to be postponed and not integrated into the current system.

This packet editor should definitely be completed in a future version of the program, as it would make the testing of firewalls much easier. Packets created using this editor could then also be saved into a `tcpdump` compatible format for later use. (Thus one could build a collection of interesting packets to rapidly test different firewalls.)

Simulated system log

Some extensions create system log entries, such as the LOG or ULOG target extensions. A simple system log could be simulated to show the user what kind of output the rules will create. This could be implemented in the target extension's plugin file.

Support packet mangling

Currently the class used to abstract IPv4 lacks functions to modify entries in the IP header. Implementing these would allow some target extensions to be implemented, such as the TOS extension.

Showing the final packet

To complement the above feature the system should be able to show the final packet as it arrives in userland[18] or leaves the wire, as this packet is potentially different from the starting one due to packet mangling rules.

3.8.7 Importing firewalls

From a remote computer

A nifty feature would be to be the ability to import the firewall from any remote computer the user has valid authentication for. This could be achieved by using `ssh` to execute `iptables-save` on the remote computer as root.

3.8.8 Uploading firewalls

To the local machine

A useful feature especially for the home user would be to load the current firewall into the local running kernel with `iptables`.

To remote machines

An even more useful feature for a network administrator would be the ability to update the netfilter of a remote machine. This could be achieved by using `ssh` to remotely execute `iptables-restore`.

3.8.9 Porting the program to KDE

The KDE desktop environment is built using Qt, however features many improved classes. One example is the **KFileDialog** class which is a subclass of **QFileDialog**. It provides essentially the same base functionality but is more refined and useful in many ways. KDE also provides many invisible structural advantages such as a configuration framework for applications, the KIO framework, and libltdl[19].

For a large part this is using KDE classes instead of the Qt classes, for instance using **KFileDialog** instead of **QFileDialog**. Other sections require a bit more work: the configuration system (discussed in greater detail below) should be implemented using the KDE technology instead of a custom one.

A port to KDE would have many advantages, but most importantly KDE has a rather large userbase and exposing this application as open source to the masses could have a log of interesting effects including, but not limited to: third party patches and improvements, a user base which can report bugs and helping the KDE project.

3.8.10 Implement configuration system

This section deals with how the configuration system, for which initial work exists in the project (such as the virtual network hardware configuration), can be used and improved.

Using system files

The settings dialog was planned to have options for the IP protocols presented in a protocol combobox (such as the one on the rule property dialog), and the IP protocol the system knows about. Currently the program has a few hard-coded defaults, but they should be made configurable.

The mapping from name to number already exists in the standard file `/etc/protocols`. The settings dialog should be able to use this file and allow the user to make custom adjustments as well, for instance inventing a new IP protocol, which could be useful for testing purposes.

Other interesting files are `/etc/services` and `/etc/hosts`.

Make fonts configurable

At the moment the fonts used in the visualisation widget are hardcoded into the system. Fortunately, they are hardcoded in a nice way (as **const int**) and thus making them configurable through the settings dialog would be an easy to implement feature.

[18] Userland is the opposite of kernel space. It is where user processes are run.

[19] libtldl provides a generic interface to various dynamic loaders, and is part of KDE or GNU libtool.

Query available extensions

Sometimes it might be useful to only show a few extensions, such as the most common ones or only the ones available on a specific computer. Which extensions are available on any given kernel can be determined by querying the two pseudo files `/proc/net/ip_tables_matches` for available match extensions and `/proc/net/ip_tables_targets` for possible targets. (As usual this can also be done using `ssh` to obtain this information from remote computers.)

The actual extensions shown should be configurable using the settings dialog, with an *Only Available Extensions* quick-configure option.

Chapter 4

Conclusion

4.1 Evaluation against the requirements

I will now compare the produced program to the initial requirements listed in section 3.1.3 (on page 11), and evaluate how the requirements were met or how a requirement was changed during the implementation phase.

4.1.1 Requirements

Saving and loading a firewall

After the initial data structures for the rules, chains and tables were devised, functionality to load a firewall from an XML firewall file was implemented. Since no editing facilities were present at the time, these XML firewalls were hand crafted initially. Once some editing features were implemented, the ability to create an XML firewall save file from the current firewall was added. Thus, this requirement is fully met. Because it is such a basic and obvious requirement, it never changed during the project.

Import a firewall from some standard format

Since 'standard format' is a bit vague, the first thing to do was to decide what this standard format would be. Also reasons had to be found why we did not use this standard format as the native file format. (See section 3.3.1 on page 15 for more details, in particular item number 3.) Once it was decided that the output of `iptables-save` would be this standard format, a simple parser was implemented to fulfill this requirement.

Export a firewall to a shellscript

This was also one of the most fundamental requirements of the system, after all of what use is a firewall we cannot actually use? Since throughout the program the names of extension options are identical to the command line `iptables` expects (minus the leading `--`), fulfilling this requirement was also fairly straight forward. Although fundamental, the intention of this requirement (to be able to use a firewall created in the program) could have been met in other ways. Section 3.8.2 (on page 39) outlines one such approach.

Create or modify any (Netfilter based) firewall

As it stands, this requirement is not met completely. However, the system has the potential to support any kind of netfilter firewall. All the basic things, such as user defined chains and all core matches, have been implemented. However, some of the more exotic extensions do not yet have a corresponding XML file. This work however does not even require the program to be recompiled.

Since all dialogs dealing with chains and rules are dynamic (i.e. created based on the extensions the system knows about), all possible netfilter based firewalls can be created or modified (i.e. imported and then modified). Given all extensions have a corresponding XML file, then (in terms of logic) the system is correct (i.e. any netfilter firewall can be created and represented by the system), however it is not sound: some firewalls can be created in the system which are not actually valid. (One example is to be able to put a SNAT rule into the filter table.)

This undesirable property requires modifications to the extension XML file format to be resolved.

Deal with the numerous extensions to Netfilter in a sane way

A significant amount of time was invested to address this particular requirement. A sane way was deemed to be the way requiring the least possible work and maintenance. Thus the decision was made to use XML files to contain the details of the various extensions.

Once all existing standard extensions (i.e. extensions which come with a stock Linux kernel) were examined, suitable datastructures were invented. Then a suitable XML format was devised and a parser created. Finally the singular[1] class **ExtensionRegistry** was implemented, which loads all extensions at system startup, It is used by various system components to obtain information about the various extensions and their options.

Must use a graphical representation of the firewall

This is perhaps the single most important requirement, as it is the main point of the project. Needless to say, this requirement is fulfilled, as no conventional (i.e. a table), text based approach is used. The widget to display the firewall is a custom widget, specifically developed for this project.

The tool itself should not require iptables to be installed

This requirement was added to make it possible to compile and run the program on the Bath University Solaris servers. Fulfilling this requirement required no work whatsoever, as there is simply no reason to call `iptables` during the normal operation of program. The only point where a component of the `iptables` suite is required, is the `iptables-save` program to import a Netfilter firewall from the running kernel. Since Solaris does not support Linux Netfilter firewalls, trying to execute `iptables` on a Solaris machine does not make any sense in the first place.

