Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2003-56

2003-07-29

FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet

James Moscola

As computer networks increase in speed, it becomes difficult to monitor and manage the transmitted digital content. To alleviate these problems, hardware-based search (FPgrep) and search-and-replace (FPsed) modules have been developed. FP-grep has the ability to scan packet payloads for a given set of regular expressions and pass or drop packets based on the payload contents. FPsed also scans packet payloads for a set of regular expressions and adds the ability to modify the payload if desired. The hardware circuits that implement the FPgrep and FPsed modules can be generated, compiled, and synthesized using a simple web interface. Once... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Moscola, James, "FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet" Report Number: WUCSE-2003-56 (2003). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1101

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1101

FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet

James Moscola

Complete Abstract:

As computer networks increase in speed, it becomes difficult to monitor and manage the transmitted digital content. To alleviate these problems, hardware-based search (FPgrep) and search-and-replace (FPsed) modules have been developed. FP-grep has the ability to scan packet payloads for a given set of regular expressions and pass or drop packets based on the payload contents. FPsed also scans packet payloads for a set of regular expressions and adds the ability to modify the payload if desired. The hardware circuits that implement the FPgrep and FPsed modules can be generated, compiled, and synthesized using a simple web interface. Once a module is created it is programmed into logic on a Field Programmable Gate Array (FPGA). The FPgrep and FPsed modules use FPGAs to process packets at the full rate of Gigabit-speed networks. Both modules, along with several supporting applications were developed and tested using the Field Programmable Port Extender (FPX) platform. Applications developed for the modules currently include a spam filter, virus protection, an information security filter, as well as a copyright enforcement function.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-56

FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet - Master's Thesis, August 2003

Authors: Moscola, James

July 29, 2003

Abstract: As computer networks increase in speed, it becomes difficult to monitor and manage the transmitted digital content. To alleviate these problems, hardware-based search (FPgrep) and search-and-replace (FPsed) modules have been developed. FPgrep has the ability to scan packet payloads for a given set of regular expressions and pass or drop packets based on the payload contents. FPsed also scans packet payloads for a set of regular expressions and adds the ability to modify the payload if desired. The hardware circuits that implement the FPgrep and FPsed modules can be generated, compiled, and synthesized using a simple web interface. Once a module is created it is programmed into logic on a Field Programmable Gate Array (FPGA). The FPgrep and FPsed modules use FPGAs to process packets at the full rate of Gigabit-speed networks. Both modules, along with several
supporting applications were developed and tested using the Field Programmable Port Extender (FPX) platform. Applications developed for the modules currently include a spam filter, virus protection, an information security filter, as well as a copyright enforcement function.

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

SEVER INSTITUTE OF TECHNOLOGY

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: July 29, 2003

STUDENT'S NAME: James Moscola

This student's thesis, entitled FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet has been examined by the undersigned committee of four faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: ______________________ Chairman

____

____

____

Short Title: Managing Digital Content Moscola, M.Sc. 2003

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FPGREP AND FPSED: PACKET PAYLOAD PROCESSORS FOR MANAGING THE FLOW OF DIGITAL CONTENT ON LOCAL AREA NETWORKS AND THE INTERNET

by

James Moscola

Prepared under the direction of Dr. John Lockwood

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

August, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

FPGREP AND FPSED: PACKET PAYLOAD PROCESSORS FOR MANAGING THE FLOW OF DIGITAL CONTENT ON LOCAL AREA NETWORKS AND THE INTERNET

by James Moscola

ADVISOR: Dr. John Lockwood

August, 2003
Saint Louis, Missouri

As computer networks increase in speed, it becomes difficult to monitor and manage the transmitted digital content. To alleviate these problems, hardware-based search (FPgrep) and search-and-replace (FPsed) modules have been developed. FPgrep has the ability to scan packet payloads for a given set of regular expressions and pass or drop packets based on the payload contents. FPsed also scans packet payloads for a set of regular expressions and adds the ability to modify the payload if desired. The hardware circuits that implement the FPgrep and FPsed modules can be generated, compiled, and synthesized using a simple web interface. Once a module is created it is programmed into logic on a Field Programmable Gate Array

(FPGA). The FPgrep and FPsed modules use FPGAs to process packets at the full rate of Gigabit-speed networks. Both modules, along with several supporting applications were developed and tested using the Field Programmable Port Extender (FPX) platform. Applications developed for the modules currently include a spam filter, virus protection, an information security filter, as well as a copyright enforcement function.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank all the people who have make this work possible. Firstly, I would like to thank my advisor John Lockwood for giving me the opportunity to work on this project and his never-ending support.

Next, I would like to thank Dr. Ronald Loui for all of his support and ideas on the project as well as for teaching me the joy of GAWK.

I would also like to thank all the people in the FPX group (past, present, and future) for their contribution to the FPX platform. Without their work, this project would not have been possible. I would especially like to thank Todd Sproull for his continuous help in the lab debugging and for his work on the NCHARGE control software. For all of his hard work on the NID, I would like to thank David Lim.

For his initial work (and continuing support) on this project, and for being a great friend I would like to thank Michael Pachos. Without Michael I would not have been involved with this work.

For financial support, I would like to thank my research sponsor Global Velocity and all those involved in the company. For without them I would have been evicted long ago. In particular, I would like to thank Matthew Kulig, David Reddick, and Tim Brooks for their unending interest and support on the project.

Finally, I would like to thank my wife Stephanie for being patient and supportive throughout my lengthy education. Without her motivation this thesis may have never been come to completion.

James Moscola

Washington University in Saint Louis
August 2003

Chapter 1

Introduction

1.1 Motivation

With the ever-increasing popularity of the Internet, it is becoming increasingly more difficult to manage digital content and to ensure the safety of machines connected to a network. On a daily basis, email accounts are flooded with spam, network machines are infected with viruses, servers are crashed by Denial of Service (DOS) attacks, company secrets are leaked, and millions of files are traded illegally. The Internet has truly become a dangerous place from both a user perspective as well as a corporate perspective.

Firewalls and Network Intrusion Detection Systems (NIDS) attempt to protect networks from the perils of the Internet. They provide protection for Local Area Networks (LANs) by enforcing Access Control Policies (ACPs) for both incoming and outgoing traffic and by alerting a system administrator if any of these policies are broken. Currently, the scope of most ACPs only covers packet headers. Some firewalls and NIDS also support ACPs that do exact string matching within the packet payload. However, this is usually done in software making it too slow and inefficient for modern network backbones. Current firewalls and NIDS also do nothing to protect against the widespread distribution of illegal digital content.

By giving firewalls and NIDS the ability to perform regular expression (RE) matching in hardware, instead of software, the power of these systems can be greatly enhanced. With regular expressions, a single ACP is capable of enforcing rules which previously took multiple ACPs to enforce (just as one IP address/netmask pair can be used to specify multiple IP addresses). More importantly, regular expressions can give ACPs the ability to enforce rules on mutable content such as that found in many Denial Of Service (DOS) attacks and viruses. By doing all of the RE processing in hardware, every byte of every packet can be scanned while still maintaining the speed and throughput required for very high speed network backbones.

This thesis describes the implementation of hardware-based search (FPgrep) and search-and-replace (FPsed) modules that can be used to augment the capabilities of modern firewalls and NIDS. To achieve the performance required by today's high-performance networks, hardware devices known as Field Programmable Gate Arrays (FPGAs) are employed. FPGAs offer a method for implementing functions in hardware using reprogrammable chips. In addition, FPgrep and FPsed utilize regular expressions to provide a powerful method for specifying a single string pattern or set of string patterns that may be searched for within the payload of a packet as it passes through a network. The regular expressions can range in complexity from a simple single character string to a string consisting of multiple wildcards.

By combining the power of regular expressions and the flexibility of FPGAs on the Field Programmable Port Extender (FPX) [14, 15], the FPgrep and FPsed packet payload processors may be used to process packet contents within a Local Area Network or even over the Internet.

1.2 Thesis Outline

The thesis begins with a brief history on the methods used for string matching, and string matching in hardware. This is followed by a description of the Field Programmable Port Extender platform and a set of layered Protocol Wrappers used for the hardware modules. Next is a description and results for both the FPgrep and FPsed hardware modules. After describing the modules, there is a description on how they can be automatically generated and programmed into the FPX platform. Finally, different applications for the technology are discussed and how each is implemented.

Chapter 2

String Matching

2.1 Background on String Matching

Both string matching and regular expression matching have been studied extensively in the past. Prior to the late 1970's, string searching was accomplished using a brute force method. The brute force method assumes that a pattern of length m may occur at any position in a string of length n, and then sets about testing each possible orientation. The brute force method's worst case running time is $O(nm)$. The major problem with the brute force search is that characters in the text may be re-examined multiple times. In some cases, this can lead to poor performance. In 1977 both Knuth, Morris and Pratt (KMP) [13], and Boyer and Moore (BM) [2] developed efficient approaches to string searching. The KMP algorithm provides a way to eliminate repeated accesses to the text and delivers a guaranteed linear time searching algorithm. By using previous character information, the KMP algorithm can access the characters sequentially without any need to back up and re-read portions of the input. The BM algorithm uses an approach similar to that of the KMP algorithm, but instead searches text from right-to-left. This eliminates the need for the algorithm to examine a great deal of the text on a mismatch. Both the KMP and the BM algorithms have a worst case running time of $O(n + m)$.

Thompson [29] developed the classical approach for searching for a regular expression. Thompson's idea involves converting a regular expression into a Nondeterministic Finite Automaton (NFA) with $O(m)$ nodes. The worst case running time of the search is $O(nm)$ because the machine may be in more than one state at any given time. Consequently, there may be situations in which *all* states require a transition to a new state resulting in a slow and inefficient method of searching. A more efficient technique (in terms of running time) is to convert the NFA into a Deterministic Finite Automaton (DFA). This approach yields a worst case running time of $O(n)$. A drawback to this approach is that, theoretically, there can be up to $O(2^m)$ states, thereby requiring a great deal of memory or hardware for implementation.

2.1.1 Regular Expressions

A regular expression is a pattern that describes a set of strings. The basic building blocks for these patterns consist of individual characters that match themselves such as "a", "b", and "c". Combining characters with meta-characters ($*, |, ?$) allows more complex regular expressions to be created. If r_1 and r_2 are regular expressions then r_1* matches any string composed of zero or more occurrences of r_1; $r_1?$ matches any string composed of zero or one occurrences of r_1; $r_1|r_2$ matches any string composed of r_1 or r_2; and r_1r_2 matches any string composed of r_1 concatenated with r_2. For instance, "a" is a regular expression that denotes the singleton set {"a"}, while "$a|b$" denotes the set {"a","b"}. The expression "$a*$" denotes the infinite set {"","a","aa","aaa",...}.

Regular Expression Terminology

Before beginning a more in-depth discussion, some of the terminology used in this thesis to describe the matching of regular expressions is defined below.

> ***Start:*** The transition of a state machine from the *idle* (initial) state to a non-*idle* state.
>
> ***Accept:*** The state machine has *accepted* the substring if the state machine has determined that the substring is a member of the language defined by the RE. For instance, if the regular expression that is being searched for is "*abc?*", then the language defined by this expression is the set {"*ab*", "*abc*"}. For this language the state machine will *accept* when it detects the substring *"ab"*. This substring is a member of the set defined by the regular expression. However, the state machine cannot be sure that this is the longest possible *match* until it sees the next character. If the next character in the substring is "*c*", then the state machine will now *accept* the substring "*abc*" as part of the language. If the next character in the substring is not "*c*", then the state machine will continue to *accept* the substring "*ab*" as part of the language. Once a state machine *accepts* a substring it must *match* on that substring some time in the future.
>
> ***Match:*** The state machine has determined the boundaries of the complete substring. Using the example above, the state machine would *match* on the third character independent of what the character is. If the third character is a "*c*", then the substring "*abc*" *matches.* If the third character is something other than "*c*", then the substring "*ab*" *matches.*

Running: The state machine has *started* but not yet *failed*. The state machine may or may not be in an *accepting* state.

Reset/Fail: The state machine was *running* and a character read caused the substring to no longer be a member of the language defined by the RE. If the substring was previously *accepted* then a *match* was created over a portion of the substring up to, but not including the character that caused the state machine to *reset*.

Idle: The state machine is not *running*. A character has not yet been read that defines the beginning of the language that is being searched for.

2.2 String Matching in Hardware

The idea of string matching in hardware is a topic that has been explored for over two decades. Haskin and Hollar [8] offer a comprehensive (but dated) survey on hardware-based regular expression matchers. The authors discuss various techniques and offer a novel solution of their own. They defined three different classes of approaches: parallel comparators, cellular comparators, and finite state machines. Both the parallel and cellular comparator techniques have substantial drawbacks. The parallel technique can only handle a fixed number of terms of a fixed maximum size. The cellular technique can require extensive logic to implement. The last technique, the use of a finite state machine, is capable of handling exact character and character class matching. Haskin discusses a technique of simulating an NFA by replicating a number of DFAs where an *idle* DFA *starts* every time the possible beginning of a term is recognized.

More recent work in the area of string matching on FPGAs has been done by Sidhu and Prasanna [22] as well as by Franklin, Carver, and Hutchings [7]. The work by Sidhu and Prasanna was primarily concerned with minimizing the time and space required to construct NFAs because they run their NFA construction algorithm in hardware as opposed to software. Their work yielded an exceptional approach to string matching in hardware. Franklin, Carver, and Hutchings followed with an analysis of this approach for the large set of expressions found in a Snort database [18].

2.2.1 Nondeterministic vs Deterministic Finite Automata

In other work, NFAs were chosen due to the shorter time and smaller space required for *constructing* the automata [22]. In contrast, this work was not concerned with the time and space required for constructing the automata. It was however, concerned with the size of the completed automata. Theoretically, DFAs can contain up to $O(2^n)$ states, where n is the number of characters in the expression. However, in practice it was found that the number of states required is most often less than or equal to n. In addition to this, DFAs can only have one active state and thus can be represented more compactly than NFAs if the state of the search needs to be stored.

To verify that DFAs are generally more compact with real data, spam-matching rules were extracted from the current version (2.60) of the *SpamAssassin* [23] program. State machines were then generated from the 358 regular expressions found in this database using a Java-based lexical analyzer generator called JLex [1]. As input, JLex takes a regular expression. It outputs a Java file that contains state and transition tables for a minimized DFA that implements the given expression. The JLex tool was run to generate an NFA and a minimal DFA from each of the regular expressions. It was found that most of the expressions created DFAs that were optimized to contain fewer states than the NFA. Figure 2.1 shows the ratio of the number of states in the JLex-optimized DFA to the number of states in the NFA. Note that where the x-axis equals 1 is the point where the DFA and the NFA are equal in size. To the left of this point represents DFAs that are smaller than the corresponding NFAs and to the right represents DFAs that are larger.

