Development Framework for Firewall Processors

T.K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu and N. Dulay
Department of Computing,
Imperial College,
180 Queen's Gate,
London SW7 2BZ, England
{tkl97, sy99, w.luk, m.sloman, e.c.lupu, n.dulay}@doc.ic.ac.uk

Abstract

High-performance firewalls can benefit from the increasing size, speed and flexibility of advanced reconfigurable hardware. However, direct translation of conventional firewall rules in a router-based rule set often leads to inefficient hardware implementation. Moreover, such low-level description of firewall rules tends to be difficult to manage and to extend. We describe a framework, based on the high-level policy specification language Ponder, for capturing firewall rules as authorization policies with user-definable constraints. Our framework supports optimisations to achieve efficient utilisation of hardware resources. A pipelined firewall implementation developed using this approach running at 10MHz is capable of processing 2.5 million packets per second, which provides similar performance to a version without optimisation and is about 50 times faster than a software implementation running on a 700MHz PIII processor.

1 Introduction

Internet Protocol (IP) packet filtering is considered an effective firewall architecture, and is often used for network security [2, 12]. Packet filters control data flow between a protected network and the outside space. Each packet contains a header which gives information about the type of transport layer used, source and destination addresses, a header checksum, and some optional administrative bits. A packet filter works by checking the content of the IP packet header and then decides whether communication is allowed based on a set of rules.

Currently, most packet filters rely on processors running entirely in software. However, with the recent advance in field-programmable gate array (FPGA) technology, custom-developed hardware packet filters that outperform their software counter parts become possible [6, 7, 8, 10]. Software based packet filters suffer from increased look-up times as the number of filter rules grow. They therefore have difficulty in keeping up with the current network throughput, and may reduce network performance. Hardware, on the other hand, also has limitations; for instance, the amount of available reconfigurable resources on an FPGA can limit the number of concurrent matches in a packet filter. While some studies [6, 7, 10] focus on optimisation of the usage of hardware resources, they do not take into account the redundancy among the firewall rules in a rule set, and they did not utilize information other than those offered by the IP packet headers.

We describe how Ponder [4, 5], a policy specification language, can be used to capture an authorization policy. User-definable constraints, which are not normally found in the syntax of conventional firewall rules, is supported for conditional checking of information other than those offered by IP headers. We define policy types, which uses domain hierarchies, that map to an intermediate firewall representation. In addition, knowledge of network topology and available services within organization can assist 'don't care' discovery and rule elimination. Finally, we describe a parallel matching process by using filters to separate acceptance and rejection actions, and pipelining to achieve higher throughput.

The rest of the paper is organised as follows. Section 2 gives an overview of our development framework. Section 3 illustrates the platform-specific hardware implementation, while Section 4 provides a summary of our work.

2 Design framework

Organizational security policies restrict the acceptable behaviour on the network by controlling access to resources or services [2, 12]. Firewalls use packet filtering to implement authorization policy, whereby packets are permitted or denied according to their source or destination IP and/or port addresses. Rules are applied at the network edge for both incoming and outgoing traffic. The order-

```
access-list 104 permit tcp host 195.172.121.56 host 195.172.33.110 range 23 27
```

Figure 1 An example of a Cisco firewall rule. This rule says that any TCP protocol packet coming from IP address 195.172.121.56 destined for IP address 195.172.33.110 with destination port address in the range 23 to 27 are permitted.

ing of the firewall rules within a rule set is significant. A packet is sequentially checked against each rule, starting from the beginning of a rule set, until a match for the conditions specified in a rule is found or the end of the rule set is reached.

The syntax of the rules is firewall specific [3, 9], although a typical rule contains data fields for packet type, source and destination addresses, and action to be performed when the rule is matched – usually *PERMIT* and the *DENY* actions. Some fields may not be relevant in all rules; and they can be considered as 'don't care' during hardware implementation. An example of a Cisco firewall rule [3] is shown in Figure 1.

There are three basic design objectives of our approach, as follows.

- To use high-level programming languages for writing firewall rules, especially for managing authorization policies for a complex large-scale organizational network. Moreover, methods and mechanism in expressing and optimizing firewall rules for reconfigurable hardware are provided. We use Ponder, an existing high-level policy specification language, for producing reconfigurable hardware rather than creating an entirely new language.
- To use hardware resources efficiently to overcome physical limitation on the size of reconfigurable hardware. Our emphasis is on sharing of hardware functional units and parameterised library blocks; such as IP and port address comparators. In addition, there is scope for future extension for hardware reuse by run-time reconfiguration (RTR), and also hardware software co-operation.
- To allow multiple levels of optimization to be carried out. In particular, to allow choice of various low-level hardware specific implementation techniques.

Ponder is a declarative, object-oriented language for specifying security and management policy for distributed object systems [4, 5]. Firewall rules do not usually have constraints; however, the ability to incorporate constraints in Ponder enables conditional checking of information other than those offered by IP headers. For example, a constraint may limit the applicability of a rule to specific days or times. Ponder allows policies to be specified for a large number of objects, as well as providing abstractions for scalability.