Allow the editing of any given rule in a firewall

This is also a straight forward requirement. Since there are no special, magic rules in a firewall, as long as the required XML files for the extensions used exist, any rule can be edited.

Allow IPv6 in future versions

In order to fulfill this requirement care was taken to ensure that no facts about IP are hard-coded.

In fact, most IPv4 specific code sections are encapsulated in the **IPPacket** class. At the time of writing a corresponding **IPv6Packet** class does not exist yet, this is future work. (See section 3.8.3 on page 39 for more details.) When representing IP addressed, they are either stored as a string or in the IP-version independent Qt class **QHostAddress**.

Simulate a packet going through the firewall

This is the second major requirement of the system, without it the program would just be another iptables front end (even though it has a different approach for displaying the firewall). This was the last feature implemented, and also took a significant amount of time to design and implement.

The requirement itself it met partly. Two restrictions (which can be overcome, see section 3.8.6 on page 41 for more details) exist: Not all extensions are supported and it is impossible to hand craft packets using in-program facilities.

However the proof of concept implementation is functional, and allows visualisation of packets captured of the wire (using `tcpdump`). Once this feature is fully implemented (along with a packet creator), the program will become even more useful.

4.1.2 Conclusion

Overall, the requirements were met well enough. Since any changes to the requirements were minor details, the resulting program does indeed match the initial expectations. Although most of the requirements are common sense, they provided a valuable framework to create a solution in, allowing for plenty of creativity. Since they did not go into great detail, a lot of time was saved in the initial design phase. By delaying the important detailed design decisions to when they were needed, a rapid edit-compile-test cycle was possible. The framework established however was very valuable, as it ensured no 'wrong' or short-sighted decisions were made at any point.

[1] A class with only one instance

4.2 Reflection on things learned

4.2.1 Acquired knowledge

Even though a large part of the project involved writing code, a major portion of the time was spent reading various bits of documentation, howtos, RFCs, source code and examining different implementation approaches to various problems. This has resulted in gaining some valuable knowledge, in particular:

Networking

Many aspects of the project required indepth knowledge of how IPv4 based networking works, in general and on Linux in particular. Specifically, knowing how the lower levels (IP packets and Ethernet frames) work has been very useful for implementing the simulation feature. However, mid level knowledge of IP networking was acquired as well, in particular for the TCP, UDP and ICMP protocols.

Attending the third year "Computer Networking" course at the University was also very helpful, I got the chance to ask some questions relevant to my project.

Netfilter

As the Netfilter/iptables system is a central aspect of the project, it was researched thoroughly. The following knowledge was acquired due to this:

- How different types firewalls are constructed
- How the Netfilter/iptables system works
- How to assess the quality of firewalls, common problems in them and their solutions

XML

Dealing with XML during the project has made me aware of the following things:

- How XML works and how to create sensible file formats
- How to efficiently parse XML files (with Qt)
- How useful XML is, its advantages and disadvantages
- When it is feasible to use XML

Qt

Although I had a little bit of exposure to the Qt toolkit prior to the project, implementing a large program, using as many Qt features as possible, has convinced me that Qt could very well be the most powerful toolkit created so far.

As the implementation of the program progressed, I discovered more and more features and different (usually more efficient) approaches to certain problems, mostly due to the excellent documentation available.

In particular, learning how to use the Qt **QString** class and the container classes (combined with the realization that Qt can also create non-gui programs, i.e. servers) could simplify much of my future work.

UNIX dynamic loader

The plugin system of the program was the first one I have ever written myself. During this the following things were learned:

- How *dlopen* and the related functions work
- How the GNU tools store symbols names
- How to properly compile shared objects

4.3 Things that went very well

Although I think that overall implementing the ideas went rather well in general, two aspects of the implementation phase deserve special mention.

4.3.1 Extensions

Although the implementation is neither particularly clever or was exceedingly easy, the final result works very well. If the program were distributed in binary form, any user can easily make the system support their extensions for editing. Even if simulation support is a bit more work, as it requires to compile a shared object, the ability to create, edit and display a firewall using this extension is very useful.

4.3.2 Simulation

Even though this feature consumed a lot of time and energy, most of it was spent of figuring out how to make this feature as clean and easy to maintain as possible. The implementation phase on the other hand was very short and productive: The core simulation code is object orientated, concise and very simple. The code for the shared objects to support the various extensions is also very brief, and only one C++ file is needed per extension. (No need for header files or meta object code.)

4.4 Things I would have done differently

This section deals with the various aspects of the project which I am not happy about or that simply didn't go as planned.

4.4.1 Time Management

Although the total time spent on the project was pretty much equal to the time estimated, the actual work done was not done when it was supposed to be done. This was caused by two things: In the first half of the project (1st semester) not much work was done and the bulk of the work was done the the second half. By starting to work earlier this would not have happened. Secondly, the time plan as produced in the project proposal was rendered almost useless as some requirements changed and I learned more about the subject matter. This means some tasks included far more work than initially proposed, however some tasks[2] vanished completely.

4.5 Final Conclusion

To conclude, I believe a potentially very valuable firewall tool has been created, which uses some new and unique features that sets it apart from other solutions. Even though the project faced some difficulties, especially in planning, in the end the final result and things learned made it a worthwhile project, and I intend to work on it in the future and implement many of the improvements mentioned in section 3.8.

I have used the tool myself to create a complete NAT firewall for the Linux router I use at home, and also used it to inspect some very big firewalls (such as the ones created by SuSE's YaST tool). The representation used makes it fairly easy to spot some problems in such firewalls, such as dead code or useless rules.

I also believe that writing a KDE port of the tool, and submitting it to the KDE project, could help some people who either wish to create Netfilter firewalls or to simple learn about them using the simulation feature.

[2]Such as the creation of a table-based (non-graphical) representation of a firewall as a first prototype.

Bibliography

[1] Postel, J., (1980) *User Datagram Protocol*, [Internet RFC] 768, [Online] Available at http://www.ietf.org/rfc/rfc0768.txt

[2] Information Sciences Institute, University of Southern California, (1981) *Internet Protocol*, [Internet RFC] 791, [Online] Available at http://www.ietf.org/rfc/rfc0791.txt

[3] Postel, J., (1981) *Internet Control Message Protocol*, [Internet RFC] 792, [Online] Available at http://www.ietf.org/rfc/rfc0792.txt

[4] Information Sciences Institute, University of Southern California, (1981) *Transmission Control Protocol*, [Internet RFC] 793, [Online] Available at http://www.ietf.org/rfc/rfc0793.txt

[5] Rusty Russel (2002) *Linux 2.4 Packet Filtering HOWTO* [Online] Available at http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html (accessed May 10 2004)

[6] Rusty Russel (2002) *Linux 2.4 NAT HOWTO* [Online] Available at http://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.html (accessed May 10 2004)

[7] The Bifrost Team *Bifrost*, A web-based iptables frontend [Online] Available at http://bifrost.heimdalls.com (accessed May 10 2004)

[8] The Bifrost Team *Bifrost Demonstration Site*, A demo for Bifrost [Online] Available at http://bifrost.heimdalls.com/demo.html (accessed May 10 2004)

[9] The Firewall Builder Team *Firewall Builder*, An iptables frontend [Online] Available at http://www.fwbuilder.org (accessed May 10 2004)