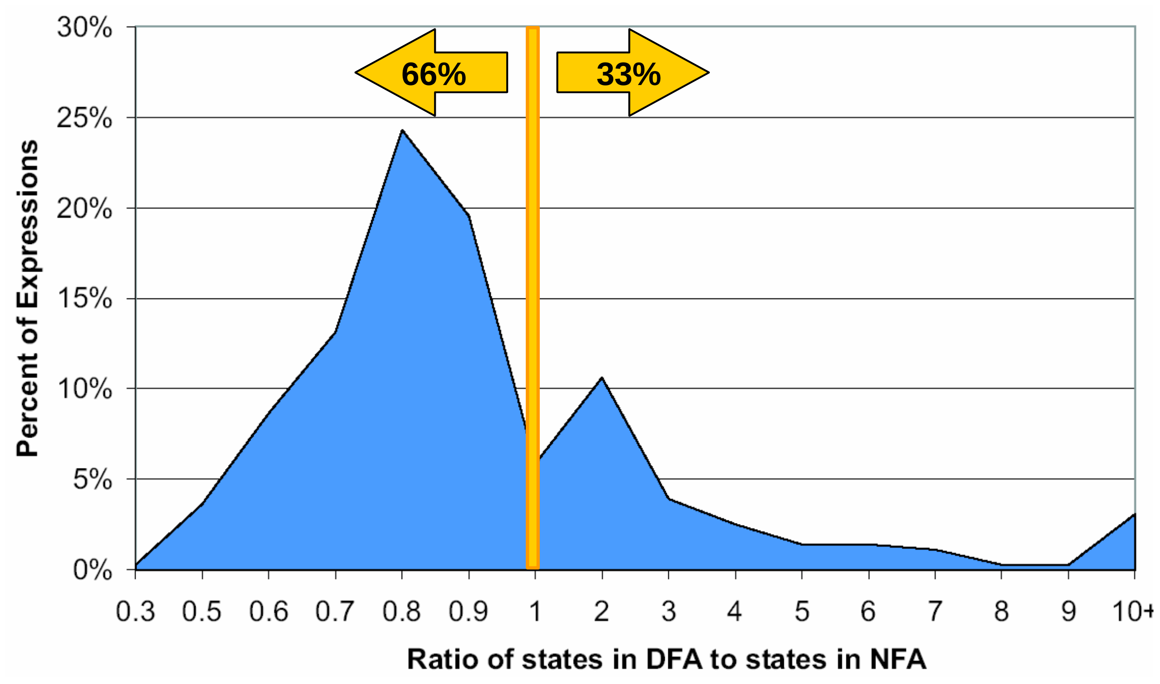


Figure 2.1: The ratio of the size of the DFA to the NFA for all REs used by the *SpamAssassin* program. Note that the majority of the DFAs optimize to be smaller than the NFAs

Through this experiment it was found that about two-thirds, or 66% of the DFAs were smaller than the NFAs. Only 2.5% of the expressions had more than a 10x expansion and only 0.5% went beyond 40x.

A typical *SpamAssassin* expression used in this experiment is:

```
U\.?S\.?(D\.?)?[\    ]*(\$[\     ]*)?([0-9]+,[0-9]+,[0-9]+
|[0-9]+\.[0-9]+\.[0-9]+|[0-9]+(\.[0-9]+)?[\    ]*milli?on)
```

The output of JLex for this particular expression is:

```
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 78 states.
Working on character classes.:
::.:.::.::..........:::..:::
NFA has 15 distinct character classes.
Creating DFA transition table.
Working on DFA states...........................
Minimizing DFA transition table.
24 states after removal of redundant states.
Outputting lexical analyzer code.
```

From the above output, it can be seen that the NFA for this expression contains 78 distinct states whereas the DFA only contains 24. This represents a DFA to NFA ratio of .3 to 1 [16].

Chapter 3

The Field Programmable Port Extender and Protocol Wrappers

3.1 Field Programmable Port Extender (FPX)

The FPX is a reprogrammable logic device that provides a hardware platform for the user to deploy packet processing network modules [14, 15]. It can act as an interface between the line cards and the WUGS (Washington University Gigabit Switch) [5] as shown in Figure 3.1. It can also be used in a stand-alone configuration. The FPX is composed of two FPGAs: the Network Interface Device (NID) and the Reprogrammable Application Device (RAD) [27].

3.1.1 Network Interface Device (NID)

The NID controls how packet flows are routed to and from hardware modules programmed into the RAD. It also provides mechanisms to dynamically load hardware modules over the network. The combination of these features allows these modules to be dynamically loaded and unloaded without affecting the switching of other traffic flows or the processing of packets by other modules in the system. As show in Figure 3.2, the NID has several components, all of which are implemented in FPGA hardware. It contains a four-port switch to transfer data between ports; Virtual Circuit lookup tables (VC) on each port in order to selectively route flows; a Control Cell Processor (CCP), which is used to process control cells that are transmitted and received over the network; logic to reprogram the FPGA hardware on the RAD; and synchronous and asynchronous interfaces to the four network ports that surround the NID.

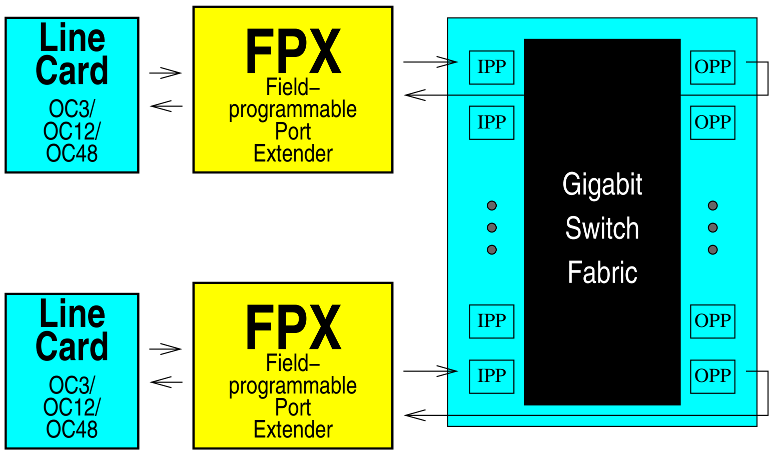


Figure 3.1: Configuration for the WUGS, FPX, and the line cards

3.1.2 FPX Reprogramability

The RAD can be programmed and reprogrammed to hold user-defined network modules [24], and is connected to two SRAM and two SDRAM components (Figure 3.2). In order to reprogram the RAD over the network, the NID implements a reliable protocol that fills the contents of the on-board RAM with configuration data that are transmitted over the network. As each cell arrives, the NID uses the data and the sequence number in the cell to write data into the RAD Program SRAM. Once the

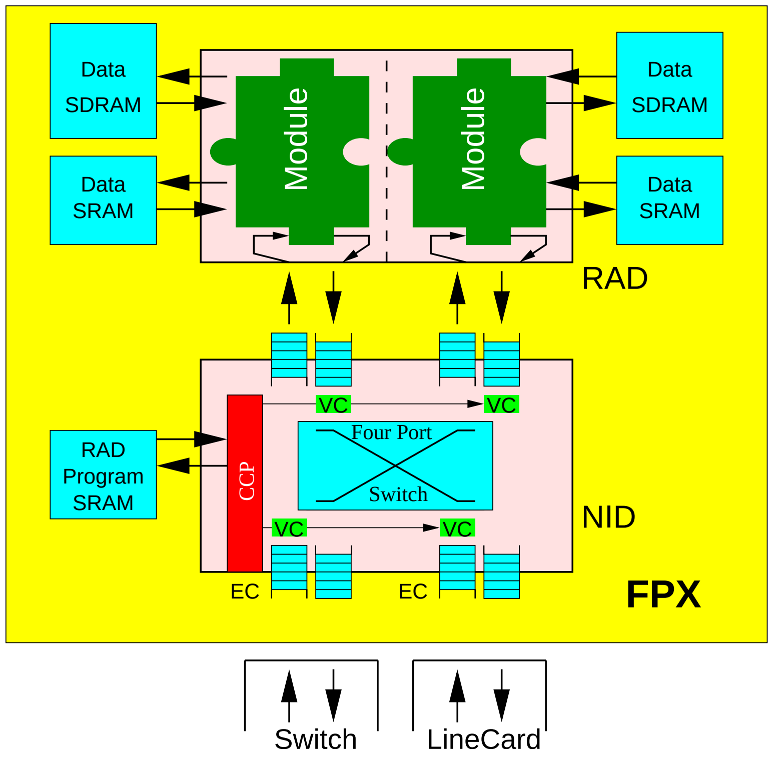


Figure 3.2: Major components of the FPX

last cell has been correctly received, the FPX holds an image of the reconfiguration bitstream that is needed to reprogram the RAD. At that time, another control cell can be sent to the NID to initiate the reprogramming of the RAD using the contents of the RAD Program SRAM.

The FPX supports partial reprogramming of the RAD by allowing configuration streams to contain commands that only program a portion of the logic on the RAD [9, 10]. Rather than issue a command to reinitialize the device, the NID writes the frames of reconfiguration data to the RAD's reprogramming port. This feature enables the other modules on the RAD to continue processing packets during the partial reconfiguration. Similar techniques have been implemented in other systems using software-based controllers [30].

3.2 Protocol Wrappers

Protocol Wrappers [3, 4] are used in the FPgrep and FPsed modules to streamline and simplify the networking functions to process ATM cells, AAL5 frames and IP packets directly in hardware. They use a layered design and consist of different processing circuits within each layer. The block diagram of the Protocol Wrappers is shown in Figure 3.3. At the lowest level, the Cell Processor processes raw ATM cells between network interfaces. At the higher levels, the Frame Processor processes variable length AAL5 frames while the IP Processor processes IP packets. Additionally, there is a UDP Processor that transmits and receives UDP messages, but this level was not required for the FPgrep and FPsed modules.

Different layers of abstraction are important for structuring the functions of a network because doing so allows relevant details to be exposed and irrelevant details to be hidden. In this manner, an application that interacts with IP packets can effectively interface just with the IP the Protocol Wrapper.

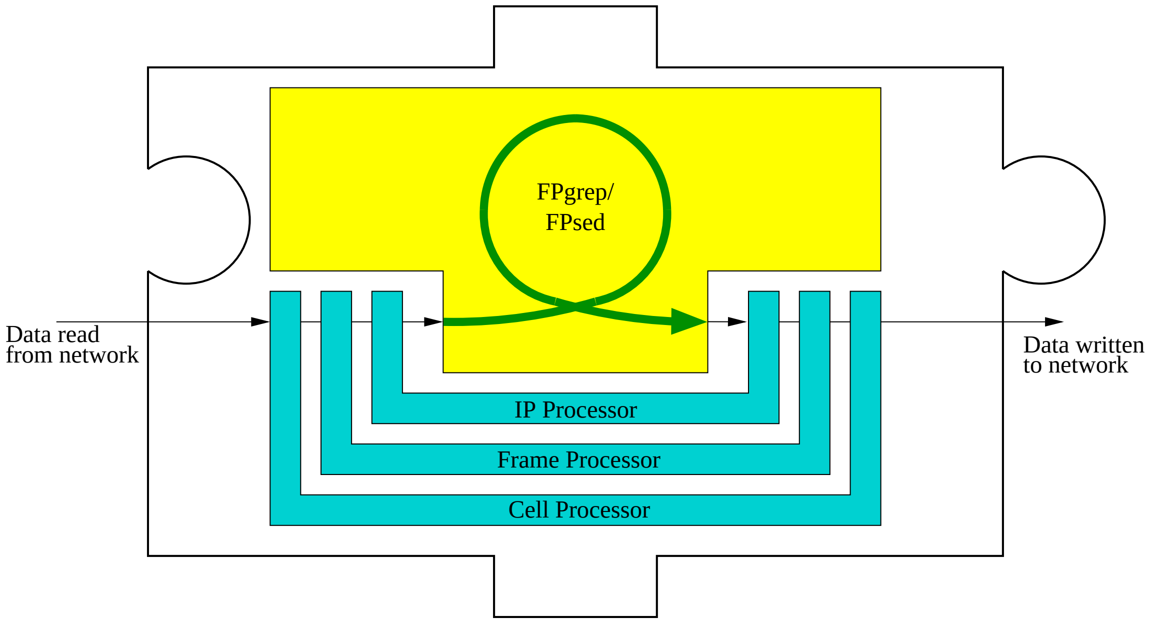


Figure 3.3: Block diagram of FPGrep/FPSed module in the Protocol Wrappers

3.2.1 Required Modifications to the Protocol Wrappers

The original Protocol Wrappers, while feature rich, were lacking in certain features required to process traffic on a real network. Below is a description of several modification made to the Protocol Wrappers to and why each was required.

The most important modification to the Protocol Wrappers was the ability to handle any frame and IP packet size. The original Protocol Wrappers were only able to handle frames and packets that were of modulus four in byte length. This shortcoming meant that the Protocol Wrappers were only capable of handling 25% of real network traffic. The other 75% of the traffic would either be corrupted or just dropped by the wrappers. To fix this issue, the last two bits of the AAL5 length field, and the last two bits of the IP length field were used to determine exactly how many bytes needed to be processed.

Additionally, the original Protocol Wrappers were only capable of processing IP packets that were encapsulated in a frame with the same byte length. This is because the IP Processor did not actually check the IP length field when receiving packets. Instead the IP Processor just assumed that any data that was received from the Frame Processor was valid IP data. In many cases, this shortcut would be harmless. However, in the lab it was found that commercial routers often add additional data to a frame in lieu of padding. This “garbage” data is likely there to give the AAL5 CRC-32 more robustness. The simple fix to this problem was to modify the IP Processor to actually check the IP length field and only process the appropriate amount of data.

Another problem of the original Protocol Wrappers was that all frames attempted to traverse through all layers of the wrappers, and the IP Processor would drop any packets that were not IPv4 packets. This means that frames that encapsulated ARP packets, DHCP packets, or any other non-IPv4 packet passed from the Frame Processor to the IP Processor and would be dropped. This behavior was unacceptable because real networks have many different types of packets that are required for efficient communication. In order to handle this problem, the design of the IP Processor was modified to bypass non-IPv4 packets around the user application as opposed to dropping them. This approach required adding an additional buffer to the output side of the IP Processor.