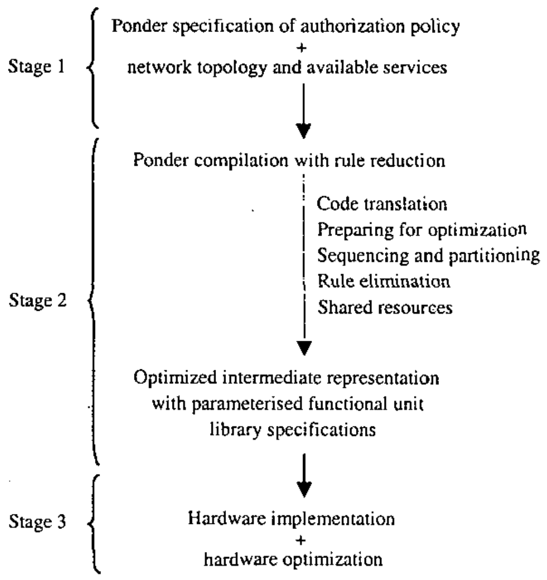
Stage 1
Ponder specification of authorization policy
+
network topology and available services
Stage 2
Ponder compilation with rule reduction
Code translation
Preparing for optimization
Sequencing and partitioning
Rule elimination
Shared resources
Optimized intermediate representation
with parameterised functional unit
library specifications
Stage 3
Hardware implementation
+
hardware optimization

Figure 2 An overview of the development steps.

An overview of the development steps of our framework for a reconfigurable hardware packet-filtering firewall is shown in Figure 2. There are three main stages in the design flow: policy capture, rule optimization, and hardware implementation.

In the first stage, an authorization policy is captured in a Ponder specification together with the information on the organization's network topology and services. In the second stage, this specification is translated into a platform independent intermediate representation. A series of operations, including construction of IP address trees, sequencing, rule elimination, and shared resources, are then performed to optimise the representation before it is taken for hardware implementation. In the third stage, this optimized representation is used to target a particular reconfigurable hardware platform. Designs of the packet filter in hardware is captured using a hardware description language. Hardware optimizations can be applied at this stage.

The advantages of this three-stage approach are twofold. First, it allows multiple levels of optimization to be performed, based on different sets of criteria and information available. In particular, it permits using platform-specific optimisations as well as software techniques. Second, it enables testing of different hardware implementation techniques on size and speed optimizations.

3 Implementation

Before the implementation of a packet filter can be carried out, the optimized intermediate firewall representation generated from the Ponder specification is transformed to a format suitable for compiling into hardware. Each firewall rule is transformed to a corresponding hardware filter rule, usually stored in a database.

Our development framework currently involves the Handel-C language [1] and the RC1000-PP [11] reconfigurable hardware development platform to produce hardware packet filters. There are several reasons for using Handel-C. First, it enables rapid and incremental development, starting from a software C description to highly-optimised pipeline implementations. Also, it is well-integrated with the RC1000-PP, a reconfigurable hardware development board containing a Xilinx Virtex XCV1000 FPGA and four memory banks; this platform has been used as an experimental vehicle for various applications [11].

A pipelined packet filter has been developed using our development framework to achieve high throughput [8]. Pipelining is a technique that can result in major performance gains, particularly for regular architectures. Instead of dealing with one packet at a time, a pipelined packet filter processes multiple packets concurrently.

An additional optimisation is to perform acceptance matching and rejection matching in parallel, as shown in Figure 3. This can be achieved after the conflicts among the rules in a rule set is resolved in the sequencing and partitioning step. The filter rules for the acceptance and rejection matching are formed as the acceptance and rejection database respectively; they are respectively stored in two of the four available RAM banks. During initialization, rules stored in the RAM banks are loaded into the corresponding registers on the FPGA. Thereafter, parallel matching of filter rules can be performed on hardware. This optimisation is particularly effective when the number of acceptance rules is similar to that of rejection rules, so that the two matching processes would take similar time.

In one of our implementations, each filter rule is implemented as a pipeline stage. Packets stream through the cascade of comparators for acceptance and rejection matching. Parallel matching is performed on all stages when the pipeline is filled. The comparison results also flow through the pipeline in synchrony with the packets. When a match

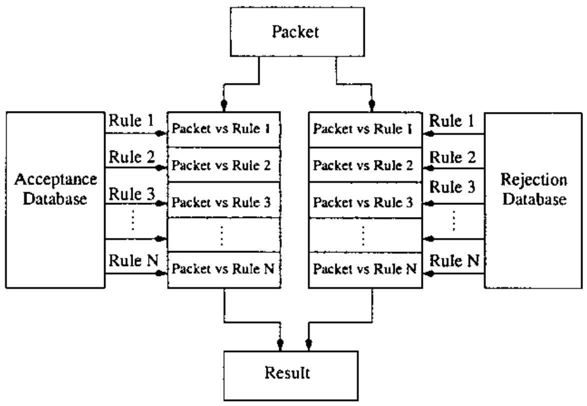

Figure 3 Parallel packet filtering.

is found, the remaining pipeline stages will not perform further comparison on that packet but deliver the result from the previous stage to the next stage.