[10] SuSE Linux GmbH. *YaST2*, The configuration tool of SuSE Linux [Online] Available at http://www.suse.com (accessed May 10 2004)

[11] Porter, K. (2003) *iptables visualization tool* [Online] Available at http://linux.derkeiler.com/Newsgroups/comp.os.linux.networking/2003-09/1080.html (accessed May 10 2004)

[12] The GTK Developers *The GIMP Toolkit*, A toolkit [Online] Available at http://www.gtk.org (accessed May 10 2004)

[13] The gtkmm Developers *gtkmm*, The C++ interface to GTK+ [Online] Available at http://www.gtkmm.org (accessed May 10 2004)

[14] Trolltech Inc. *Qt*, The Qt Toolkit [Online] Available at http://www.trolltech.com (accessed May 10 2004)

[15] Trolltech Inc. *qsa*, The QSA Scripting Language [Online] Available at http://www.trolltech.com/products/qsa/index.html (accessed May 10 2004)

[16] Trolltech Inc. (2004) *Trolltech Why doesn't Qt use templates for signals and slots?* [Online] Available at http://doc.trolltech.com/3.3/templates.html (accessed May 10 2004)

[17] Trolltech Inc. (2004) *Properties for QObject classes* [Online] Available at http://doc.trolltech.com/3.3/properties.html (accessed May 10 2004)

[18] The KDE Developers (1996-2004) *The KDE API documentation* [Online] Available at http://developer.kde.org/documentation/library/cvs-api/ (accessed May 10 2004)

[19] The wxWidgets Team *wxWidgets*, A toolkit [Online] Available at http://www.wxwidgets.org (accessed May 10 2004)

[20] The KDE User Interface Team (1996-2004) *KDE User Interface Standards* [Online] Available at http://developer.kde.org/documentation/standards/kde/style/basics/index.html and http://developer.kde.org/documentation/design/ui/index.html (accessed May 10 2004)

[21] The Kate Authors *kate*, The KDE Advanced Text Editor [Online] Available at http://kate.kde.org (accessed May 10 2004)

[22] The Konqueror Authors *konqueror*, The web-browser and file manager for KDE [Online] Available at http://konqueror.kde.org (accessed May 10 2004)

[23] The dia Authors *dia*, A vector based diagram drawing program [Online] Available at http://www.lysator.liu.se/ alla/dia/ (accessed May 10 2004)

[24] The Ehereal Authors *Ethereal*, A Network Packet Sniffer and Analyzer [Online] Available at http://www.ethereal.com (accessed May 10 2004)

[25] The tcmpdump Authors *tcpdump*, A popular packet sniffer and packet capturing library [Online] Available at http://www.tcpdump.org/ (accessed May 10 2004)

[26] The UML Developers *User Mode Linux*, A linux kernel port to usermode [Online] Available at http://user-mode-linux.sourceforge.net/ (accessed May 10 2004)

[27] Fabrice Marie (2002) *Netfilter Extensions HOWTO* [Online] Available at http://www.netfilter.org/documentation/HOWTO/netfilter-extensions-HOWTO.html (accessed May 10 2004)

[28] Rusty Russel and Harald Welte (2002) *Linux netfilter Hacking HOWTO* [Online] Available at http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html (accessed May 10 2004)

[29] Rusty Russel (2001) *Linux Networking-concepts HOWTO* [Online] Available at http://www.netfilter.org/documentation/HOWTO/networking-concepts-HOWTO.html (accessed May 10 2004)

Appendix A

The toolkit: Qt

Since there is neither a de-facto nor de-jure standard toolkit under the UNIX/X11 platform, there exist a lot of different toolkits. These toolkits sometimes have conflicting naming schemes, vastly different implementation languages and bindings and philosophies. This section is a brief introduction to the Qt toolkit; explaining some of the terminology used later in this dissertation.

A.1 Basic Concepts

Qt itself is built from C++ classes. Although in the end Qt is standard C++ code (i.e. compilable with any standard compliant compiler) it uses a preprocessor. In particular any class derived from **QObject** should be preprocessed before compiling. The preprocessor (called `moc`) implements Qt's meta object system. Although this preprocessing step is very uncommon for any library it was introduced by Qt's authors for various reasons, but in particular:

> It was not possible to fully exploit the template mechanism in multi-platform applications due to the inadequacies of various compilers. Even today, many widely used C++ compilers have problems with advanced templates. For example, you cannot safely rely on partial template instantiation, which is essential for some non-trivial problem domains.

Trolltech provides a more indepth list[16] of reasons why the preprocessor approach is a good idea. In practice it doesn't make too much of a difference since the preprocessing step can be integrated very easily into build systems and will thus become transparent to the programmer.

A.1.1 User interface XML files

Qt comes with an excellent user interface construction tool called `designer` (or Qt Designer). It builds any kind of gui and in face all dialogs in the project are constructed with designer.

The complementary tool `uic`[1] converts the XML `.ui` files into compilable C++ code along with a matching header file. Thus if one wants to just change the layout of a dialog it is unnecessary to touch any code at all. Integrating `uic` into the build system is fairly trivial as well.

A.1.2 Signals and Slots

One of the new features Qt adds to C++ is the signal/slot system. Any subclass of **QObject** can have signals and slots. Signals can be emitted at any time during runtime. Slots are functions which upon receiving signals perform some action. It is required to connect signals to slots first. Thus a button will emit the signal *clicked()* if the user clicks on the button. The parent parent dialog could have a slot called *slotCancel()*. If this slot happens to be connected to the signal *clicked()* of that button, the function *slotCancel()* is executed.

In summary slots are normal functions which can be connected to signals with the *connect* function. Signals are abstract entities which can be emitted at any time using the *emit* function.

[1] User Interface Compiler

A.1.3 Properties

Although not used by this project another feature Qt adds is a property system for subclasses of **QObject** which are made available to the meta object system. An indepth explanation can be found in the Qt documentation[17].

A.1.4 Widgets

Any user interface element in Qt is called a Widget and is a subclass of **QWidget**. This includes the obvious things like buttons or menus, but also the windows and dialogs themselves. Qt relies heavily on subclassing. Thus rather than creating a new **QDialog** object and then populate it with widgets, we subclass **QDialog** and set up the widgets in the constructor.

A.1.5 Layouts

Qt uses layouts to position all of its widgets. For instance the programmer could create a **QHBoxLayout** which arranges the widgets it contains in a horizontal box. The layout objects completely take care of resizing and positioning the widgets which greatly simplifies implementation of interfaces and also makes the program a whole lot more portable (since this approach can deal perfectly with vastly different font sizes).

Appendix B

Installation Guide

The appendix outlines the steps required to compile and run the program.

B.1 Hardware requirements

None that I know of, as long as your architecture has a wordsize of at least 32 bit.

B.2 Software requirements

The following software is required to compile and run the program:

- A C++ compiler (such as g++)
- Qt version 3.3.0 or later (It might also work with earlier versions, but this was not tested.)
- A POSIX compliant libc (such as glibc)
- The libdl library for the dynamic loader

Any random recent Linux system should work. Bath universities Solaris environment will also work if Qt is installed first.

