A Simple, Configurable, and Adaptive Network Firewall for Linux

James M. Westall

Department of Computer Science
Clemson University
Clemson, SC 29634

Abstract- It is increasingly important that workstations, especially those with permanent connections to the Internet, be defended against network delivered attacks. Firewalls constitute a useful component of the defense arsenal. However, the standard Linux tools for constructing firewalls have significant limitations. Simple firewalls can be built, but they protect at one of two extremes. They either assume that every host on the Internet, not known *a priori* to be trustworthy, is malicious and thus interfere excessively with legitimate use of the network, or they assume that every host, not known *a priori* to be malicious, is trustworthy and thus provide little, if any, protection. The IP masquerading facility provides a more useful firewall, but it is complex to configure and requires that a workstation be set aside as a dedicated gateway. In this paper we provide a survey of firewall technologies and describe the design and implementation of a new facility for providing firewall protection to Linux hosts. We believe the new facility provides a reasonable balance of protection and unobtrusiveness while retaining both simplicity and efficiency.

Keywords: firewalls; network security; Linux.

1 Introduction

The past ten years have seen a tremendous increase in the number of user administered host systems that are directly attached to the Internet. Unfortunately, this growth has been accompanied by a corresponding increase in the number of network-based attacks being directed at seemingly randomly chosen victim systems.

Since automated tools, which are readily available on rogue web sites, are commonly used to launch these attacks, virtually every host system that has a permanent connection to the Internet has been, or eventually will become, a target. Thus, it is increasingly important that host systems attached to the Internet be equipped with defense mechanisms.

One obvious defense is to increase the robustness of system software so that target systems become less vulnerable. The regular distribution of *ad hoc* security patches by all major vendors of system software is evidence of the ongoing work in this domain. Despite these efforts, the goal of providing system software that is not vulnerable to attack remains very elusive.

The continued vulnerability of system software to network based attacks[1] has motivated the development of alternative defense mechanisms designed to prevent the delivery of packets associated with an attack to their intended targets. Defense mechanisms of this type are generally known as *firewalls*.

The focus of this paper is a firewall system for Linux, somewhat unimaginatively named *fw*, that was developed in the Department of Computer Science at Clemson University. The development of *fw* was motivated by several successful intrusions that took place approximately one year ago. Attackers from outside the University gained root privileges on Linux workstations located in faculty offices and research labs. Then, as is often the case, they proceeded to use our compromised systems to launch further attacks on systems located at other universities.

In the remainder of this introduction we provide a brief survey of firewall technologies. Firewall designs are now generally classified as belonging to one of two, three, or four broad categories[4, 6, 11]. Proponents of the four category classification identify: packet filters; circuit level gateways; application level gateways; and stateful inspection[11]. We prefer the two category classification: packet filters (including stateful inspection filters) and gateways. Firewall design objectives are independent of design classification and include the following:

- preventing packets associated with an attack from reaching their targets (safety);
- *not* interfering with legitimate use of the network (unobtrusiveness);

[1] Defending against virus bearing objects embedded in voluntarily received data such as e-mail is not a firewall mission, and our use of the term *attack* throughout the remainder of the paper is assumed to exclude this type of problem.

- ease of installation and management (simplicity);
- minimal processing overhead and delaying of legitimate packets (efficiency).

These design objectives are clearly mutually conflicting. The safest firewall, disconnecting the network cable, is also the most obtrusive. It is also the case that the security objectives of an academic research lab are not (or at least should not be) the same as those of a nuclear weapons research lab. Therefore, some degree of configurability is also a desirable attribute of a firewall system.

1.1 Packet filters

The earliest and simplest type of firewall is the *packet filter*. Packet filters are implemented as code modules within the protocol stack of a host or router and are driven by rule sets called *filter rules*. These rules are applied to every packet that is sent or received and specify whether the packet should be discarded or routed normally. A packet is said to match a rule if some combination of IP address, port number, and transport protocol number matches the corresponding elements of the rule. When a packet matches a rule, that rule is applied to the packet.