Since our main concern is about the raw processing power of the packet filter in a firewall system, network traffic is not considered in our current work. Each pipeline stage in the above implementation takes 4 cycles to complete a comparison. When clocked at 10 MHz, the hardware packet filter has a peak throughput of 2.5 million packets per second on a Xilinx Virtex XCV1000 device, which is approximately 50 times faster than a software implementation running on a 700 MHz PIII processor. This estimate has not taken into account of latency, which varies with the number of pipeline stages and can affect the network throughput.

It is assumed that sufficient hardware resources are available to accommodate a complete set of filtering rules. Otherwise, performance will be degraded due to the fact that a filter rule set will need to be divided into multiple smaller groups. Additional time would be required for swapping in different groups from one or both of the acceptance and rejection filter-rule databases at run time. In contrast to implementations where the filter rules are embedded into the hardware [6, 7, 10], loading different groups of rules in our design does not require reconfiguring the FPGA, because our implementation uses RAM banks to store the filter-rule tables for the acceptance and rejection databases.

A hardware packet filter is limited to the number of rules that can be implemented in the hardware used. On the other hand, a software version is only limited by the amount of physical storage available for storing the rule sets. This flexibility results in increased search time: it illustrates the trade-off between a fast hardware implementation, and a slower but more flexible software implementation.

The optimized intermediate representation produced by the rule reduction mechanism can reduce the usage of hardware resources. Hence, it is possible to incorporate a larger rule set. This approach requires hardware implementation to be able to incorporate irregular structures for the filter-rule matching, while most existing pipeline structures are regular.

4 Summary

We have presented a design flow for developing hardware packet filters, adopting high-level policy specification language using domain hierarchies. It supports user definable constraints, and enables conditional checking of information other than those offered by IP headers. We have tested this method on authorization policies for both incoming and outgoing traffic. The results achieved ranging from reduction of one-third to two-thirds on rule counts.

Our pipelined packet filter implementation, running at 10 MHz, is capable of processing 2.5 million packets per second. It is approximately 50 times faster than a software implementation running on a 700 MHz PIII processor.

Current and future work includes using Ponder constraints to incorporate run-time reconfiguration and hardware software co-operation into the framework. Exploration of various hardware-level optimization techniques, such as methods based on binary decision diagram [10] and content addressable memory [6], are under investigation. The former is capable of producing a more compact representation of filter rules, while the latter is capable of fast database search on irregular structures.

Acknowledgements

The support of UK Engineering and Physical Sciences Research Council (Grant number GR/R 31409, GR/R 55931 and GR/N 66599), Celoxica Limited and Xilinx, Inc. is gratefully acknowledged.

References

[1] Celoxica Limited, *Handel-C v3.1 Language Reference Manual*, http://www.celoxica.com/.

[2] W.R. Cheswick and S.M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*, Addison-Wesley, 1994.

[3] Cisco Systems Inc., *Cisco PIX Firewall Command Reference*, http://www.cisco.com/.

[4] N. Damianou, N. Dulay, E. Lupu and M. Sloman, "The Ponder Policy Specification Language", *in Proc. Workshop on Policies for Distributed Systems and Networks*, LNCS 1995, Springer, 2001, pp. 18–39.

[5] N. Damianou, *A Policy Framework for Management of Distributed Systems*, PhD Thesis, Imperial College, 2002.

[6] J. Ditmar, K. Torkelsson and A. Jantsch, "A dynamically reconfigurable FPGA-based content addressable memory for Internet Protocol characterization", *Field Programmable Logic and Applications*, LNCS 1896, Springer, 2000.

[7] P.B. James-Roxby and D.J. Downs, "An Efficient content-addressable memory implementation using dynamic routing", *in Proc. Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2001.

[8] W. Luk, S. Yusuf and R. Nagarajan, "Incremental Development of Hardware Packet Filters", *in Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, CSREA Press, 2001, pp. 115–118.

[9] R. Russel, *Linux IPCHAINS-HOWTO*, http://www.linuxdoc.org/HOWTO/IPCHAINSHOWTO.html.

[10] R. Sinnappan and S. Hazelhurst, "A Reconfigurable Approach to Packet Filtering", *Field Programmable Logic and Applications*, LNCS 2147, Springer, 2001.

[11] H. Styles and W. Luk, "Customising graphics applications: techniques and programming interface", *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, pp. 77–87, 2000.

[12] E.D. Zwicky, S. Cooper and D.B. Chapman, *Building Internet Firewalls*, Second Edition, O'Reilly & Associates, 2000.