The next change that was required was to deal with different types of encapsulation methods on networks. The original IP Processor was only capable of processing packets that were embedded directly into a frame without any additional encapsulation headers, such as an LLC/SNAP or a MAC header. This limited the utility of the IP Processor on real network traffic because most routers use some type of packet encapsulation. The solution to this was to add a new control signal to augment the original three: *start-of-frame*, *start-of-payload*, and *end-of-frame* signals. The new control signal, *start-of-IP-header*, allows the IP wrapper to pass headers that arrive after the *start-of-frame* signal, but before the IP header itself.

Other changes to the Protocol Wrappers include removing a great deal of unused logic from the Cell Processor as well as addressing several bugs related to frame size and dropped frames in the Frame Processor.

Chapter 4

FPgrep: Packet Payload Scanning Using Regular Expressions

The FPgrep content scanner is a hardware module that is capable of searching network packet payloads for a list of regular expressions. The scanner was implemented as a module on the FPX platform. The scanner utilizes the Protocol Wrappers to reassemble cells into IP packets and to delineate the header and payload fields. When designing the content scanner, four initial behaviors were desired: (1) the ability to scan every byte of every packet's payload for a given set of expressions, (2) the ability to actively drop packets that match a given expression, (3) the ability to generate an alert packet identifying which expressions in the given set match, and (4) the ability to send an alert packet to a log server when a match is detected. This chapter describes how the hardware module accomplishes each of these tasks.

4.1 Searching

FPgrep searches the input for substrings that belong to the language defined by the regular expression. When FPgrep *matches* a substring in a packet it has the ability to transmit information about the packet to a monitoring host system. The information sent can include the sender's and receiver's IP addresses and/or any other data transmitted over the connection.

The search runs in linear time (proportional to packet size) $O(n)$ and in constant space. That is, there is never a need to examine a character more than once and the amount of hardware is proportional to the size of the regular expression. Approximately one flip-flop is required per character.

When a regular expression search is requested, a ".*" is prepended to the beginning of the original regular expression. It is natural to think about it this way since searching involves finding any number of characters followed by a *matching* substring. The prepended ".*" allows the machine to recognize a *matching* substring anywhere in the record [12].

If the ".*" is not prepended, then there are situations in which a substring that should be *matched* by the machine is missed. This situation arises if a machine M enters the *running* state when it encounters character c_i and then transitions to the *failed/reset* state when it encounters c_{i+n}. If the machine simply continues reading beginning with the next character, c_{i+n+1}, it would not detect a substring whose first character is in the range c_{i+1} to c_{i+n}.

A small example illustrates this problem. Assume "*ARL*" is being searched for. If both "*ARL*" and ".**ARL*" are converted into DFAs, two functionally different machines (DFA_1 and DFA_2 respectively) are produced. These DFAs can be seen in Figure 4.1 and Figure 4.2.

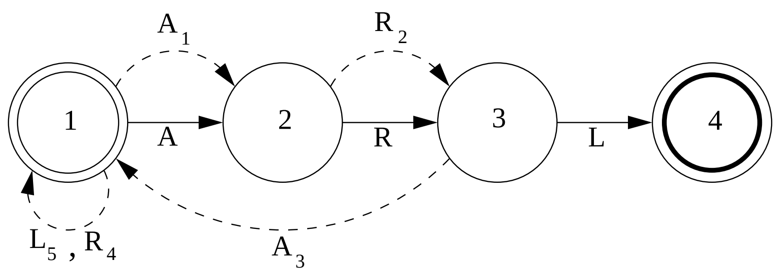


Figure 4.1: DFA_1 for "ARL" does not reach the *accept* state

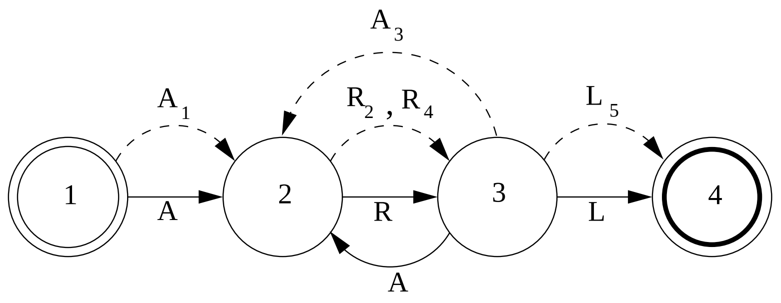


Figure 4.2: DFA_2 for ".*ARL" does reach the *accept* state

When the input to the machines is "$A_1R_2A_3R_4L_5$", DFA_1 will recognize that "A_1" followed by "R_2" is part of the language. When "A_3" is input, the machine *fails* and thus transitions to the *idle* state. It is clear that machine will not find the substring "$A_3R_4L_5$", since when the next character "R_4" is input into DFA_1 it remains in the *idle* state. On the other hand DFA_2 does operate correctly and finds the substring.

4.2 FPgrep Hardware Implementation

The FPgrep hardware module has been implemented to perform a complete payload scan of every packet that passes through it. The module uses the previously mentioned technique of prepending a ".*" to each regular expression to aid in the search. A block diagram of the implementation can be seen in Figure 4.3.

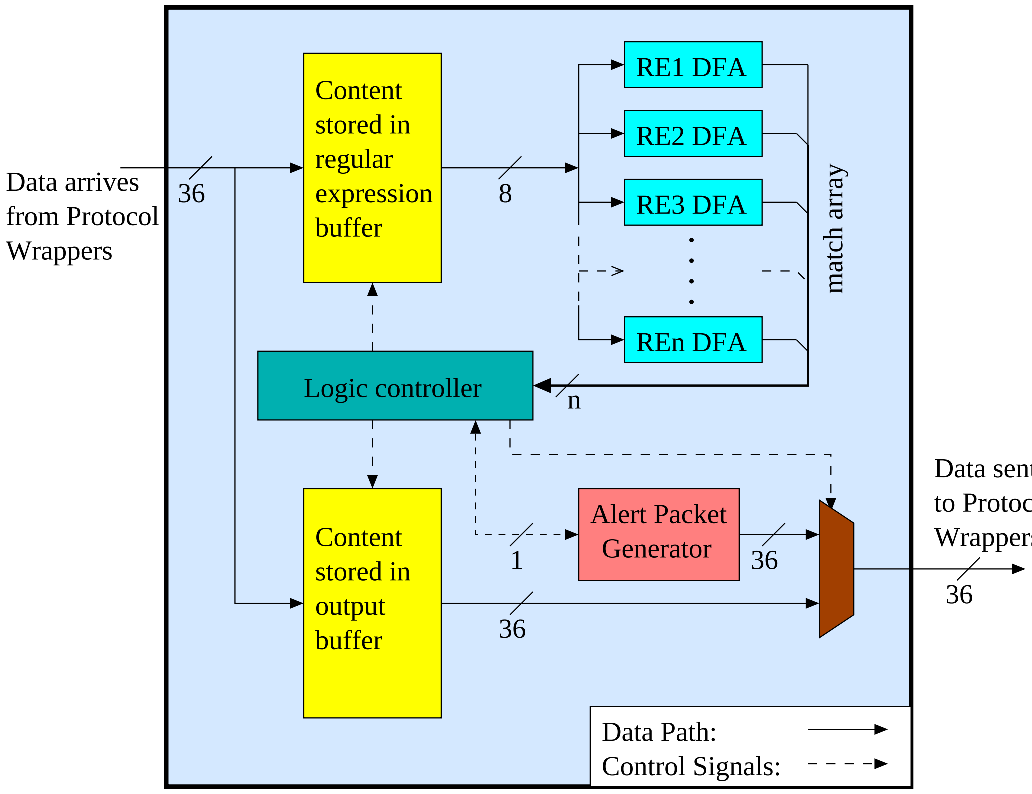

Data arrives from Protocol Wrappers
36
Content stored in regular expression buffer
8
RE1 DFA
RE2 DFA
RE3 DFA
REn DFA
match array
Logic controller
n
Content stored in output buffer
1
Alert Packet Generator
36
36
Data sent to Protocol Wrappers
36
Data Path:
Control Signals:

Figure 4.3: Block diagram of FPgrep content-scanning module

4.2.1 Logic Controller

The logic controller performs most of the work. It controls the reading and writing of the memory buffers, the data that is sent to the regular expression state machines, and the alert packet generator. The controller has three main operations: (1) *Receiving Packets*, (2) *Processing Packets*, and (3) *Outputting Packets*. Each of these three operations is controlled independently of the other two. All three operations run in parallel.

Receiving Packets

Packets enter the module in 32-bit chunks after passing through the Protocol Wrappers. The Protocol Wrappers assert control signals to indicate the beginning of a frame, the beginning of IP packet headers, the beginning of an IP packet payload, and the end of a frame. There is also a data enable signal to indicate the presence of a valid 32-bit data word on the incoming bus. Every valid data word, along with the four control signals, is written to two parallel 512x36 dual-port memory buffers. By using two identical buffers, it is possible to read newer packets for processing while older packets (that are not being dropped) are read for output. This could be achieved with a single tri-port memory buffer if available.

Processing Packets

Once data are available in the input buffer (a packet has started arriving), the module can begin processing the packet (Figure 4.4). To process a packet with the content scanner, a counter is used to address one of the 512x36 memory buffers. On each clock tick, one byte is read from the memory buffer and sent to each of the regular expression DFAs. All of the DFAs search in parallel. Each DFA maintains a 1-bit *match* signal that is asserted high when a match is found within the packet that is being processed. When the counter reaches the end of the packet, one or more of the following can occur:

1. If the *match* signals from *all* of the DFAs indicate no match was found, then a pointer to the packet is inserted into an *output queue*.
2. If any of the *match* signals indicate a match was found but do not require dropping the packet, then a pointer to the packet is inserted into an *output queue*.
3. If one or more of the *match* signals indicates a match was found that requires dropping the packet, then a pointer to the packet is *not* inserted into an *output queue*, hence the packet is dropped.

4. If one or more of the *match* signals indicates a match was found that requires an alert packet to be sent, then a pointer to the original packet is inserted into an *output queue*. This enables the alert packet to use header information from the original packet. Also, a special pointer is inserted into a *match queue* to indicates an alert packet should be output. The special pointer contains a bit array that indicates which DFAs found a match. It should be noted that if a match is found that requires an alert packet but does not require dropping the packet, two pointers (one for the original packet and one for the alert packet) are inserted into the *output queue*.

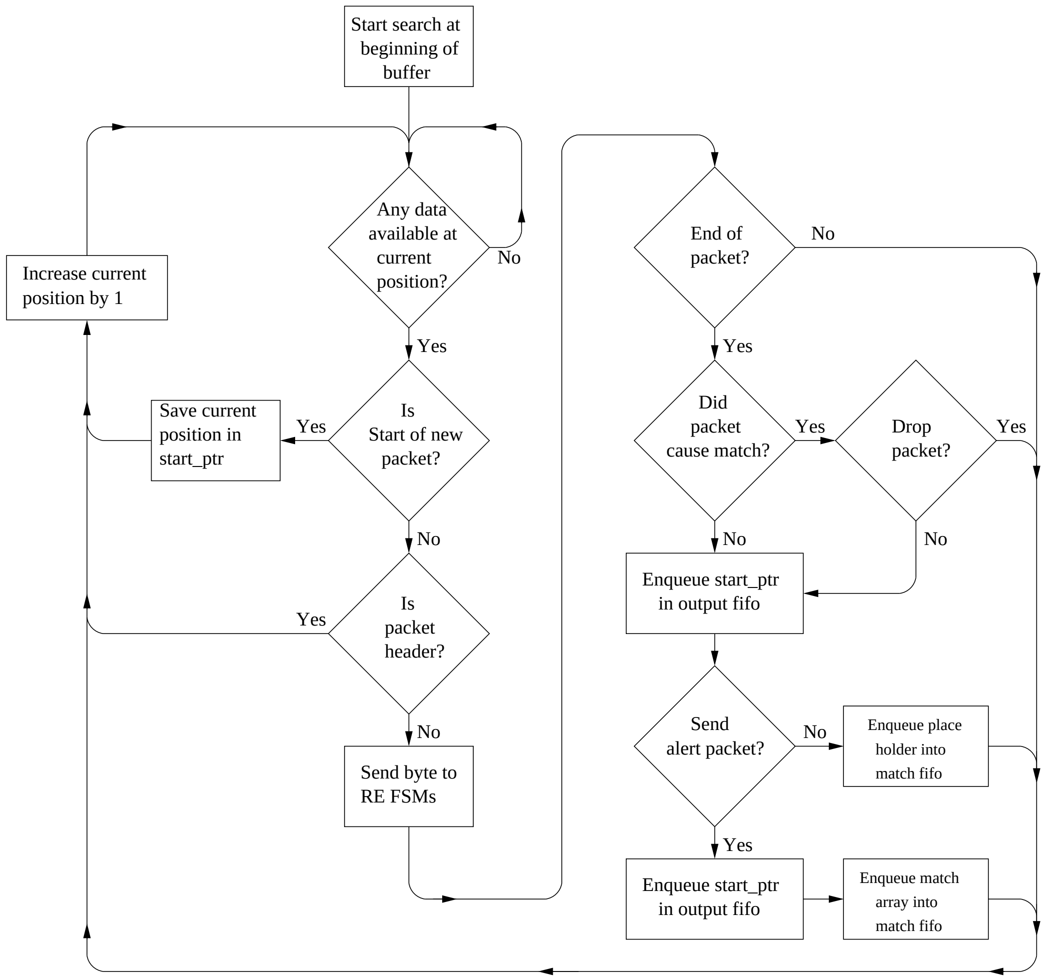

Start search at beginning of buffer
Any data available at current position?
No
Yes
Increase current position by 1
Is Start of new packet?
Yes
Save current position in start_ptr
No
Is packet header?
Yes
No
Send byte to RE FSMs
End of packet?
No
Yes
Did packet cause match?
Yes
Drop packet?
Yes
No
No
Enqueue start_ptr in output fifo
Send alert packet?
No
Enqueue place holder into match fifo
Yes
Enqueue start_ptr in output fifo
Enqueue match array into match fifo

Figure 4.4: FPgrep flow diagram for the input/searching process

Outputting Packets

A packet is output from the content scanner whenever there is an available pointer in the *output queue*. Each pointer that is dequeued can be either for a regular packet or for an alert packet. This is determined by dequeuing a pointer from the *match queue*. If the pointer from the *match queue* is all zeros, or null, then a regular packet should be output. If the pointer from the *match queue* is not null, then an alert packet should be output.