Rule sets are often characterized as being permissive or non-permissive. With permissive rule sets, the default condition is to route the packet normally, and the rules specify conditions under which a packet is to be discarded. Gateway routers are often configured in this way. For example, rules may demand that "suspicious" packets, such as those that are source routed or appear to carry spoofed source addresses, be dropped, but others are allowed to pass. With non-permissive rule sets, a packet is discarded by default unless it matches a rule that specifies it should be routed normally.

Packet filters that use permissive rule sets tend to be unobtrusive, but not very safe. In contrast, non-permissive rule sets can provide a large measure of safety but can be so obtrusive as to be unsuitable for the academic environment. A tool such as Mason can be used to reduce the obtrusiveness problem somewhat. It should be emphasized that Mason is *not* a firewall, but rather a tool for generating rule sets that can make a non-permissive use of the standard Linux packet filters less obtrusive. Mason provide a learning mode in which: "You leave Mason running on the firewall machine while you are making all the kinds of connections that you want the firewall to support (and want it to block). Mason gives you a list of firewall rules that exactly allow and block those connections[3]." The obvious limitation of this approach is the implied requirement that the user have *a priori* knowledge of all the web hosts in the world that should be allowed or blocked.

Stateful inspection is a technique now employed by some of the more prominent commercial firewall products. Stateful inspection filters differ from traditional packet filters in significant ways. First, the rule set is typically more complex and includes rules that are applied to network, transport, and application layer headers. Second, rules can be dynamically created and destroyed when network connections or pseudo-connections are established or closed. The advantages and disadvantages of stateful inspection have been summarized in the following way: "Stateful multilayer inspection firewalls offer a high level of security, good performance and transparency to users. They are expensive, however, and due to their complexity are potentially less secure than simpler types of firewall if not administered by highly competent personnel[11]."

1.2 Gateways

The second major class of firewall is the *gateway*. In the context of firewalls, a gateway is a typically a router or host system, sometimes referred to as a *bastion host* [4, 6], which is either logically or physically interposed between the external Internet and the hosts that it protects. Since the bastion host is itself subject to attack, it should export a carefully screened set of services to the outside world. To facilitate recovery in the event that the bastion is compromised, it should ideally host only those files needed to boot and run the operating system.

Two varieties of gateway are in widespread use. The *circuit level gateway* shown in figure 1 is commonly known as *IP masquerading* or as a *NAT* (network address translation) gateway [2]. Here, the bastion host contains at least two network interfaces. One interface connects it to the outside Internet and others to local networks containing the protected hosts. The local networks commonly use addresses in the $192.168.x.x$ space that is reserved for private networks and is not routeable in the Internet. The bastion host is configured as the default gateway for the protected hosts making the *NAT* process completely transparent to applications running on the protected hosts.

When a protected host wishes to contact a host in the external Internet, it simply sends the packet to its default gateway, the bastion. The bastion changes the source IP address to the address of its own interface to the external Internet, changes the source port number if it is already in use, records the original source IP address and source port number in a table indexed by final source port number, and then forwards the packet.

When a packet arrives from the external Internet, the bastion performs a table lookup using the destination port number. If a valid mapping entry exists, the destination IP address and port number are replaced with the values found in the table and the packet is forwarded to the private network. This technique allows systems behind the bastion to

[2] In addition to its role as a firewall, a *NAT* gateway also provides a mechanism through which a user with only a single ISP provided address can connect multiple systems to the Internet.