B.3 Optional software

The system can take advantage of the following software at runtime:

- iptables

B.4 Installing Qt

Usually Qt is installed on most Linux distributions as it is a requirement for the KDE desktop environment. In case it is not, these are the steps to install it:

1. Download Qt (at least version 3.3.0 or later) from Trolltech's website: `http://www.trolltech.com/download/`
2. Untar it somewhere. Inside the untared directory set up the following environment variables:
   `QT_DIR` = path to where Qt will be installed (usually `/opt/qt`)
   `LD_LIBRARY_PATH` to include `$PWD/lib`
   `PATH` to include `$PWD/bin`
3. Qt does not use the GNU autotools, but a similar configuration script is used. To show all possible options execute
   `./configure --help`
   Configure Qt to fit into your setup. Building a multithreaded, shared Qt is strongly recommended.
4. Build Qt by executing
   `make sub-tools`
   Parallel builds using the `-j` option to make are supported.

5. Install Qt by executing
   `make install`

6. Add `QT_DIR` to your profile
   Add Qt's library path (`$QT_DIR/lib`) to your /etc/ld.so.conf (on Linux)

B.5 Building the project

Currently only building the project is possible, as mentioned in section 3.8 of of the main text. This means the program is run inside the build directory. The following steps should be enough to compile and run the program:

1. Untar the source and cd into the source directory

2. The project does not use the GNU autotools either, but due to the extremely limited requirements this should not matter too much. The default compilation flags defined on the first few lines of the Makefile should work. (However, The `--export-dynamic` flag should be removed from the linking options on Solaris.)

3. Build the dependancies by running:
   `make dep`

4. Once this is done build the project and all plugins by running:
   `make`
   Parallel builds are supported as well.

5. The executable `iptv` can now be run.

Appendix C

User Manual

C.1 Introduction

When the program is started a new main window appears. The firewall displayed is the empty default firewall. Figure C.1 shows a newly started `iptv`.

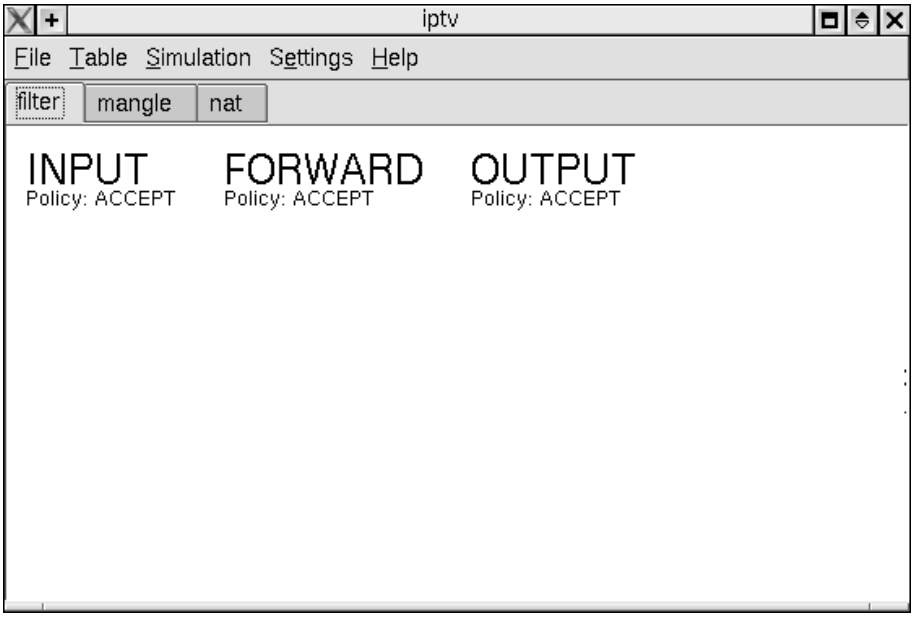

Figure C.1: A new instance of `iptv`

This window is called the 'main window' and is the primary interface of the application. It can be split into two parts: The menu bar and the document area.

In this manual 'clicking' refers to clicking with the primary mouse button, 'middle-clicking' to clicking with the middle mouse button, and 'sec-clicking' to clicking with the secondary mouse button. For a right-handed mouse they refer to left-clicking, middle-clicking and right-clicking respectively.

C.2 Main Window Menus

This section describes each of the five menus of the main window.

C.2.1 The File Menu

The file menu deals with saving and loading firewalls, and also by convention contains the *Quit* option.

New

Selecting this menu entry will create a new default firewall. It will contain the three standard tables filter, mangle and nat. Each of the tables will contain empty standard chains with their policy set to

ACCEPT. This is basically the firewall you get when booting the kernel.

Load Firewall...

This menu entry will bring up a file chooser dialog to select a firewall to load. The firewall to load must be saved in the XML firewall format (described in appendix E).

If any unsaved changes to the previously edited firewall exist, you are asked whether they should be saved or not.

Save Firewall

This causes the current firewall to be saved to its current name. If the firewall has never been saved before, i.e. it has no name, the program will ask for a name. The save format is the XML firewall format.

Save Firewall As...

Similar to the above entry, except that the program will always ask for a new name.

Import Firewall > From Running Kernel...

This menu entry will only work if:

- iptables-save is installed your computer
- you have permission to execute iptables-save

Selecting this option causes the netfilter firewall from your currently running Linux kernel to be imported into the program.

Import Firewall > From iptables-save File...

A file chooser dialog will ask for an iptables-save dump file. Such a file can be created by executing '`iptables-save > foo`' as a privileged user. The firewall contained in that file will then be loaded into the program.

Export Firewall...

This writes a shellscript containing `iptables` commands to construct the current firewall to a specified file. Such a file can be used in runlevel script to initialise the netfilter firewall during the boot process. Alternatively the file will replace the currently active firewall in the kernel by the one contained in it.

The script will first fully delete any firewall present in the kernel before updating it. This makes it possible to execute it no matter what the current firewall is, and also guarantees that the firewall will be exactly the same as stored in the script.

Quit

Selecting this entry will cause the application to quit. If any unsaved changes to the firewall exist, you are asked whether they should be saved or not.

C.2.2 The Table Menu

This menu contains options regarding the currently edited netfilter table. It is exactly the same as the table context menu as described in section C.3.4 on page 58.

C.2.3 The Simulation Menu

This menu accesses the packet simulation features of the program. Simulation is explained in full detail in section C.6 on page 63.

New Simulation...

The opens a new packet simulation dialog.

C.2.4 The Settings Menu

This menu contains entries related to the configuration of the program.

C.2.5 The Help Menu

This menu contains entries providing information about things related to the program. If the program gets ported to KDE (see section 3.8.9 of the main text) online help will also be available through this menu.

About iptv...

Displays the version, author information and license of this program.

About Qt...

Displays some information about Qt.

C.3 Using the Main Window

C.3.1 Introduction

This section deals with how to use the part of the main window where the firewall tables, chains and rules are displayed.