In the case of a regular packet, a counter is assigned the value of the pointer and used to address the 512x36 output memory. The packet is then output 32-bits per clock cycle until the end of the packet is detected. The most significant 4 bits of the output memory are used to recreate the necessary control signals for communicating with the Protocol Wrappers.

When an alert packet pointer is dequeued, a UDP alert packet has to be generated since one does not already exist. The alert packet can be addressed to a predetermined log server (specified at compile time but also runtime reconfigurable) or to the destination of the original packet that caused the alert packet to be generated. The payload of the alert packet contains the source and destination IP address of the packet that caused the alert, an associated identification number for *all* of the regular expressions that matched in the original packet, along with several other fields. More details on the format of the alert packet can be seen section A.1 in Appendix A.

4.3 Increasing Throughput via Parallel Scanners

As mentioned in the sections *Receiving Packets* and *Processing Packets*, packets enter the content scanner at a rate of 32-bits per clock cycle, and are processed at 8-bits per clock cycle. As a result, the content scanner can only process data at one-quarter of the maximum input rate. In order to process data at the full input rate, four parallel content scanners are arranged as shown in Figure 4.5. Arriving packets are dispatched to an available content scanner in a round-robin fashion. With four parallel scanners, the module is now capable of scanning 32-bits per clock cycle.

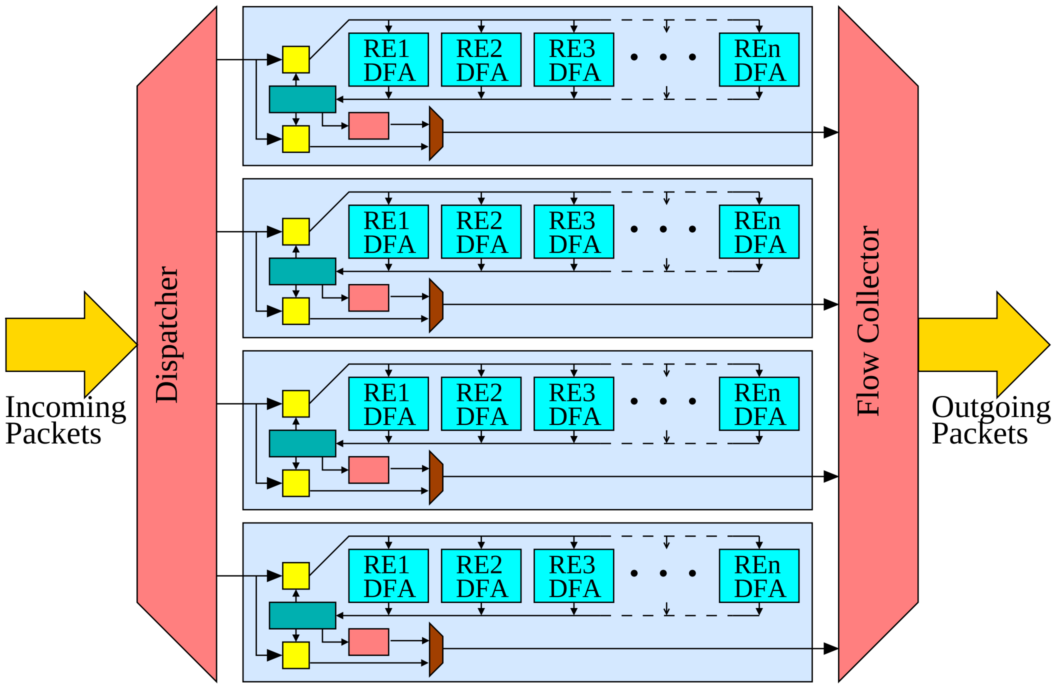


Figure 4.5: Arrangement of four parallel FPgrep scanners

4.4 Results

Several different versions of the content scanner were synthesized with the Protocol Wrappers into the RAD of the FPX. The regular expression set for each of the content scanners consisted of 21 regular expressions. The expressions chosen were aimed primarily at dropping spam. For example, "*Get Rich Quick*" and "*(L|l)imited (T|t)ime (O|o)ffer*" were among the expressions in the set. On average, each regular expression was 20 characters long. Note that these were simpler expressions than those found in the *SpamAssassin* database.

Each of the scanners was tested in the lab using NCHARGE [24] for initial testing. NCHARGE allowed single packets to be sent to the content scanner for easier debugging. Later stages of testing were conducted using real Internet traffic via web browsers, email clients, and FTP clients. The configuration of the lab setup is shown in Figure 4.6. This type of testing allowed placement of pseudo-viruses and other content on the Internet to verify detection and potential dropping by the scanning module.

The following subsections describe the device utilization and throughput of the content scanner modules on a Xilinx Virtex XCV2000E-6 part.

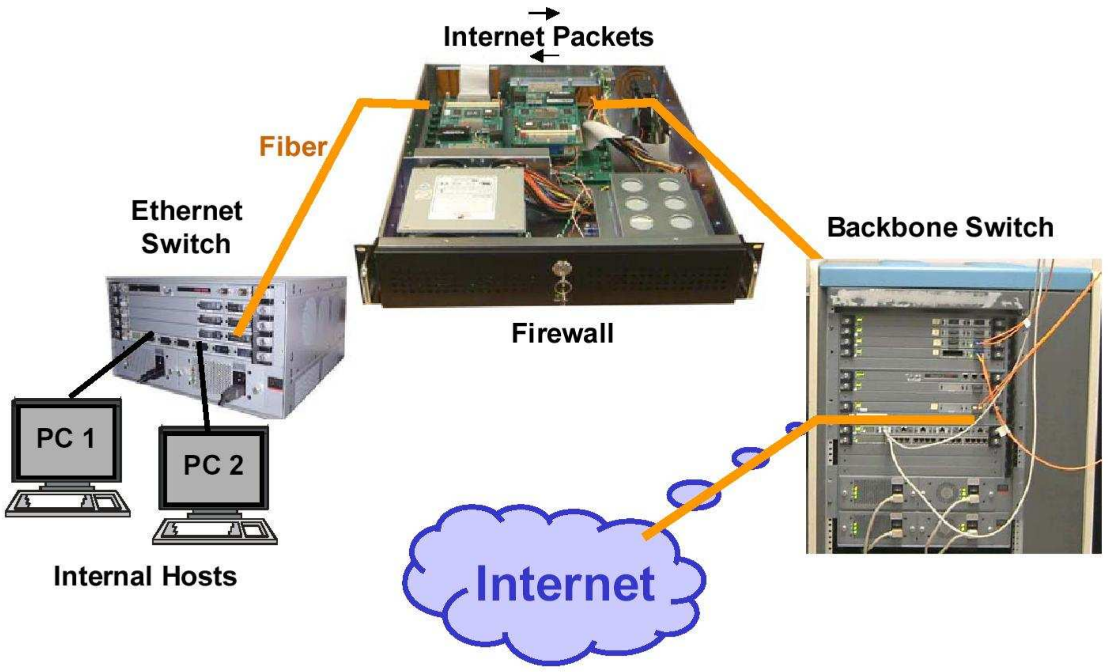


Figure 4.6: Laboratory test layout

4.4.1 Device Utilization

Device utilization for three different modules is shown in Tables 4.1, 4.2, and 4.3. Table 4.1 shows the device utilization for a module containing only the Protocol Wrappers. These values represent the overhead of the packet processing done by the Protocol Wrappers. Table 4.2 details the device utilization for a single content scanner that implements the spam filter with the Protocol Wrappers. Table 4.3 shows the device utilization for the four parallel content scanners shown in Figure 4.5 with the Protocol Wrappers. The chart in Figure 4.7 helps to illustrate the relative sizes of each of the modules.

Table 4.1: Device Utilization for Protocol Wrappers

Resources	**Virtex XCV2000E Device Utilization**	**Utilization Percentage**
Logic Slices	2410 out of 19200	12%
Flip Flops	2870 out of 38400	7%
Block RAMs	19 out of 160	11%
External IOBs	142 out of 512	27%

Table 4.2: Device Utilization for FPgrep Module with Single Content Scanner

Resources	**Virtex XCV2000E Device Utilization**	**Utilization Percentage**
Logic Slices	3615 out of 19200	18%
Flip Flops	3981 out of 38400	10%
Block RAMs	33 out of 160	20%
External IOBs	142 out of 512	27%

Table 4.3: Device Utilization for FPgrep Module with Quad Content Scanners

Resources	**Virtex XCV2000E Device Utilization**	**Utilization Percentage**
Logic Slices	6508 out of 19200	33%
Flip Flops	6182 out of 38400	16%
Block RAMs	66 out of 160	41%
External IOBs	142 out of 512	27%

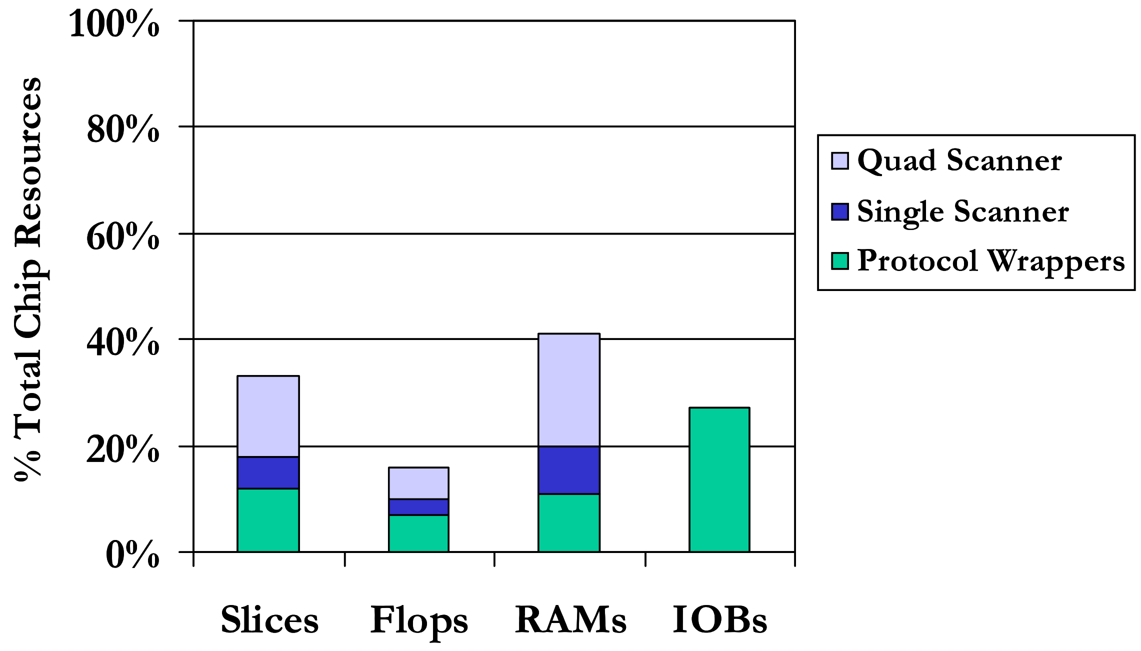


Figure 4.7: FPgrep device utilization

As mentioned earlier, simplified DFAs use an average of n states for an n length regular expression. This translated into the hardware as using on average 1 flip-flop per character (minus the overhead associated with the Protocol Wrappers and the logic controller). Figure 4.8 shows the relationship between the number of characters in the search string and the number of slices required to implement the FPgrep hardware module.

4.4.2 Throughput

The single-scanner spam filtering module with 21 regular expressions currently places and routes at 37 MHz. The critical path of the module was found to be the fanout of the 8-bit lines to each of the DFAs. The quad-scanner module has similar results; it also places and routes at 37 MHz. Given that each scanner is capable of processing

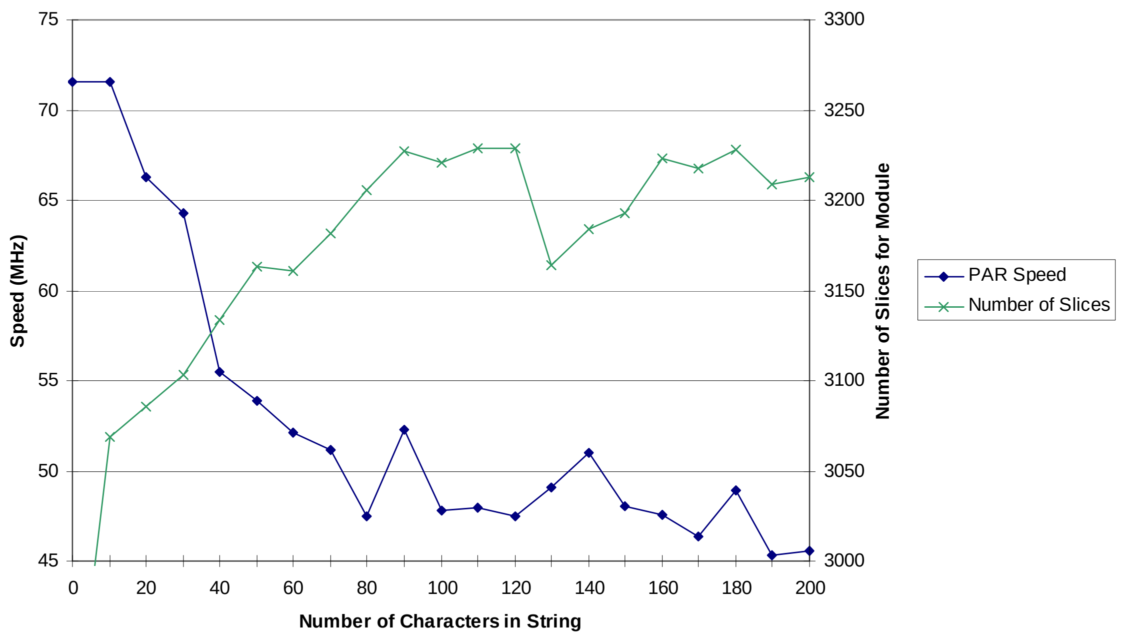


Figure 4.8: Speed and slice utilization as a function of the number of characters in the search string

8-bits of data per cycle, the throughput of the single-scanner spam filter can be calculated as 37 MHz $\times$ 8 bits = 296 Mbps. By running four content scanners in parallel, the module can reach 37 MHz $\times$ 8 bits $\times$ 4 = 1.184 Gbps.