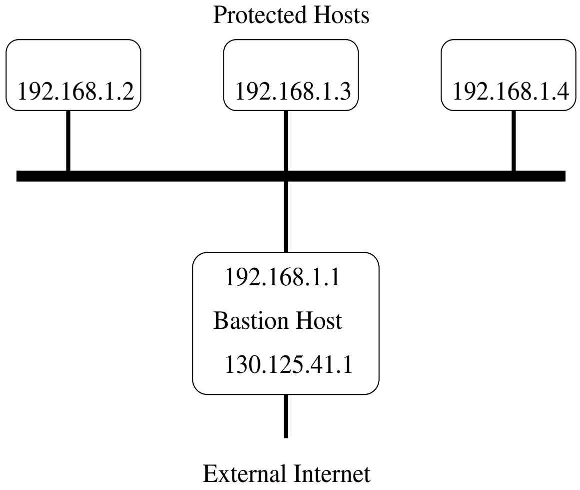

Figure 1: A *NAT* Gateway.

establish TCP sessions or UDP pseudo-sessions with hosts in the external Internet but not vice versa.

NAT is also not without disadvantages. It requires that a bastion host be available, it is non-trivial to configure and, as pointed out by the author of the Linux HOWTO on the subject, "Some TCP/IP application protocols will not currently work with Linux IP Masquerading because they either assume things about their port numbers or encode TCP/IP addresses and/or port numbers in their data stream[5]."

The second major class of gateway is the *application level gateway* which is also referred to as a *proxy host*[11]. In a proxied environment, protected hosts must be configured to send all service requests to the proxy host. The proxy host reissues the request to the intended destination on behalf of the protected host. When the response is returned to the proxy host, it is forwarded it to the protected host on whose behalf it was issued.

Proxies can provide a very secure environment and, in the web environment, can provide significant performance increases via caching. However, their use also has a number of significant disadvantages. Proxies must be deployed on a per application (e.g. http) protocol basis and thus, when used as a firewall the access of the protected hosts to the external Internet is severely constrained. Because of these constraints, the use of proxy hosts as firewalls will not be further considered in this paper.

2 Firewall support in Linux

Linux has included support for both packet filtering firewalls and *NAT* gateways (IP masquerading) since at least kernel version 2.0. The programs *ipfwadm* and *ipchains* may be used with version 2.0.x and 2.2.x kernels respectively to specify filter rules for handling input, output, and forwarding of packets. Both packet filters and *NAT* gateways can be built upon their rule sets. An excellent description of the use of *ipfwadm* to set up a *NAT* gateway can be found in [7], and a more complete description pertaining to 2.0.x, 2.2.x, and 2.4.x is contained in the Linux IP Masquerade HOWTO[5].

When *ipfwadm* supported filter rules fail to address a particular need, the installable kernel module facility in Linux [1, 2] together with the *register_firewall()* kernel function provides a convenient mechanism for constructing *ad hoc* firewalls. A kernel module is a collection of C language functions containing at least two functions with the specific names, *init_module()* and *cleanup_module()*. The collection is compiled as a single object file, say *fw.o*, and then loaded with the command */sbin/insmod fw.o*. The *init_module()* function is executed when the module is loaded, and the *cleanup_module()* function is executed when the module is removed, via the */sbin/rmmod fw* command. The module facility is generic in nature and supports installable file systems, character, block, and network device drivers in addition to firewalls. The *init_module()* function for an installable firewall is written as follows:

```
int init_module(void)
{
   int rc;
   rc = register_firewall(PF_INET, &fw_ops);

/* Kernel routines use printk to  */
/* print to the system log.       */

   printk("Reg_Fw returned %d \n", rc);
   return(rc);
}
```

The *fw_ops* entity that is passed to *register_firewall* is a kernel defined structure that is statically initialized by the firewall as shown:

```
struct firewall_ops fw_ops =
{
  0,             /* Next firewall */
  fw_forward,    /* Forward       */
  fw_input,      /* Input         */
  fw_output,     /* Output        */
  PF_INET,       /* PF            */
  255            /* Priority      */
};
```

Multiple firewalls are supported, and the null link field is used by the kernel to chain them together in priority order. The output, input, and forward fields are pointers to functions that are invoked each time a packet is transmitted, received or forwarded. The packet in question is either dropped or processed normally depending upon whether the function returns *FW_ACCEPT* or *FW_BLOCK*.