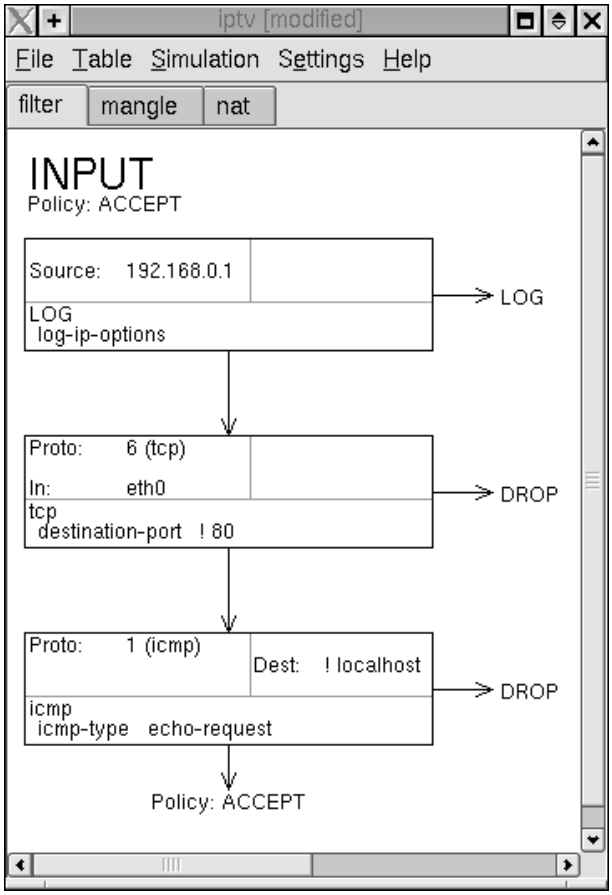

Figure C.2: An example firewall.

Figure C.2 shows one chain an example firewall. The table displayed is the filter table, because the 'filter' tab is activated. The chain shown is the INPUT chain. The big text reading 'INPUT' in this figure is called the 'chain header'. Directly underneath the chain header the chain's policy is shown.

The three boxes arranged under the chain header are each representing one rule of that chain. These boxes are called 'Rules', because that is what they represent. The topmost box represents the

first rule. The very last rule at the bottom of the chain also repeats the chain's policy. The arrow pointing to the right from each rule it that rule's target.

The text inside each box describes all conditions of the rule. The two columns in the upper half of each box contain the code conditions, the bottom section all extensions used.

C.3.2 Scrolling

In case the entire firewall does not fit onto the main window scroll bars are shown. They allow the view to be scrolled around to view any section of the firewall. The following alternative forms to scroll have are available:

- Using the mouse wheel scrolls the view up or down
- Clicking on an empty point with the primary mouse button will cause the system to enter the click-scroll state. While holding down the mouse button the view can be scrolled by moving the mouse in the appropriate direction. To exit the click-scroll mode release the mouse button.
- Middle-clicking anywhere (even over rules or chain headers) in the main view causes the program to enter the click-scroll as well. This is very useful if the table is very full and there is not much empty space to click on.

C.3.3 Moving Rules

Any rule can be moved by drag and drop. Click on a rule, hold down the mouse button and move the mouse. A bar appears where the rule would be when the mouse button would be released. Releasing the mouse button will cause the dragged rule to be moved to that position. Note that rules can be moved from chain to chain, but not from table to table. Also note that it is still possible to use the middle-click-scroll feature (or the mouse wheel) to both scroll and drag a rule at the same time.

C.3.4 The Table Popup Menu

The table popup menu can be opened by sec-clicking on any spot not occupied by a chain header or a rule. Alternatively, the popup menu for any table can be reached via the popup menu of any chain header it contains.

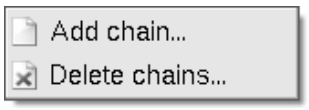

Figure C.3: Table Popup Menu

Add chain...

Selecting this entry will bring up a dialog asking for the name of a user defined chain. If the 'Ok' button in this dialog is clicked, a new, empty user defined chain will appear in the current table.

Delete chain...

This option will show a list of all user defined chains, in the current table, of which any number can be selected. Upon clicking the confirmation button all selected chains and any rules they might contain get deleted.

C.3.5 Tables

Changing the Current Table

To change the current table it suffices to click on the tab with the same name. As an alternative way of changing the current table, the cursor keys can be used. Pressing the 'cursor-left' key will cause the table to the left of the current one being selected. Similarly, the 'cursor-right' key will select the table to the right of the current one.

C.3.6 The Chain Header Popup Menu

The chain header popup menu can be opened by sec-clicking on a chain header. Alternatively, the popup menu for any chain header can be reached via the popup menu of any rule it contains.

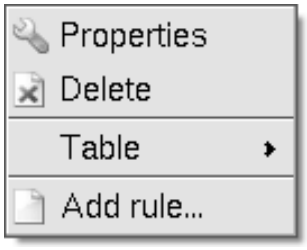

Figure C.4: Chain Popup Menu

Properties

This menu brings up the chain properties dialog of the chain the chain header represents. Section C.4 explains how to use this dialog.

Delete

Deletes the chain and all rules it contains. Only user defined chains can be deleted.

Table >

This menu entry opens a submenu which is the popup menu of the current table.

Add rule...

This section shows the rule property editor for a new rule (see section C.5.1 on how to use it). Note that the new rule is inserted at the end of the chain if the chain popup menu was opened by sec-clicking on the chain header. If the chain popup menu was obtained from the popup menu of a rule, the new rule will be added after that rule.

C.3.7 The Rule Popup Menu

The rule popup menu can be opened by sec-clicking on a rule.

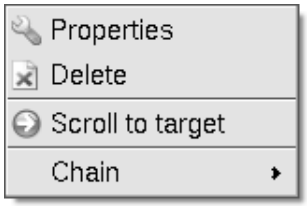

Figure C.5: Rule Popup Menu

Properties

Selecting the popup menu entry will cause the rule property editor (see section C.5.1) to be displayed for that rule.

Delete

This entry deletes the rule.

Scroll To Target

In case the rule has an user defined chain as its target, this option will cause view to be scrolled to a position in which the the chain header of that rule is visible. Furthermore, the chain header will be highlighted.

Chain >

This will cause the popup menu of the chain containing the rule to be displayed.

C.4 The Chain Property Editor

The chain property editor is used to edit chains.

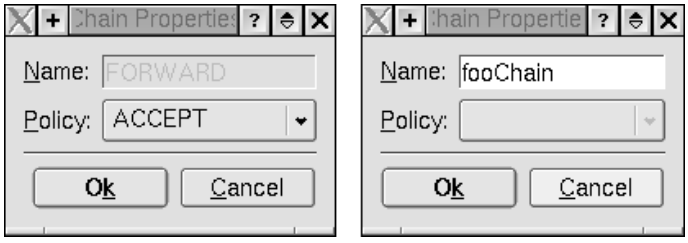

Figure C.6: Chain Property Editor

In case a standard chain is edited, as seen in the left of the above figure, only its policy can changed. The combo box offers a selection of all standard targets, user defined chains and supported target extensions.

Since a user defined chain does not have a policy, only its name can be modified. The system will check for duplicate or impossible (such as the empty name) names.

C.5 The Rule Property Editor

C.5.1 General

The rule property editor is used to change any aspect of a rule, except its position in a chain. It is divided into two section: core and extensions.