When limiting the scanner to only several small DFAs (thus minimizing the fanout bottleneck), the module is capable of achieving frequencies in the range of 70-80 MHz. At these frequencies, the module is capable of exceeding 2.5 Gbps. The graph in Figure 4.8 displays the relationship between the number of characters in the search string and the speed of the FPGA.

Chapter 5

FPsed: Packet Payload Search-and-Replace Using Regular Expressions

The streaming content editor, FPsed, was implemented as a module on the FPX platform. The content editor has the ability to perform regular expression searches and replacements on network packets passing through the module. The function is similar to the substitute command of Unix's stream editor utility (sed). The scanner utilizes the Protocol Wrappers to process IP packets and delineate the header and payload fields in hardware

5.1 Search-and-Replace

The FPsed module performs both replacement and global replacement operations on packet payloads. The task of string replacement of a regular expression is not as straightforward or efficient as searching. String replacement requires that the machine do more than simply determine the presence of *matching* substrings in a record. The machine must also determine the position of the first and last character of all complete substrings that are *matched* by the machine. It is this requirement that causes the task of regular expression search-and-replace to be more complicated and less efficient than a simple search.

Searching for the complete substring is logical when the goal is to replace that substring. Consider the task of replacing every occurrence of a certain hexadecimal string associated with a computer virus "$37F43(B^{+}|7^{*})$" with the text "*Virus Pattern Detected*". For the input string "$3_1 7_2 F_3 4_4 3_5 B_6 B_7 B_8$", the substring could be replaced from the point where the machine *starts running*, "3_1", to the point where the substring is *accepted* "B_6". But this would allow a portion of the virus to remain in the content stream. In most situations, it is preferable to replace only complete substrings.

To search for complete substrings, a "$.*$" can no longer be prepended to a regular expression before it is converted to an FSM. This is because prepending a "$.*$" would make it difficult to determine the first character of the *matched* substring. It is not believed that there is a general, easily automatable, method for determining the position of the first character in a *matching* substring when a "$.*$" is prepended. For instance, suppose all occurrences of "*ARL*" were to be replaced. Figure 5.1 is a state machine for the regular expression after prepending a "$.*$".

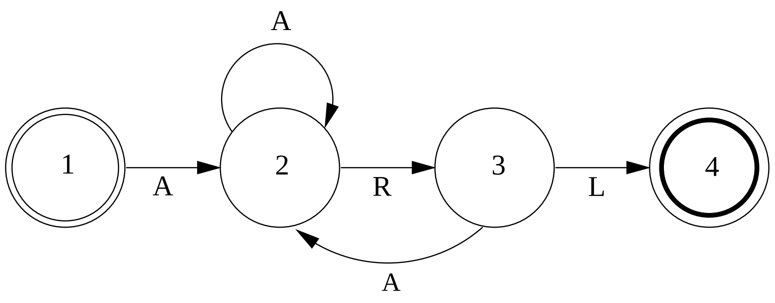


Figure 5.1: DFA for "$.*ARL$"

If the string "$A_1A_2A_3A_4R_5L_6X_7$" is input to the machine it would begin *running* when it encounters "A_1" and *matches* when "X_7" is read. This would have the effect of replacing "$A_1A_2A_3A_4R_5L_6$" when the intention is to replace "$A_4R_5L_6$". One could counter that this would not be a problem if the machine simply kept a count of how many characters were read after entering state 2, resetting the count every time it entered state 2. For this particular example that would be true, but there are examples that do not have such a simple and formulaic solution.

A slightly more complicated example is "*.*AR*L*", seen in Figure 5.2. Once again there is a solution to the problem, but this time it is more complicated. In this situation, the machine would again have to keep track of the number of characters read after entering state 2. However, this time the machine should only reset its count if the input causing the transition to state 2 is not an "R".

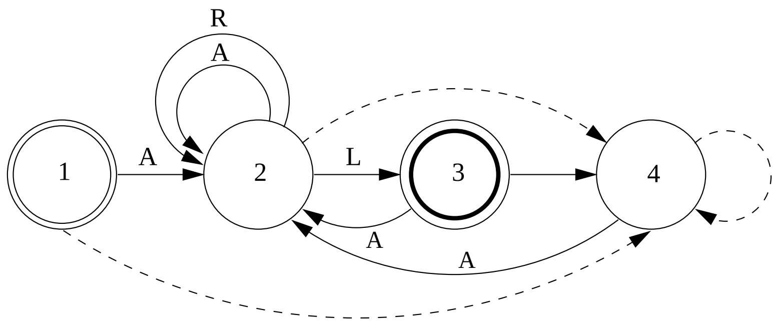


Figure 5.2: DFA for "*.*AR*L*"

An even more complicated example is the machine for "*.*A(AR)*L*" in Figure 5.3. It is left to the reader to see that the machine becomes quite complex.

It is therefore difficult to devise a general method for determining the start of the string. One can always find a more complicated regular expression that would require the addition of more rules to the method.

It has been shown that prepending a "*.**" to regular expressions that are to be replaced is clearly not a viable solution as it was with searching. In Chapter 4 on

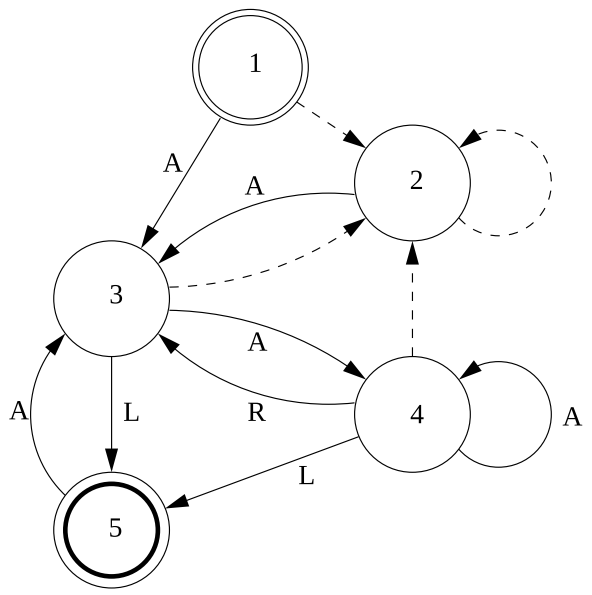


Figure 5.3: DFA for “*.*A(AR)*L*”

FPgrep, it was shown that if a “*.**” is not prepended, the searching functionality will not have the correct semantics if each character is read only once. Because of this, FPsed must employ a different technique for dealing with the problem of finding all complete substrings that *match* a particular regular expression. A solution to this problem is to use a *brute force* (or backtracking) method of searching.

5.1.1 Brute Force

The brute force technique of string searching is a simple and well studied technique. The idea behind this method is that the machine checks all characters of the input to determine if they could be the beginning of a substring that *matches* the regular expression.

The brute force technique works like this: The machine attempts to read the input string through the automaton beginning with character c_i. If, while processing the string, the machine *fails*, it immediately initiates a new search beginning with character c_{i+1}. This ensures that all characters are checked to determine if they would cause the machine to begin *running*.

As an example, assume the machine is searching for "ABC" and the input is "$A_1B_2A_3B_4C_5D_6$". In this case the machine would begin *running* on "A_1", continue to *run* on "B_2" and *fail* on "A_3". The machine would then backtrack and begin reading the input from character "B_2". It then begins *running* when it encounters "A_3", *accepts* the substring on "C_5" and *matches* the substring when it reaches "D_6".

The worst case running time occurs when every input character is a possible *starting* position, and all but the last character of the input *matches* the regular expression. For example, if the machine is searching for "$A{*}B$" and the input is "$A_0A_1 \ldots A_{n-1}$". The machine must make $O(nm)$ comparisons, where n is the string length and m is the pattern length, to determine that the pattern does not occur in the record.

The worst case condition is unlikely to appear when searching for English language expressions, but it is less rare when searching binary text. Davies and Bowsher [6] examined the efficiency of the backtracking technique when searching strings from the English language and binary strings. Their experiments involved keeping track of the number of references to an input string divided by the number of characters occurring before the *matched* substring (the index position of the pattern minus one) thus obtaining the number of inspected characters in the text string per character passed. The results of their experiments showed that when searching English text, the backtracking method referenced the text string 1 to 1.1 times per character passed. When searching binary text for an expression of length six or greater, approximately 2 characters were inspected for every character passed.

5.2 FPsed Hardware Implementation

The FPsed hardware module has been implemented to perform a brute force method for search-and-replace. The hardware consists of several components, all of which are controlled by the logic controller. A block diagram for the module can be seen in Figure 5.4.

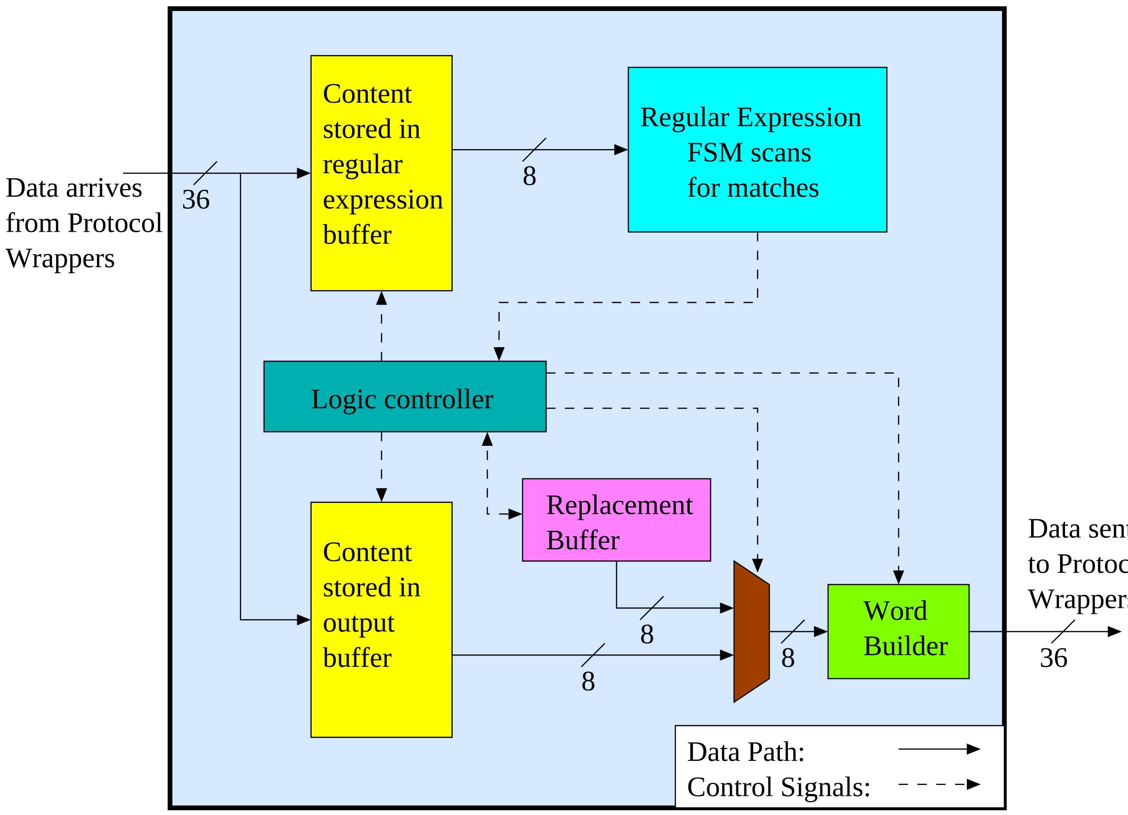

Data arrives from Protocol Wrappers
36
Content stored in regular expression buffer
8
Regular Expression FSM scans for matches
Logic controller
Replacement Buffer
Content stored in output buffer
8
8
8
Word Builder
Data sent to Protocol Wrappers
36
Data Path:
Control Signals:

Figure 5.4: Block diagram of FPsed search-and-replace module

5.2.1 Logic Controller

The logic controller is the most complex entity in the design. It controls parallel dual-ported memory buffers, the regular expression machine, the replacement buffer and the word builder using control signals that it generates. Like the FPgrep controller, the FPsed logic controller has three main phases that operate in parallel: (1) *Receiving Packets*, (2) *Processing Packets*, and (3) *Outputting Packets.*

Receiving Packets

As packets come into the module from the Protocol Wrappers, they are first written into the two parallel 512x36 dual-port memory buffers. The lower 32 bits written into the memory buffers are the incoming data. The upper four bits are used to store the four control signals from the Protocol Wrappers (start of frame, start of IP headers, start of IP payload, and end of frame). The write-side address lines are shared by the two memory buffers, as is the write-enable signal. This ensures that the contents of the two buffers are identical. The address lines are controlled by a simple counter. Each time a new word is written to memory, the counter is incremented to address the next location. Finally, because the buffer size is limited, when the buffer is almost full a congestion signal is sent upstream to notify the Protocol Wrappers and the NID to stop sending cells. Once the hardware has had a chance to catch up and complete processing of the received cells, the congestion signal is removed.