All three functions share a common interface as illustrated in the following permissive input packet filter:

```
int fw_input(
struct firewall_ops *this,
int pf,
struct device *dev,
void *phdr,
void *arg)
{
   unsigned int     addr;
   struct iphdr     *iph;
   struct fwnetype  *ne;
   int              rc = FW_ACCEPT;

   iph = (struct iphdr *)phdr;
   addr = ntohl(iph->saddr);

   if (is_badguy(addr))
      rc = FW_BLOCK;

   return(rc);
}
```

In the above example, *is_badguy()* is a function which is contained in the firewall module and is responsible for determining if the source IP address of the incoming packet matches a drop rule. To implement a non-permissive strategy, it is necessary only to change the default return code to swap *FW_BLOCK*s and *FW_ACCEPT*s and replace the call to *is_badguy()* with a call to *is_goodguy()*.

Firewall support in Linux changed significantly in the transition from 2.2.x to 2.4.x kernels. The new facility is referred to as *netfilter*, and it provides hooks that are functionally equivalent to those of *register_firewall()* via the *nf_register_hook()* kernel function.

3 The *fw* firewall

Although the firewall construction tools provided in Linux distributions appear to be both reasonably flexible and comprehensive, we found that they did not match our needs very well. The packet filters that could be configured with *ipfwadm* are limited to purely permissive or purely non-permissive behavior. The permissive configuration was not safe enough, and the non-permissive configuration, even when augmented with Mason[3] was too obtrusive. The capabilities of the *NAT* gateway better fit our needs, but it required a dedicated bastion and was a nuisance to set up and administer. This motivated the development of *fw*. Our design objective was to develop a firewall whose safety and unobtrusiveness were comparable to that provided by a *NAT* gateway, but which was simple to configure and did not require a dedicated bastion host.

The *fw* firewall is a packet filter built upon the framework described in the previous section. The filter is driven by both static rules and dynamically created rules. The dynamically created rules automatically expire unless periodically refreshed, and thus represent the soft-state of the system. This approach is in contrast to the use, by some commercial stateful inspection filters, of hard-state information pertaining to the existence of connections. The relative merits of hard-state versus soft-state in various aspects of network engineering have been much discussed. The disadvantages of the hard-state approach are summarized by Turner [10] (in an IP versus ATM context) as follows:

- Correct operation depends critically upon state consistency.
- Loss of consistency due to design flaws or hardware/software failure can be catastrophic.
- Hard-state systems require extensive error recovery code (can be $> 90\%$ of total system code).

Soft-state information must be periodically refreshed and is discarded after a period of time if it is not. Thus, loss of state is "designed in", and it is argued that soft-state based systems are simpler, smaller, and less subject to catastrophic failures such as unanticipated deadlocks or livelocks.

The rules that control the operation of *fw* are structures consisting of four elements.

```
typedef struct
{
   unsigned int prefix; /* IP addr pfx */
   int          pfxlen; /* Pfx length  */
   unsigned int action; /* Action bits */
   unsigned int timeout;/* Expiry time */
}  fw_rule_t;
```

The first two fields are used to specify an IP address or a range of IP addresses. Addresses are specified using the standard CIDR (Classless Interdomain Routing) [9] mechanism in which the prefix specifies the most significant bits of the first address in a block of contiguous IP addresses, and the prefix length provides the number of bits that are significant. The rule matching logic seeks to find the longest prefix that matches the non-local IP address carried by a packet. The timeout value specifies the absolute time in standard Unix time format at which a dynamic rule expires. A value of -1 means that the rule is permanent.

The action bits define the actions to be taken when the rule is matched.

```
#define DENY      1
#define ALLOW     2
#define LOG       4
#define DYNAM     8
#define CREATE   16
```

The *DENY* and *ALLOW* bits specify whether a packet should be discarded or processed normally. When a packet being *transmitted* by the protected host matches a rule in which the *CREATE* bit is present, a new rule is created. The action bits of the new rule are set to *ALLOW* | *DYNAM*, and the prefix is set to the destination address in the transmitted packet. The prefix length and timeout period length for new rules are configurable parameters of the firewall. We are presently using a prefix length of 32 and a timeout of 120 seconds.