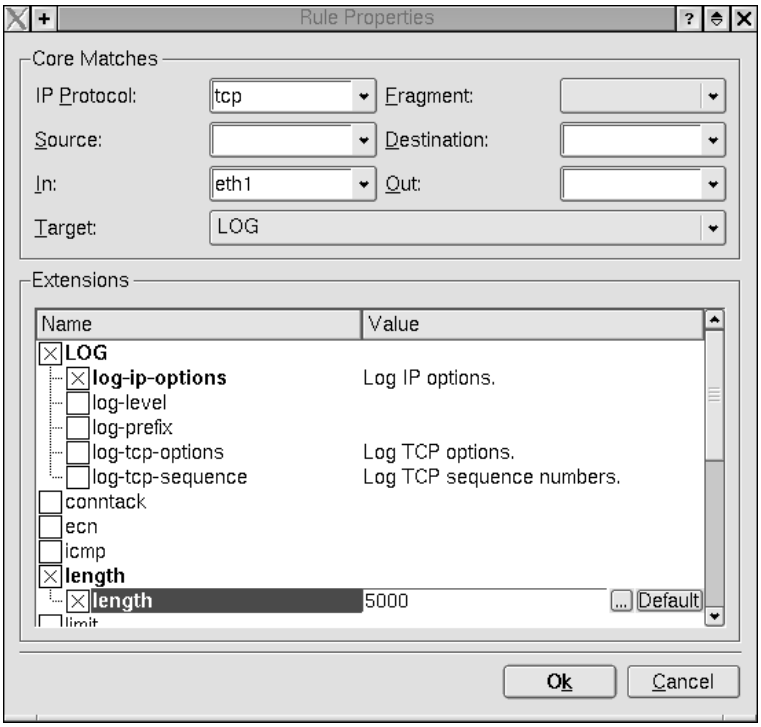

Figure C.7: Rule Property Editor

The core section simple features one field for each of the 7 core rule properties. The extension section initially displays all possible extensions. To use an extension, click on the box to the left of its name. Its name will then be displayed in a bold font and its options (if any) will be shown underneath it.

Similarly, to use an option, click on the box next to its name. Used options will be displayed in a bold font. The extensions are presented in a list with two columns: name and value. An extension option which does not take an argument, such as the log-ip-options in the above figure, can be turned on and off using only the box. The text in the value describes the action taken (or match condition) if the option is used.

More complicated options, such as the length option in the figure above, need an argument. To change its value, simple click on the name of the extension or on its value. A simple one line text editor along with two buttons will be displayed. You now have to option to either change its value right there, or to click one of the two buttons. The button labeled 'Default' will reset the value of the extension option to its default value (usually this is the empty string).

The '...' button will open another, more sophisticated editor for the selected option.
The editor shown in figure C.8 is the editor that would be shown if the '...' button for the length option would be clicked. The actual editor shown is specific to the extension option. Usually a short help text is displayed, along with a text entry field to modify the option. If an option can be inverted, a checkbox to inverted the value entered is shown.

C.5.2 More Inline Editors

Sometimes an option does not need such as '...' editor. Currently, two other such inline editors exist besides the standard text line editor:

Booleans

Sometimes an option does not take any arguments, but can be inverted nontheless. This kind of option is represented as such:

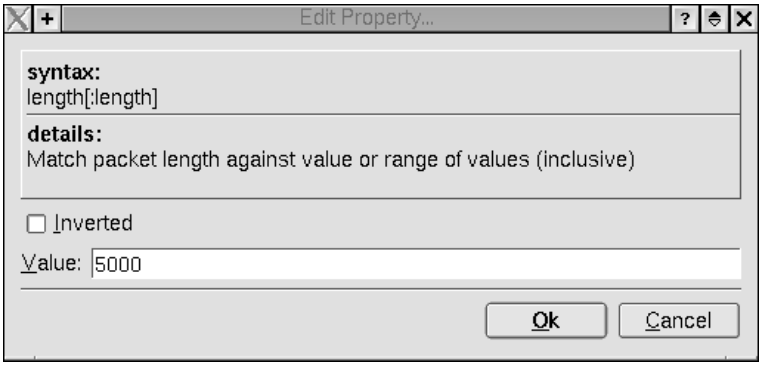

Figure C.8: The editor for the length option

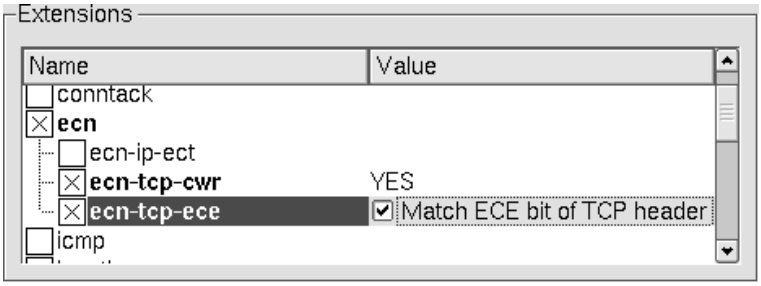

Figure C.9: Yes/No Inline Editor

Enumerates

Some options can take on value out of a list of predefined values. An example of this is the icmp-type extension option.

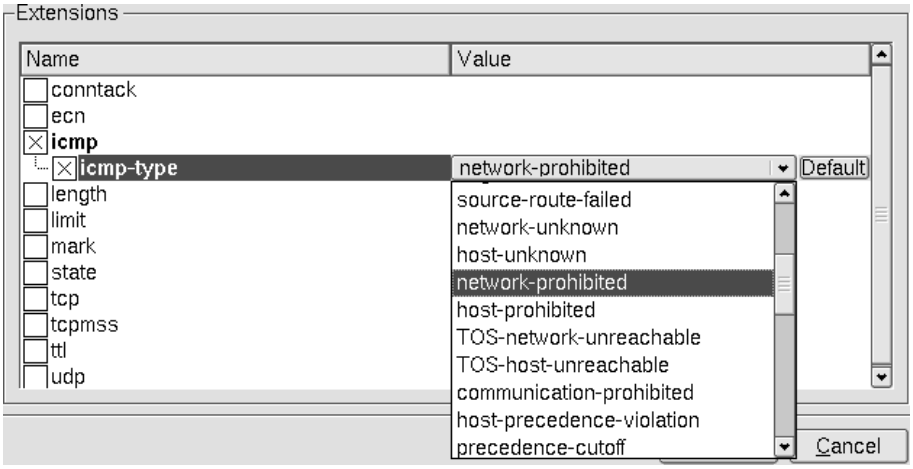

Figure C.10: Combobox Inline Editor

C.5.3 Editors

The detailed editors for the options have been made as simple as possible. However some options require more complicated editors, some of which are introduced in this section.