Process Characters In Regular Expression Machine

The bytes that are input to the regular expression machine come from the read side of one of the dual-port memories. Only the payload data are sent through the regular expression machine for processing. To address the memory, a counter is used to step through the memory one byte at a time. The bytes are sent to the regular expression machine and processed. Control signals from the regular expression machine tell the controller which address to read next using the following rules:

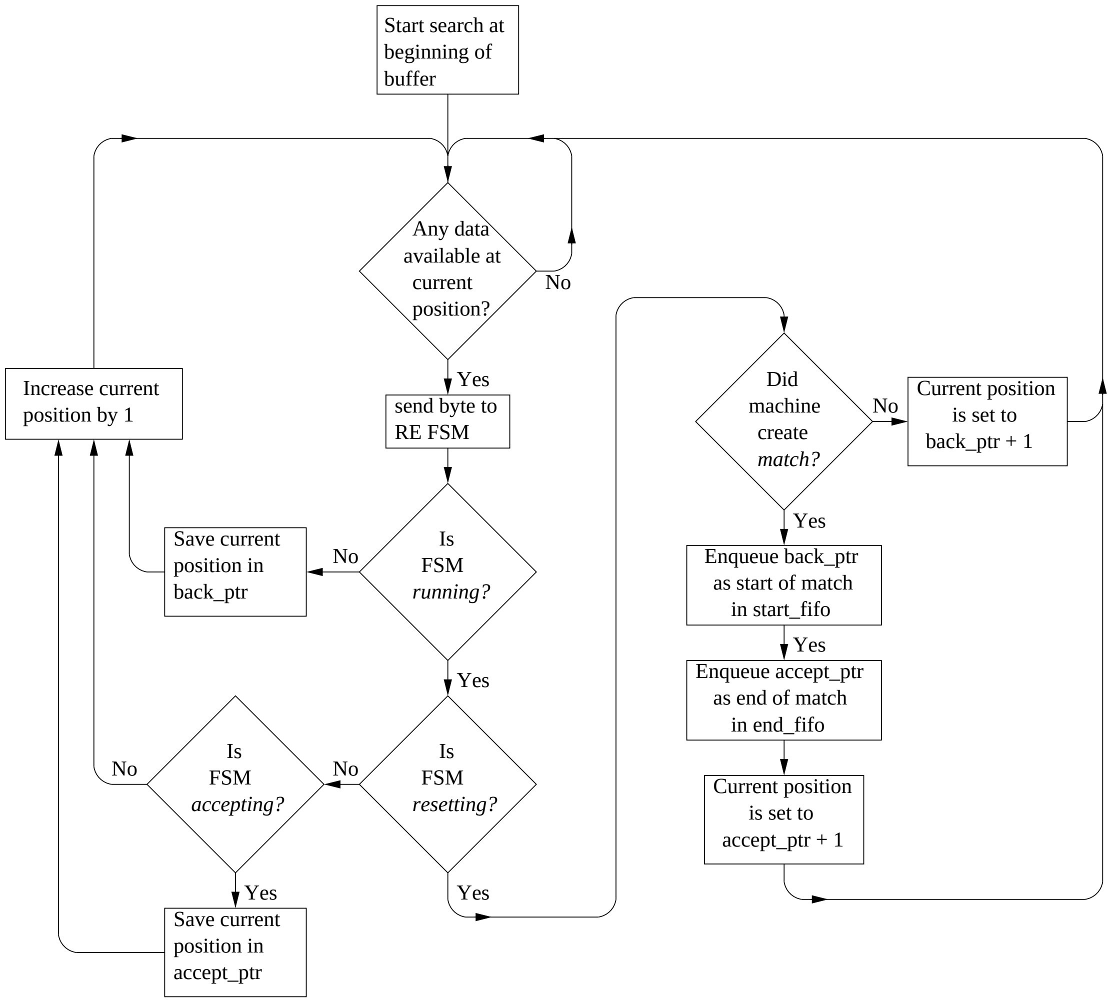

Start search at beginning of buffer
Any data available at current position?
No
Yes
send byte to RE FSM
Is FSM *running?*
No
Save current position in back_ptr
Yes
Is FSM *resetting?*
No
Is FSM *accepting?*
No
Yes
Save current position in accept_ptr
Increase current position by 1
Yes
Did machine create *match?*
No
Current position is set to back_ptr + 1
Yes
Enqueue back_ptr as start of match in start_fifo
Yes
Enqueue accept_ptr as end of match in end_fifo
Current position is set to accept_ptr + 1

Figure 5.5: FPsed flow diagram for the input/searching process

1. If no control signals are returning from the regular expression machine (i.e. not *running*), the controller updates the current *back_ptr* and increments the byte address by one.

2. If while *running* the controller receives an *accepting* signal from the regular expression machine, it stores the address of the byte that created the *match* into *accept_ptr* and continues to increment the byte address by one.

3. If the controller receives a *resetting* signal from the regular expression machine, one of two things can happen:

   (a) If while *running*, the controller did not receive an *accepting* signal, it backs up the byte address and begins processing data immediately proceeding the byte that previously *started* the machine.

   (b) If an *accepting* signal was received (a *match* was found), it backs up the byte address and begins processing data immediately proceeding the byte that caused the machine to *match*. Also, to remember where the *match* has occurred, the *starting* byte address and the ending byte address of the string are stored in two fifos (the *start_fifo* and the *end_fifo*).

When the end of a packet is reached, the controller sends a reset signal to the regular expression machine to *reset* it. A flow diagram of the above process can be seen in Figure 5.5.

Output Modified Packet

The output process (Figure 5.6) examines bytes from the read port of the second dual-port memory. It can only output data that are completely done being processed by the regular expression machine (none of the regular expression machine pointers reference the data). As the output process steps through the available bytes, it checks the previously mentioned *start_fifo* and does the following:

1. If the current output address is not stored in the *start_fifo*, then the byte is sent to the word builder and the byte address is incremented by one.

2. If the current output address is stored in the *start_fifo*, then the following occurs:

   (a) The byte is not sent to the word builder. Instead a signal is sent to the replacement buffer to begin outputting a replacement.

(b) The byte address for the output process is assigned the value stored in the *end_fifo*.

(c) The output process waits for a done signal from the replacement buffer before processing additional bytes.

Finally, as data are read from the memory and output, the four control bits are used to assert the appropriate signals back to the Protocol Wrappers.

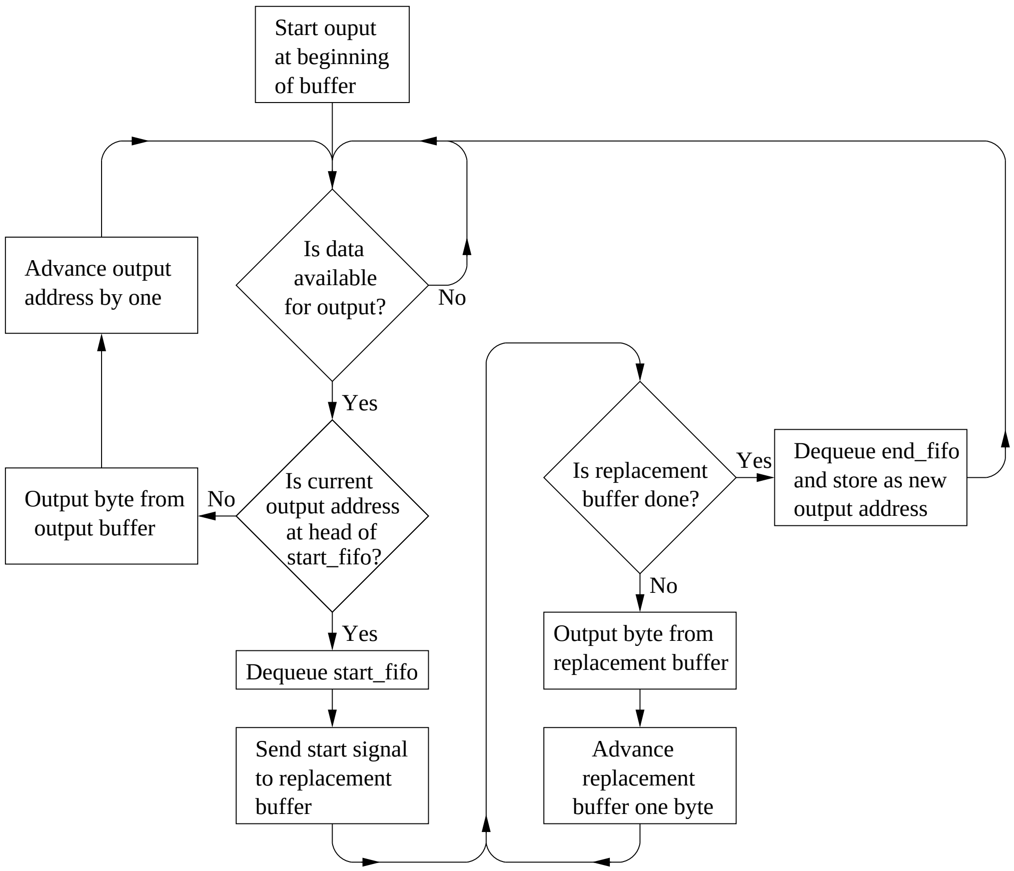

Start ouput at beginning of buffer
Is data available for output?
No
Yes
Advance output address by one
Is current output address at head of start_fifo?
No
Yes
Output byte from output buffer
Dequeue start_fifo
Send start signal to replacement buffer
Is replacement buffer done?
Yes
No
Dequeue end_fifo and store as new output address
Output byte from replacement buffer
Advance replacement buffer one byte

Figure 5.6: FPsed flow diagram for output/replacement process

5.2.2 Replacement Buffer

The replacement buffer contains a variable length array of byte strings. This array contains the replacement string, which determines the length of the array. The replacement buffer has two states: *idle* and *replace*. By default the buffer is in the *idle* state until it receives a signal from the controller to begin a replacement. At this point the buffer transitions into the *replace* state and begins outputting data. The data consist of the byte values from the array beginning at position zero and advancing one position per clock tick until the end of the array is reached. A counter is used to index the array. Finally, a special status signal is raised high for one clock cycle concurrently with the last byte in the array to notify the controller that it may resume its normal output. At this time, the replacement buffer also returns to its *idle* state.

5.2.3 Word Builder (Byte-to-Word Converter)

The word builder is the final component in the data path of the packet through the hardware. The purpose of the word builder is to take bytes from the output process of the controller and buffer them until four bytes are available for a 32-bit output from the application. A counter keeps track of how many bytes have been seen. An enable signal comes from the controller to notify the word builder of each incoming byte. The first three bytes are buffered. When the fourth byte arrives it passes through and combines with the first three bytes for a 32-bit output. An output enable signal to the controller informs the controller that the 32 bits are valid for output. The world builder also has a flush signal that comes from the controller. This signal is asserted by the controller when the last byte in the packet is being sent to the word builder. It informs the word builder that it should output however many bytes it may have buffered along with the incoming (last) byte. If there are not four bytes for output, padding (a byte of zeros) is used to complete a 32-bit word. One last output from the word builder tells the controller the number of valid bytes of data (non-padding) that went out with the last word. This information is necessary for the controller to successfully modify the byte-length field in the frame trailer.

5.2.4 Multiple Expression Search-and-Replace

Unlike the FPgrep module, the current design of the FPsed module only allows for one search-and-replace operation per module. However, multiple search-and-replace operations can be achieved by simply daisy-chaining FPsed modules together as shown in Figure 5.7. Note that daisy-chaining the modules together gives priority to search-and-replace operations that come first in the specification.

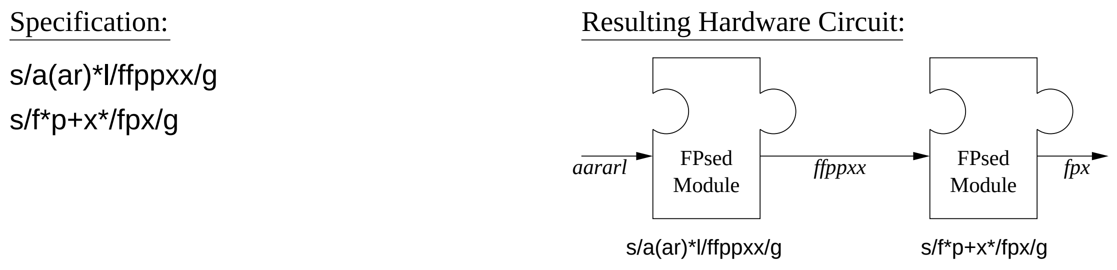
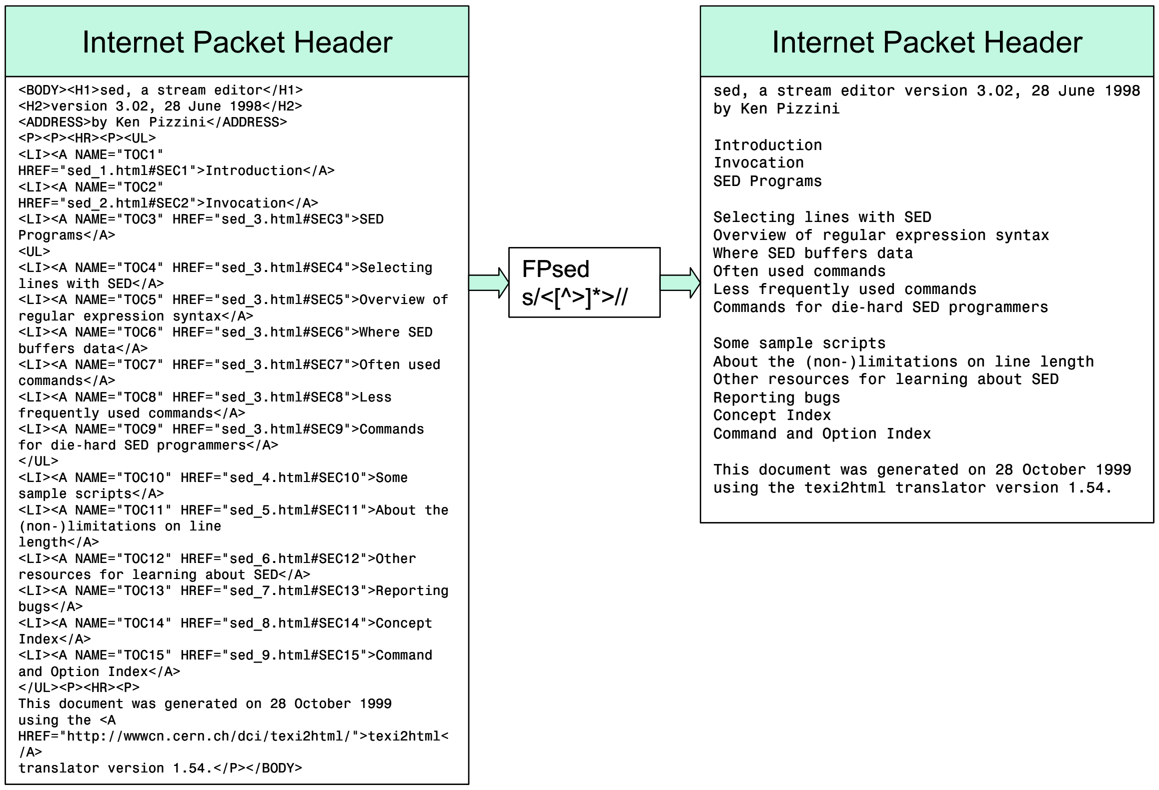


Figure 5.7: Multiple search-and-replace modules

5.3 Results

Several versions of the content editor were synthesized with the protocol wrappers into the RAD of the FPX. One module was designed to replace computer viruses as they traversed across a streaming UDP-based Internet connection. Another module was developed to remove profanity from packet payloads and was tested in the lab using a UDP-based chat client. A third module was developed to remove HTML tags from text. An example of the transformation performed by the HTML filter is shown in Figure 5.8.