The resulting behavior is that all packets from external hosts whose addresses match a *DYNAM* | *DENY* rule are dropped *unless* the host running the firewall initiates the contact. After contact has been initiated, *all* packets sent from hosts matching the newly created rule are accepted until the rule expires. Subsequent transmissions which match a dynamically created rule cause the expiration time of the rule to be extended by the length of the timeout period. This discussion is illustrated with using the following rule set:

```
fw_rule_t rule_base[MAX_RULES] =
{
  0x00000000,   0,   CREATE | DENY | LOG, -1, /* All   */
  0x00000000,  32,   DENY | LOG, -1,   /* 0.0.0.0      */
  0x3f0a0000,  16,   DENY | LOG, -1,   /* UUnet DHCP   */
  0x827f3000,  24,   ALLOW,      -1,   /* 130.127.48   */
  0x827f3800,  24,   ALLOW,      -1,   /* 130.127.56   */
  0xc0a80100,  24,   ALLOW,      -1,   /* ATM CLIP net */
  0xc0a80200,  24,   ALLOW,      -1,   /* ATM LANE net */
  0x7f000000,   8,   ALLOW,      -1,   /* Local host   */
  0x827f0e0e,  32,   ALLOW,      -1,   /* Mickey       */
};
```

Every IP address matches the first rule because its prefix length is zero. Therefore, if the protected host transmits a packet to an address, $a.b.c.d$, that does not match one of the other eight prefixes, a dynamic rule is created having prefix, $a.b.c.d$ and action *ALLOW* | *DYNAM*. Subsequent packets transmitted to or received from $a.b.c.d$ will best match the new dynamic rule and will be allowed to pass through the firewall. When no packets are exchanged during an interval of time longer than the timeout period, the rule is destroyed, and any subsequent packets arriving from $a.b.c.d$ are dropped.

The second and third rules unconditionally block input and output to addresses 0.0.0.0 and 63.10.x.x. The final six rules allow the enumerated subnets and hosts unrestricted access to the protected host even when the protected host does *not* initiate the contact.

The presence of the *LOG* bit in a *DENY* rule causes a log record to be created each time an input packet that matches the rule is dropped. The presence of *LOG* in a *DENY* | *CREATE* rule causes a log record to be generated each time a dynamic rule is created.

4 Performance evaluation

In this section we provide a qualitative assessment of the performance of *fw* and firewalls of alternative designs relative to the design objectives identified in the introduction. Simplicity and efficiency are clear strengths of *fw*. Since the rule matching logic is driven by IP address prefix alone, no transport protocol dependent code nor processing of transport layer headers is required. Our experience shows that, even in the presence of aggressive web surfing, the number of dynamic rules remains small ($<< 50$). Therefore, the rule matching logic is implemented as a linear scan of the rule base. The rule matching function requires only 13 lines of executable code and the rule creation logic is similar in size. Thus processing overhead imposed by the use of *fw* is comparable to that of other simple packet filters and considerably less than that of a *NAT* gateway.

We consider unobtrusiveness and safety in the context of three categories of trustworthiness associated with a remote host: trusted; unsafe; and unknown (not (trusted | unsafe)). Both *fw* and stateful inspection firewalls can be configured to give unconstrained access to trusted hosts or networks and are thus perfectly unobtrusive with respect to these. Such is not the case with *NAT* gateways because of the possibility that addresses or port numbers may be carried in the application data stream.

Both *fw* and stateful inspection firewalls can be configured to deny *all* access to hosts or networks known to be unsafe. This behavior is as safe as possible and perfectly obtrusive (desirably so). If a *NAT* gateway cannot be configured in this way, it may be augmented with a firewall.