Portrange

The editor for port ranges looks like this:

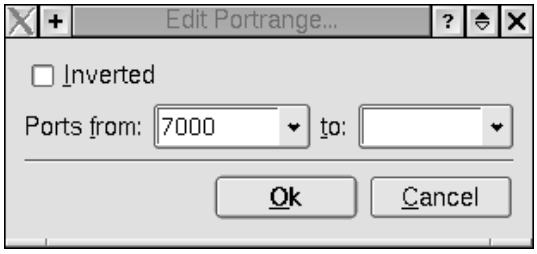

Figure C.11: Port Range Editor

Multiple Options

The editor for some of the multiple options extension options (such as the state extension option) offers a list of entries, where any combination of them can be selected by clicking on the entries.

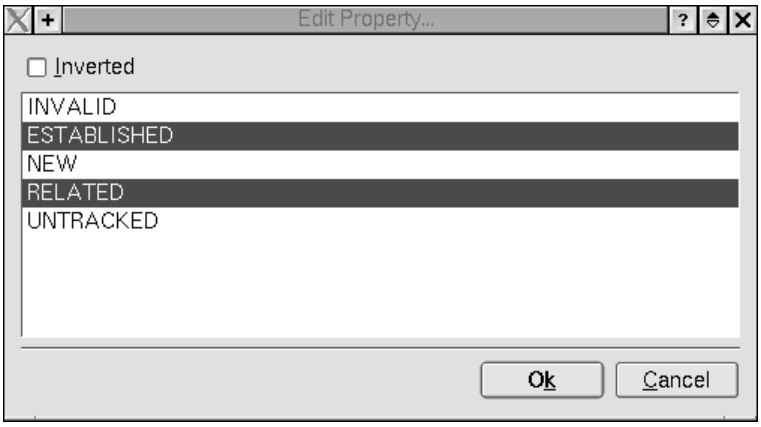

Figure C.12: Multiple Option List

C.6 Simulation

In the future an option will be available to hand-craft packets in a simple and user-friendly way. At the moment packets for simulation have to be captured off the wire with a tool such as `tcpdump`. (Available from `www.tcpdump.org`)

To start a simulation, select the 'New Simulation...' menu item from the 'Simulation' menu, or press CTRL+M.

Initially, the list in the top section will be empty. Clicking the import button, and then choosing a tcpdump dump file, will fill this list with the packets contained in that file. Currently, only ethernet dumps are supported, support for different link layer protocols (such as PPP) will be added in a future version.

Three routing modes are available: send, receive and route. They specify which chains the packet must pass through. In the current version only chains in the 'filter' table are used, support for the other tables will be added in a future version.

The comboboxes labeled 'Input' and 'Output' specify the network interface the packet arrives on and leaves the box. They can be set to 'wrong' values to simulate spoofed packets.

Clicking on one of the (IP) packets in the list will show the path of the packet through the firewall.

The arrow drawn next to the rules represents the packet. In figure C.14 the packet gets logged by the second rule, and then accepted by the last. The little bar at the end of the final arrow indicates the end point of the packet. In a future version of the program, conditions that cause a rule to not match will be highlighted.

Even if a simulation is active, the firewall can still be modified in the usual way, and the visualisation will instantly reflect the new path of the packet.

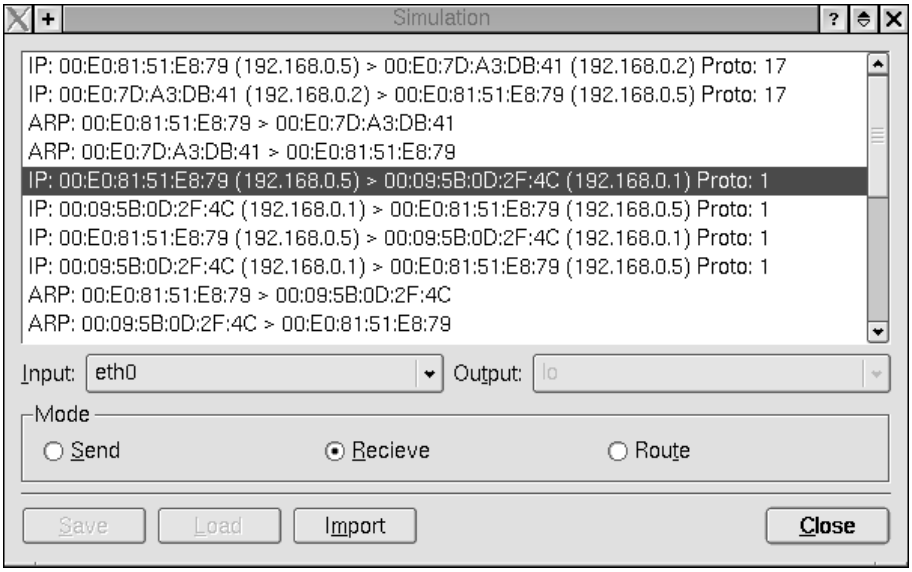

Figure C.13: Simulation Dialog

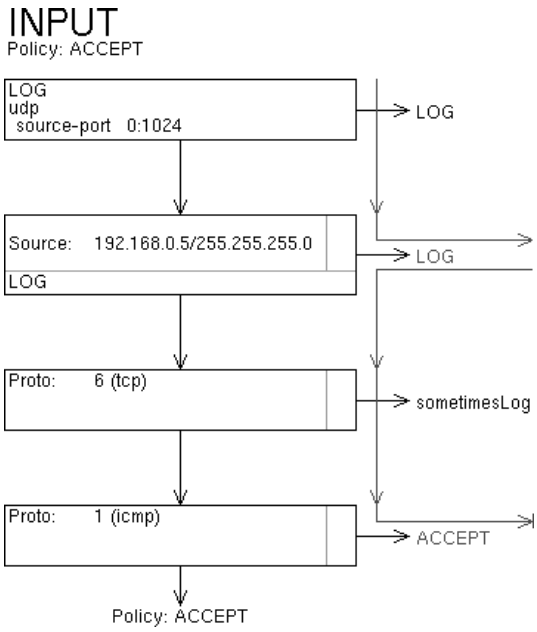

Figure C.14: A Simulated Packet

Appendix D

Adding New Extensions

There are essentially two steps to perform in order to add support for a new extension into the system: creating the extension XML file and the shared object containing the simulation code. This appendix is intended to serve as a complete manual and reference for adding new extensions into the system.

D.1 The extension XML file

D.1.1 Introduction

The first step to support a new extension is to create an XML file describing it. The information in the XML file will be sufficient to:

- Load, save, create or edit a firewall containing this extension
- Create a firewall shellscript containing this extension
- Importing an `iptables-save` file containing the extension
- Importing a firewall from a running kernel containing the extension

For simulation to work properly the shared object is also needed.

The XML file should be well formed (currently the parser is not very fond of errors), otherwise an extension will only be loaded partially (discarding all information after the error) or not at all.

The XML file should be named in the general way of `extension.ext` and placed in the directory `extensions/xml/`. Thus the extension file for the LOG target extension would be called `LOG.ext`.

An extension XML file contains one <extension> tag at top level describing the entire extension.

D.1.2 The `extension` tag

Description

This is the top level tag. It describes one extension and its options.

Attributes

Attribute	Description
type	Type of the extension: 'match' or 'target'
name	The name of the extension
noreturn	If specified, the target extension decides upon the fate of a packet.
simulation	The file name of the shared object to load for this extension.