The following sections describe the device utilization and estimated throughput of the content-editing modules that implements the HTML filter on a Xilinx Virtex XCV2000E-6 part.



```
<BODY><H1>sed, a stream editor</H1>
<H2>version 3.02, 28 June 1998</H2>
<ADDRESS>by Ken Pizzini</ADDRESS>
<P><P><HR><P><UL>
<LI><A NAME="TOC1"
HREF="sed_1.html#SEC1">Introduction</A>
<LI><A NAME="TOC2"
HREF="sed_2.html#SEC2">Invocation</A>
<LI><A NAME="TOC3" HREF="sed_3.html#SEC3">SED
Programs</A>
<UL>
<LI><A NAME="TOC4" HREF="sed_3.html#SEC4">Selecting
lines with SED</A>
<LI><A NAME="TOC5" HREF="sed_3.html#SEC5">Overview of
regular expression syntax</A>
<LI><A NAME="TOC6" HREF="sed_3.html#SEC6">Where SED
buffers data</A>
<LI><A NAME="TOC7" HREF="sed_3.html#SEC7">Often used
commands</A>
<LI><A NAME="TOC8" HREF="sed_3.html#SEC8">Less
frequently used commands</A>
<LI><A NAME="TOC9" HREF="sed_3.html#SEC9">Commands
for die-hard SED programmers</A>
</UL>
<LI><A NAME="TOC10" HREF="sed_4.html#SEC10">Some
sample scripts</A>
<LI><A NAME="TOC11" HREF="sed_5.html#SEC11">About the
(non-)limitations on line
length</A>
<LI><A NAME="TOC12" HREF="sed_6.html#SEC12">Other
resources for learning about SED</A>
<LI><A NAME="TOC13" HREF="sed_7.html#SEC13">Reporting
bugs</A>
<LI><A NAME="TOC14" HREF="sed_8.html#SEC14">Concept
Index</A>
<LI><A NAME="TOC15" HREF="sed_9.html#SEC15">Command
and Option Index</A>
</UL><P><HR><P>
This document was generated on 28 October 1999
using the <A
HREF="http://wwwcn.cern.ch/dci/texi2html/">texi2html<
/A>
translator version 1.54.</P></BODY>
```

```
sed, a stream editor version 3.02, 28 June 1998
by Ken Pizzini

Introduction
Invocation
SED Programs

Selecting lines with SED
Overview of regular expression syntax
Where SED buffers data
Often used commands
Less frequently used commands
Commands for die-hard SED programmers

Some sample scripts
About the (non-)limitations on line length
Other resources for learning about SED
Reporting bugs
Concept Index
Command and Option Index

This document was generated on 28 October 1999
using the texi2html translator version 1.54.
```

Figure 5.8: Example of FPsed used to strip HTML tags from a packet payload

5.3.1 Device Utilization

The utilization of FPGA resources for three different modules is shown in Tables 5.1, 5.2 and 5.3. Table 5.1 shows the device utilization for a module containing only the Protocol Wrappers. These values represent the overhead of the packet processing done by the Protocol Wrappers. Table 5.2 details the device utilization for a single content editor that implements the HTML filter along with the Protocol Wrappers. Finally, Table 5.3 shows the device utilization for four parallel content editors with the Protocol Wrappers. The chart in Figure 5.9 illustrates the relative sizes of each of the modules.

Table 5.1: Device Utilization for Protocol Wrappers

Resources	**Virtex XCV2000E Device Utilization**	**Utilization Percentage**
Logic Slices	2410 out of 19200	12%
Flip Flops	2870 out of 38400	7%
Block RAMs	19 out of 160	11%
External IOBs	142 out of 512	27%

Table 5.2: Device Utilization for FPsed Module with Single Content Editor

Resources	**Virtex XCV2000E Device Utilization**	**Utilization Percentage**
Logic Slices	2922 out of 19200	15%
Flip Flops	3223 out of 38400	8%
Block RAMs	21 out of 160	13%
External IOBs	142 out of 512	27%

Table 5.3: Device Utilization for FPsed Module with Quad Content Editors

Resources	**Virtex XCV2000E Device Utilization**	**Utilization Percentage**
Logic Slices	4131 out of 19200	21%
Flip Flops	4093 out of 38400	10%
Block RAMs	56 out of 160	35%
External IOBs	142 out of 512	27%

5.3.2 Throughput

A single content-editing RE module for the HTML filter was synthesized for the Virtex XCV2000E-6 FPGA that implements the RAD on the FPX platform. The

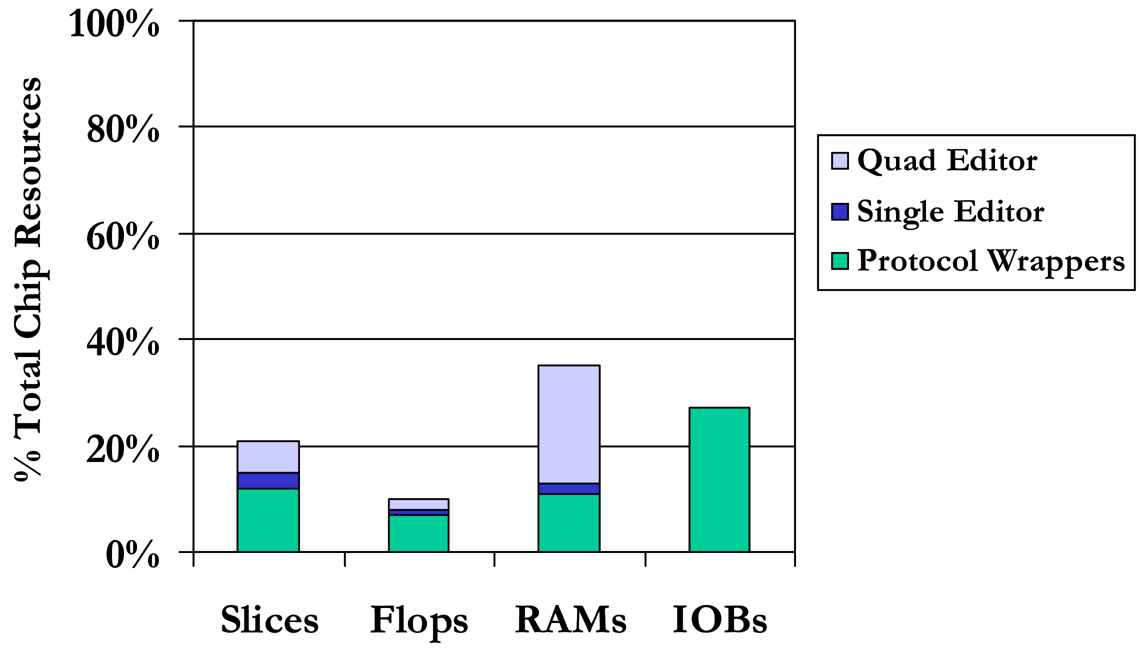


Figure 5.9: FPsed device utilization

FPGA was placed and routed using the Xilinx backend tools to run at 64 MHz. This provides a throughput of 64 MHz $\times$ 8 bits = 512 Mbps.

An experiment was run to mimic FPsed's functionality with software running on four different computers. One computer, a dual Intel Pentium 3 operating at 1 GHz running a Linux 2.2 kernel, achieved 13.7 Mbps when the sed program (version 3.02) read data from disk. To ensure that disk I/O was not a bottleneck, the same program was run completely from memory and achieved 32.72 Mbps. This is approximately 16x slower than the FPsed hardware. Another computer, an Alpha 21364 operating at 667 MHz running Linux kernel 2.4, was able to perform a search-and-replace operation on data at 36 Mbps when the input was read from disk, and 50.4 Mbps when the input was run completely from memory. On average, the fastest computers were 10x slower than a single FPsed module. Throughput results for all four computers are shown in Figure 5.10.

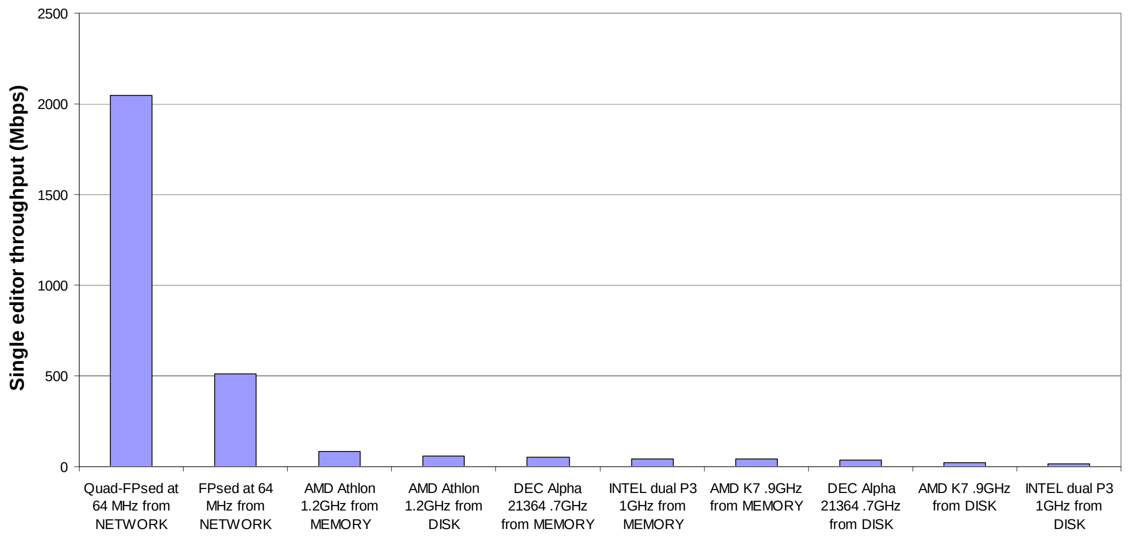


Figure 5.10: Comparison of hardware and software throughput

As with FPgrep, the throughput of the FPsed module can be further increased by instantiating multiple content editors in parallel and dispatching incoming packets to an available editor (see section 4.3). For instance, if the HTML filter uses four parallel editors, then the throughput can increase four-fold to 2.048 Gbps. This gives FPsed a 40x advantage over the software as shown in Figure 5.10.

Chapter 6

Generating the Hardware

The FPgrep and FPsed modules were designed to be easily reconfigurable when new search terms are desired. It is necessary to have certain components such as the regular expression finite state automata and the replacement buffer generated because no single finite state automaton can be used to search for every regular expression. Also, no single replacement string is suitable for all applications. To accomplish this task, a complete design flow was implemented using a combination of static VHDL components and dynamic VHDL components that are generated according to a user specification. The design flow is completely automated, starting with a list of regular expressions and ending with a hardware bitstream being programmed into the RAD of an FPX.

6.1 Hardware Generation Implementation Details

Generating the hardware for the FPgrep and FPsed modules is done using a series of scripts. These scripts are accessible via a shell command line or an easy to use web interface. The input specifications to the scripts are slightly different for FPgrep and

FPsed. For FPgrep, the specification contains a list of regular expressions, each with an identification number associated with it that is also programmed into the hardware. The identification number is inserted into an alert packet when the regular expression is matched. The syntax for FPgrep and an example of a single list entry can be seen below:

syntax:

$/expression/prop(id\ number)/$

example of a virus scanner:

$/Vi(R|r)u(S|s)/prop(6)/$

The input specification for FPsed is slightly different. It consists of a list of regular expressions and their corresponding replacement strings. The syntax for FPsed and a couple examples can be seen below:

syntax:

$s/expression/replacementstring/$

example of stripping out HTML tags:

$s/<[\hat{}>]*>//$

example of a profanity blocker:

$s/(P|p)rofa(N|n)ity/*******/$

The complete process of generating hardware from the user specified regular expression to the resulting module can be seen in Figure 6.1 (note that dashed-line blocks are for FPsed only).

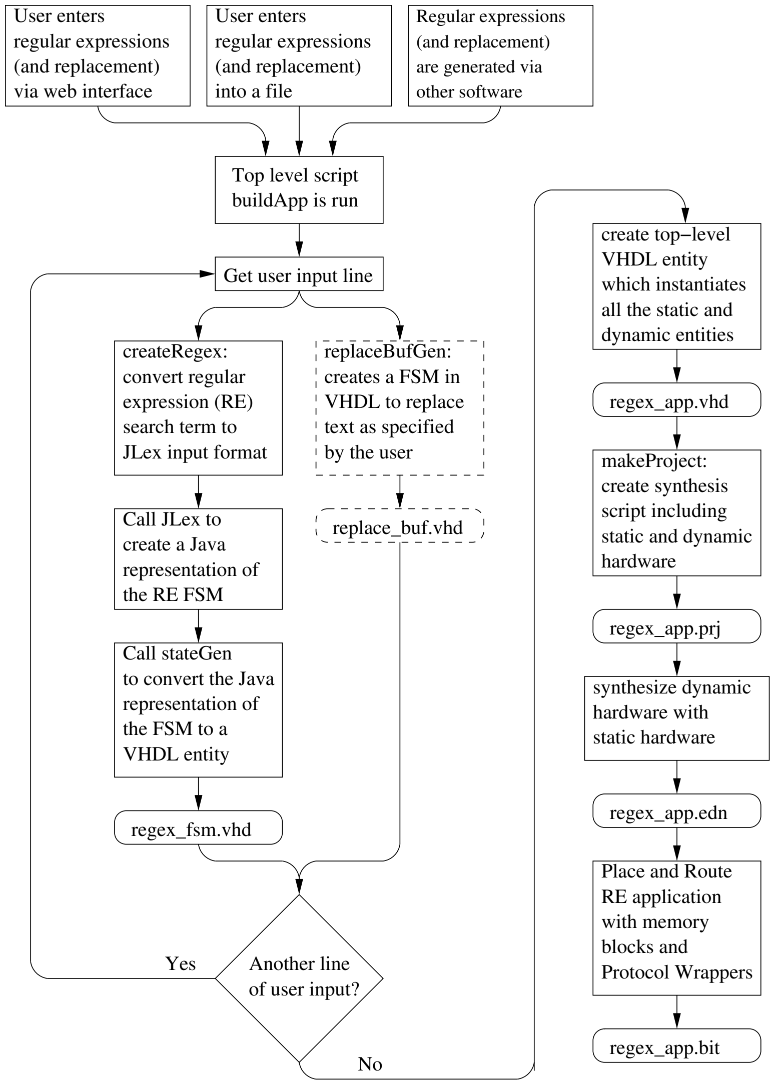

User enters regular expressions (and replacement) via web interface
User enters regular expressions (and replacement) into a file
Regular expressions (and replacement) are generated via other software
Top level script buildApp is run
Get user input line
createRegex: convert regular expression (RE) search term to JLex input format
replaceBufGen: creates a FSM in VHDL to replace text as specified by the user
Call JLex to create a Java representation of the RE FSM
replace_buf.vhd
Call stateGen to convert the Java representation of the FSM to a VHDL entity
regex_fsm.vhd
Another line of user input?
Yes
No
create top–level VHDL entity which instantiates all the static and dynamic entities
regex_app.vhd
makeProject: create synthesis script including static and dynamic hardware
regex_app.prj
synthesize dynamic hardware with static hardware
regex_app.edn
Place and Route RE application with memory blocks and Protocol Wrappers
regex_app.bit

Figure 6.1: Hardware generation from specification to bitstream

6.1.1 Top Level Script: *buildApp*

The main structure of the device is generated by a script called *buildApp*. *BuildApp* reads the list of regular expression and performs several functions. First, it runs a script called *createRegex* to create a regular expression machine for each entry in the list. For FPsed, it also runs a script called *replaceBufGen* to create a replacement buffer for each entry in the list. Finally, *buildApp* generates a VHDL entity for the top-level design (*regex_app.vhd*), which instantiates each of the previously generated components and wires them together.