In a typical academic environment, the vast majority of Internet hosts are neither trusted nor considered unsafe but belong to the unknown category. A reasonable strategy in dealing with the unknowns is to be generally unobtrusive when the protected host initiates contact, try to minimize the degree of exposure if the unknown should turn out to be unsafe, and to prevent hosts of unknown intention from initiating contact. This strategy is supported by *fw*, stateful inspection firewalls, and *NAT* gateways.

Despite the intent to be unobtrusive when the protected host initiates contact, there are situations in which *fw*, *NAT* gateways, and even stateful inspection firewalls can be obtrusive. *NAT* gateways not only suffer from the problem of network addresses and port numbers being embedded in the application data stream, but also do not support private transport protocols, and may not support public transports, such as RTP, that are not in widespread use.

Both *fw* and Linux *NAT* gateways are subject to timeouts when a session or pseudo-session remains idle for an extended period. If this occurs in a situation in which session layer semantics dictate that the remote host speak next, then that session will fail. It is suggested by Ranch[5] that running a *ping* process with an interval somewhat smaller than the timeout is an effective (if slightly ugly) way to address this problem.

Finally, all can be obtrusive with respect to the "hand-off" illustrated in figure 2. The use of a configurable prefix length in *fw* was motivated by the premise that the target of a legitimate hand-off was likely to be on the same subnet as the original server. However, results of tests using various prefix lengths indicate that, if and when this type of hand-off occurs in typical web surfing, it is made transparent by network ad-

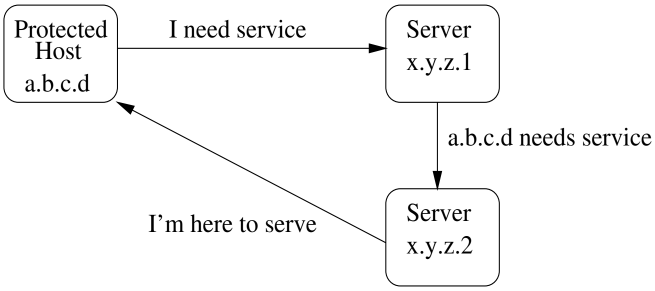

Figure 2: Server hand-off.

dress translation at the server end. These tests also showed that when these hand-offs do occur they are more likely to be in the context: "Send $a.b.c.d$ an annoying pop-up advertisement." So a bit of obtrusiveness is not always a bad thing!

In summary we know of no operational aspect in which *fw* is more intrusive than a *NAT* gateway. Because of the possibility of early timeout on idle connections, both *fw* and a soft-state *NAT* gateway are potentially more obtrusive than a stateful inspection filter that uses hard-state information. Since *fw* rules consider only IP address, *fw* never intrudes on the use of private or uncommon transport protocols as stateful inspection filters or *NAT* gateways may.

Safety is more difficult to assess than unobtrusiveness. Even under the assumption made here, that a firewall system has been correctly configured and is working as intended, accepting *any* packet from a host of unknown trustworthiness involves some risk. Nevertheless, it is possible to gain insights into safety by characterizing the risk in three ways: from what hosts is the protected host at risk; how long does the period of vulnerability last; and to what type of attacks is the protected host subject during the period of vulnerability.

Under the assumption that the firewall is working as intended, *fw*, stateful inspection firewalls, and *NAT* gateways are at risk from packets bearing the source address of essentially the same collection of hosts: those that are trusted and those that have been contacted "recently" by the protected host. This alone reduces the number of potential untrusted sources of attack at any instant in time from greater than 10^7 to a typical maximum of no more than 10.

The period of vulnerability is also similar for *fw*, stateful inspection filters, and *NAT* gateways. It is the duration of the session or pseudo-session with the possible addition of a short timeout period.

It is in the types of attack to which the protected host is vulnerable during periods of vulnerability that *fw* trades off safety in favor of increased simplicity and unobtrusiveness. In contrast to both *NAT* gateways and stateful inspection filters which pass only traffic carrying protocol and port numbers appearing to belong to legitimate sessions or pseudo-sessions, *fw* will pass any packet carrying an approved IP address. We accept this tradeoff not only to reduce obtrusiveness, but also to gain simplicity, which in turn reduces the risk that the firewall is not working as expected.