Allowed Content

The following tags are allowed inside this tag:

- option

D.1.3 The `option` tag

Description

This tag describes an extension option. It can only be occur inside an <`extension`> tag.

Attributes

Attribute	Description
name	Name of the extension option
type	The argument type of the extension option
caninvert	If set to 'true' the extension can be inverted. Default: false
postinvert	If set to 'true' the exclamation mark to invert occurs *after* specifying the extension, if set to 'false' the exclamation mark occurs *before* specifying the extension.
argcount	The number of (space-separated) arguments an extension option takes. A sensible default is chosen which depends on the type attribute.

The *type* attribute can take one of the following values:

Value	Description
none	No arguments. The options stands on its own.
portrange	For single ports or portranges with a ':'
portrangedash	For single ports or portranges with a '-'
nat	For NAT rules
enumlist	A combination out of several predefined values
enum	Just one value out of several predefined values
proto	An IP protocol
ugly	The option does not fit into any of the above.

Allowed Content

The following tags are allowed inside this tag:

- enum
- alias
- description
- detaildescription

D.1.4 The `enum` tag

Description

This tag defined predefined values for the enum and enumlist types. It can only occur inside the <`option`> tag. One <`enum`> tag is needed per value. The value is enclosed between the start and the end tag.

Attributes

None.

Allowed Content

The value to be one of the enumerated options.

D.1.5 The `alias` tag

Description

This tag defined an alternative name for an option. One example would be the destination-port extension option which is also known as the dport extension option. It can only occur inside an <`option`> tag. Any number of aliases can be defined. The alias is enclosed between the start and the end tag.

Attributes

None.

Allowed Content

The alias of the extension option.

D.1.6 The `description` tag

Description

This tag provides a brief description for options which take no arguments (of type 'none'). The description is enclosed btween the start and end tag. It can only occur inside an <`option`> tag. Only one such description should be defined per option.

Attributes

None.

Allowed Content

The brief description. This gets displayed in the **ExtensionList** widget directly and thus should be *really* brief.

D.1.7 The `detaildescription` tag

Description

This tag provides a more indepth description of the extension option. It is used for options of the type 'ugly'. It can only occur inside the <`option`> tag.

Attributes

None.

Allowed Content

The following tags are allowed inside this tag:

- syntax
- details

D.1.8 The `syntax` tag

Description

This tag defined the syntax of the option or the kind of value expected. It can only occur inside the <`detaildescription`> tag.

Attributes

None.

Allowed Content

The syntax.

D.1.9 The `details` tag

Description

This tag further describes the sytax of an extension option's argument. It can only occur inside the <`detaildescription`> tag.

Attributes

None.

Allowed Content

A description for the syntax or some other help about the exention option.

D.2 The shared object

D.2.1 Introduction

To create the shared object a new C++ file in the project root is created with a naming scheme of `ext_name.cpp`. Thus the filename for the udp extension would be `ext_udp.cpp`. Any class of the main program or the Qt library can be used when writing an extension's simulation code.

D.2.2 The init function

The extension system will call a function called *init_extensionname* once the shared object is loaded. This function must have exactly the following signature:

```
void init_extensionname(Extension *);
```

As usual 'extensionname' is replaced with the actual name of the extension, as defined in the 'name' attribute of the <`extension`> tag in the XML file. This function's task is simply to connect some function pointers.

There are two different kinds of functions the shared object has to provide: a function for the extension itself and a function for each option.

D.2.3 The extension function

The function for the extension itself must have the following signature (the actual name is irrelevant):

```
bool foo(IPPacket *);
```

It is meant to return true if an extension itself matches and false if not. In the case of a target extension it may also modify the packet.

D.2.4 The extension option functions

The functions for the options must have the following signature (again, the actual name is irrelevant):

```
bool foo(IPPacket *, ExtensionOptionUse *);
```

Similar to the above function, it is meant to return true of the condition matches and false if not. Note that the function *must not* take into account the inversion flag which may or may not be set in the **ExtensionOptionUse** class, this will be dealt with by the **Simulation** class. In the case of a target extension it may also modify the packet.

D.2.5 Connecting the function pointers

To set up the function pointers the following lines of code might be of help:

```
ExtensionOption *eo;

e->has_simulation = true;
e->simulate = &foo;

eo = e->getOptionByName("bar");
eo->has_simulation = true;
eo->simulate = &foo_bar;
```

We assume e is the pointer to the **Extension** class passed as an argument to the init function. It is planned to provide a cleaner way of doing this in the future.

D.2.6 Adding the extension to the build system

In order to automatically build the shared object along with the rest of the project one has to add the name of the compiled shared object (`ext_tcp.so` in case of the tcp extension) to the `IPTV_SO` variable in the top level `Makefile`. After that, running `make dep` to update the dependancies is recommended.

D.3 An Example

A simple and good example are the three proof of concept implementations of the tcp, udp and icmp match extensions. They are part of source tree of the project.

Appendix E

Firewall File Format

E.1 Introduction

In practice these XML files are never created by hand, but a complete specification is given to be complete.

The firewall XML file contains the <`firewall`> tag at the top level.

E.2 Tag reference

E.2.1 The `firewall` tag

Description

This is the top level tag. It describes a netfilter firewall, and possibly others in the future.

Attributes

None yet. A type attribute indicating whether it is an iptables or iptables6 firewall is planned in the future.

Allowed Content

The following tags are allowed inside this tag:

- table

E.2.2 The `table` tag

Description

This tag describes a netfilter table. It can only occur inside a <`firewall`> tag.

Attributes

Attribute	Description
name	The name of the table

Allowed Content

The following tags are allowed inside this tag:

- chain

E.2.3 The `chain` tag

Description

This tag describes a netfilter chain. It can only occur inside a <`table`> tag.

Attributes

Attribute	Description
name	The name of the chain
policy	The policy of a builtin chain

Allowed Content

The following tags are allowed inside this tag:

- rule

E.2.4 The `rule` tag

Description

This tag describes a netfilter rule. It can only occur inside a <`chain`> tag.

Attributes

Attribute	Description
target	The target of the rule
proto	Match for IP protocol. Can be name or number.
fragment	If specified as 'fragment' second or further fragment only. If specified as '!fragment' match first fragment only. If not specified at all always match.
source	Source IP address or hostname
dest	Destination IP address or hostname
in	Input network interface
out	Output network interface

Allowed Content

The following tags are allowed inside this tag:

- extension

E.2.5 The `extension` tag

Description

This tag uses an extension in a rule. It can only occur inside a <`rule`> tag.

Attributes

Attribute	Description
name	The name of the extension to use

Allowed Content

The following tags are allowed inside this tag:

- option

E.2.6 The `option` tag

Description

This tag uses an extension option in a rule. It can only occur inside a <`extension`> tag.

Attributes

Attribute	Description
name	The name of the extension option to use
inverted	Specified as 'true' if the option should be matched inversely
value	The argument(s) to the option

Allowed Content

None.

Appendix F

Source Code

Due to the length of the source code, it will not be reproduced here. It is, however, available on the cd handed in with the project. Furthermore, it is available at the following URL (as long as the author is at the University of Bath): http://students.bath.ac.uk/ma1flfs/fyp/iptv-0.9.0.tar.bz2