Marginal increases in safety can also be obtained by use of multiple defense mechanisms (often characterized using the military term *defense in depth*). We use *fw* as a final line of defense behind a front-line defense[3] of routers using permissive rule sets that block source routed packets and packets with spoofed source addresses.

5 Conclusion

In this paper we have provided a survey of existing firewall technologies, presented the design and implementation of the *fw* firewall for Linux, and analyzed its performance with respect to accepted firewall design objectives. We believe that *fw* is clearly simpler, marginally more efficient and less obtrusive, but marginally less safe than the alternative designs considered.

We believe that the capabilities of *fw* reasonably match the security needs of the academic environment. In the year's time that *fw* has been in use in the Departments of Computer Science and Electrical and Computer Engineering we know of no protected system being successfully compromised. The logging mechanism of *fw* has allowed us to identify and report several persistent but unsuccessful attacks and a few instances in which non-malevolent hosts were malfunctioning.

There do exist opportunities for the improvement of *fw*. One readily identifiable deficiency is that there is no tool similar to *ipfwadm* with which to add or delete static rules. All such rules must be present in the source code and the module compiled and reinstalled each time a rule is changed. When large rule bases are in use, the efficiency of *fw* could be improved by replacing the linear table structure with a Patricia tree[8].

It could also be argued that the rule matching logic should be extended to support protocol numbers, port numbers, and hard connection state. We would argue that if this were done, then *fw* would no longer be *fw*. It would be *yasifw*.

[3] Which *was* present at the time of the successful intrusions that motivated the development of *fw*

Source code for *fw* is available from the author via e-mail request.

References

[1] Beck, M., Bohme, H., Dziadzka, M., Kuntz, U., Magus, R., and Verworner, D., *Linux Kernel Internals, 2^{nd} ed.* , Addison-Wesley, 1998, pp 279–289.

[2] Geist, R. and Westall, J, "Bringing the High End to the Low End: High Performance Device Drivers for the Linux PC", *Proc. of the 36^{th} Annual ACM Southeast Conf.*, Marietta, GA, Apr. 1-3, pp. 251–260.

[3] Licqui, J. and Stearns, W., "Mason: The Automated Firewall Builder for Linux", http://users.dhp.com/ whisper/mason/

[4] Palmer, G. and Nash, A., "Firewalls", *The FreeBSD Handbook*, Chpt 8., http://www.freebsd.org/handbook/firewalls.html

[5] Ranch, D., "Linux IP Masquerade HOWTO", http://www.ibiblio.org/pub/Linux/docs/HOWTO/IP-Masquerade-HOWTO

[6] Ranum, M. and Curtin, M., "Internet Firewalls Frequently Asked Questions", http://pubweb.nfr.net/ mjr/pubs/fwfaq/

[7] Row, W., Adams, B., Morton, D., and Wright, H., "Security Issues in Small Linux Networks", *Proc. ACM Symposium on Applied Computing (SAC '99)*, San Antonio, TX, Feb 28 - March 2, 1999, pp. 506–510.

[8] Sklower, K. "A Tree-Based Packet Routing Table for Berkeley Unix", *Proc. 1991 Winter USENIX Conference*, Dallas, TX, 1991, pp 93–99.

[9] Stevens, R., *TCP/IP Illustrated, Vol 1.*, Addison-Wesley, 1994, pp. 140–141.

[10] Turner, J., "What's Wrong With ATM", *Washington University Workshop on Integration of IP and ATM*, St. Louis, MO, Nov 16, 1996, http://www.arl.wustl.edu/ jst/atmip/turner1/

[11] Vicom Technology Ltd., "Firewall Questions and Answers", http://www.vicomsoft.com/knowledge/reference/firewalls1.html