6.1.2 Create Regular Expression FSM: *createRegex*

The *createRegex* script is responsible for creating each regular expression state machine needed for the design. This script first creates an input file (*jlex_in*) for JLex to parse. Next, JLex parses this file creating a Java representation of the regular expression state machine. Finally, *createRegex* calls another script called *stateGen* to parse the Java file.

6.1.3 Convert Java FSM to VHDL: *stateGen*

The *stateGen* script takes in a JLex-created Java representation of a regular expression state machine. It parses this Java file and extracts the pertinent state machine information, including state transition tables and *accepting* states. Using this information, the *stateGen* script then creates a VHDL representation of the state machine. For FPgrep, the identification number associated with each regular expression is also included in the hardware. Separate state machines are created for each regular expression in the user specification that are called *regex_fsmi.vhd*, where i indicates the regular expression number.

6.1.4 Create Replacement Buffer: *replaceBufGen*

The *replaceBufGen* script creates a VHDL representation of a state machine that controls the output of a replacement string for FPsed. Note that separate replacement buffers are created for each regular expression in the user specification that are called *replace_bufi.vhd*, where i indicates the replacement number.

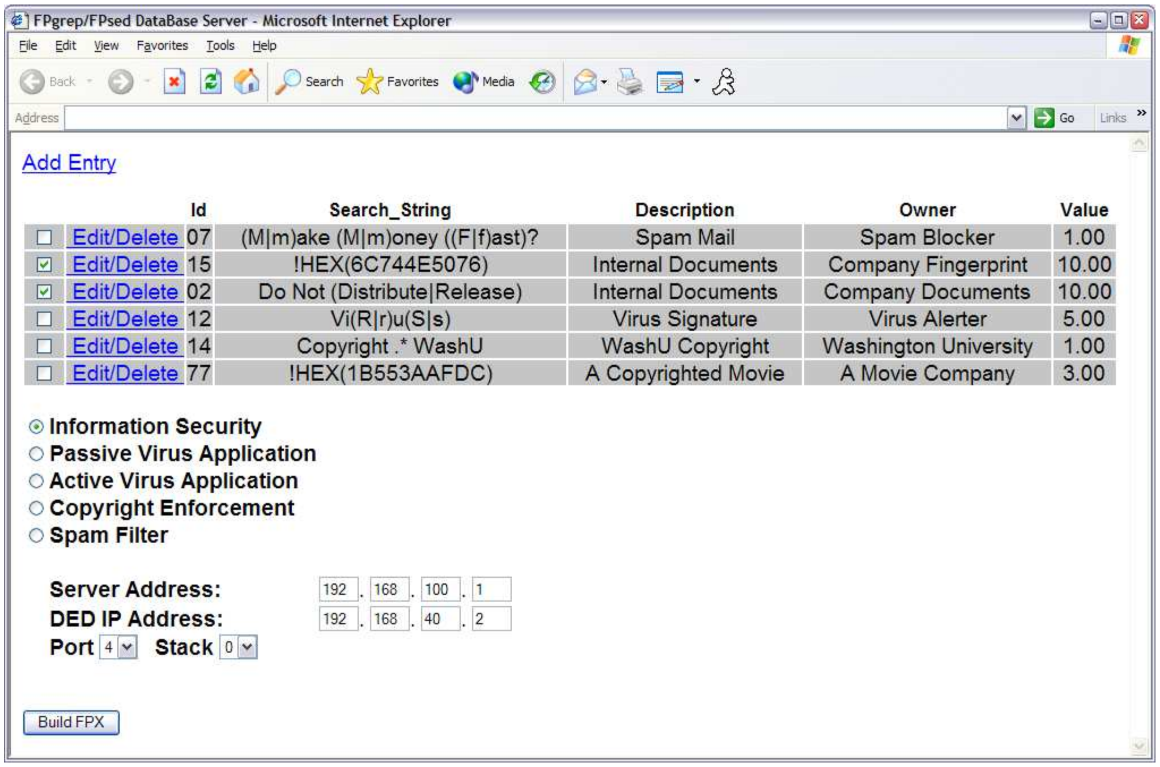
6.1.5 Create Synthesis Script: *makeProject*

Each new user specification may have a different number of regular expression strings. This creates different files each time a new module is generated. Because of this, the synthesis scripts required to build the hardware need to be different for each generated module. The *makeProject* script creates a *.prj* file which can be used to synthesize and map each specification using the Synplicity [26] synthesis tools (*regex_app.prj*).

6.2 Web Interface

A PHP-based [28] web interface was created to simplify designing new modules (currently only supports FPgrep). The interface allows regular expressions to be added, edited, or deleted from a list of available expressions. The expressions along with the associated information is stored in a MySQL [17] database. A screenshot of the interface is shown in Figure 6.2. From the web, an administrator can select any subset of the available expressions to be programmed into the hardware. The resulting module can be used for several applications such as virus protection, network security, copyright enforcement, or spam filtering. For network security and spam filtering applications, the Internet address of a server is specified to determine where alert packets should be delivered. For networks that contain multiple FPX devices, the FPX IP address, the port, and the stack specifies which FPX should be reprogrammed.

For example, if the second and third boxes of Figure 6.2 are checked, then the FPX will be programmed to scan for two regular expressions. First, it will scan packets for the hex string "$6C744E5076$" which appears in a digital document fingerprint. Additionally, it will scan packets for the regular expression "*Do Not (Distribute|Release)*". If a packet containing either of these regular expressions is found, then an alert packet is sent to the server address *192.168.100.1*. Once the


FPgrep/FPsed DataBase Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address Go Links

Add Entry

		Id	Search_String	Description	Owner	Value
☐	Edit/Delete	07	(M|m)ake (M|m)oney ((F|f)ast)?	Spam Mail	Spam Blocker	1.00
☑	Edit/Delete	15	!HEX(6C744E5076)	Internal Documents	Company Fingerprint	10.00
☑	Edit/Delete	02	Do Not (Distribute|Release)	Internal Documents	Company Documents	10.00
☐	Edit/Delete	12	Vi(R|r)u(S|s)	Virus Signature	Virus Alerter	5.00
☐	Edit/Delete	14	Copyright .* WashU	WashU Copyright	Washington University	1.00
☐	Edit/Delete	77	!HEX(1B553AAFDC)	A Copyrighted Movie	A Movie Company	3.00

[✓ Information Security]
Passive Virus Application
Active Virus Application
Copyright Enforcement
Spam Filter

Server Address: 192 . 168 . 100 . 1
DED IP Address: 192 . 168 . 40 . 2
Port 4 Stack 0

Build FPX

Figure 6.2: Web interface for generating modules

"Build FPX" button is pressed, the design scripts proceed to synthesize, place and route, and reprogram the FPX over the network. More details on the applications listed in Figure 6.2 are discussed later in Chapter 7

Chapter 7

Applications for Managing Digital Content

A variety of applications have been developed using the FPgrep and FPsed hardware modules in conjunction with a suite of software tools. The software tools include a MySQL database, a Java-based server that communicates with the database, as well as a Java-based (also ported to C#) agent that communicates with the server. Each of these components will be further described with respect to each application in the following sections.

7.1 Spam Filter

Over the past decade, spam has become a menace to anyone with an email account. Recent surveys show that between 50% and 60% of all email is now spam [11]. In order to identify and effectively eliminate spam, packet payloads need to be completely examined for spam keywords.

FPgrep offers the power and flexibility to search for a number of regular expressions representing spam keywords. By programming a list of spam keywords into the FPgrep hardware module and placing the hardware upstream from a mail server as shown in Figure 7.1, much of the spam can be eliminated. Valid packets are allowed to pass through the FPgrep module and arrive at the mail server. However, when packets containing targeted spam keywords are processed by the FPgrep module, these offending packets can be dropped before reaching the mail server. FPgrep also offers the ability to send an alert packet to the mail server (or some other log server) to log the event. The format of this alert packet can be seen in section A.1 of Appendix A.

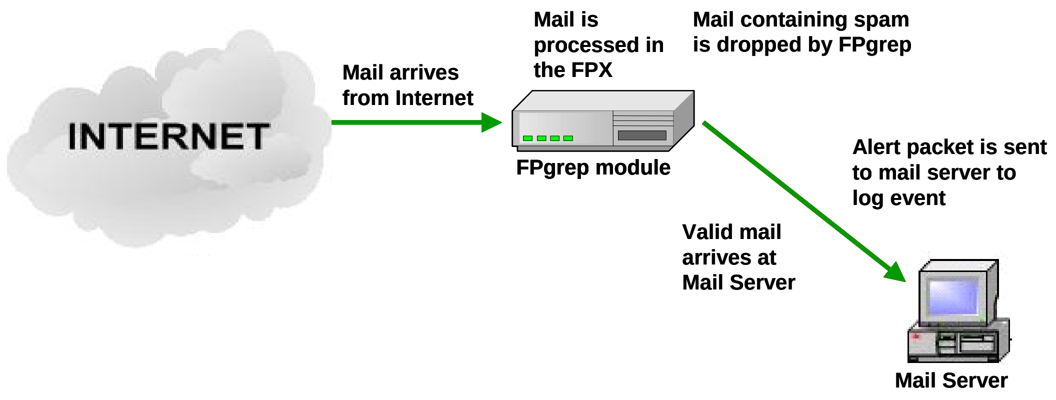

INTERNET
Mail arrives from Internet
Mail is processed in the FPX
Mail containing spam is dropped by FPgrep
FPgrep module
Alert packet is sent to mail server to log event
Valid mail arrives at Mail Server
Mail Server

Figure 7.1: FPgrep used as a spam filter

7.2 Virus Protection

Besides spam, viruses have become not only an annoyance, but also a costly hazard for businesses connected to the Internet. Figure 7.2 represents a typical network layout that is unprotected from modern viruses. If *Host C* is infected with a virus, then all the shaded nodes in the network now become vulnerable. Notice that no node on the network is safe.

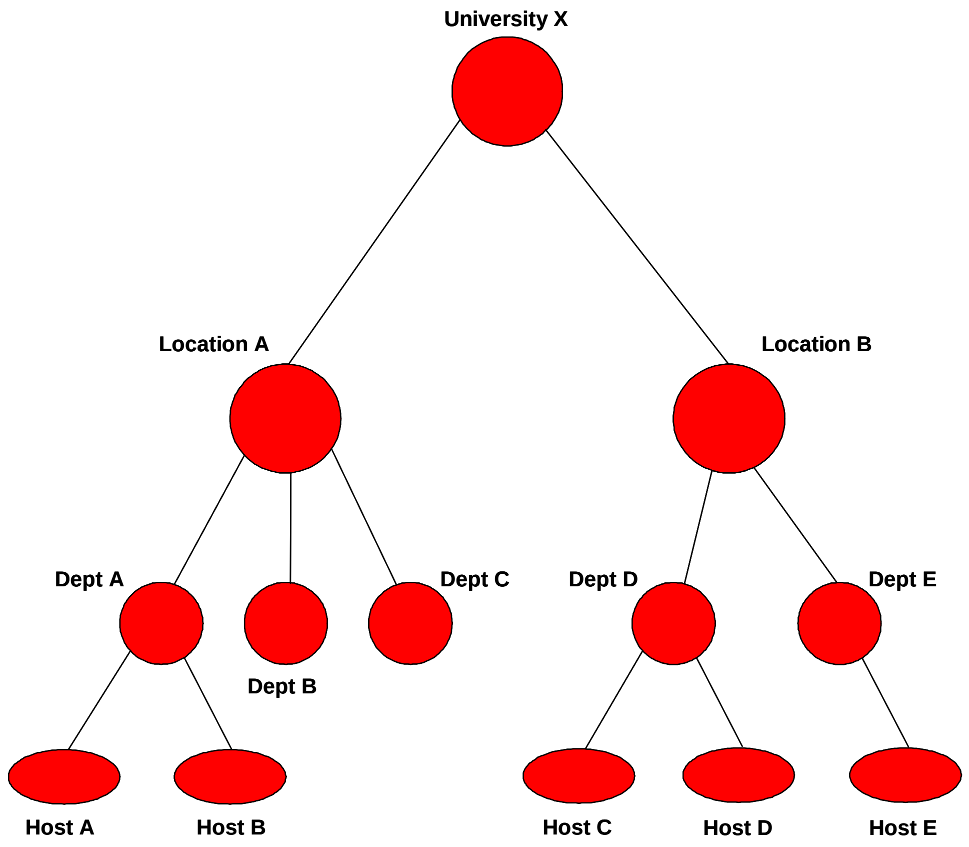


Figure 7.2: Typical unprotected network

By placing a protective system such as FPgrep or FPsed into the network and programming it to search for digital virus signatures, malicious viruses can be effectively identified and quarantined before they are able to spread. Figure 7.3 depicts the same network as in Figure 7.2. However, when the network is protected only a small portion of the nodes on the network are now vulnerable.

The following sections describe two possible virus protection solutions that can be offered via the FPgrep module: a passive approach and an active approach. Both approaches are similar in that the FPgrep module is programmed to search for digital signatures of some number of viruses. Each digital signature may consist of

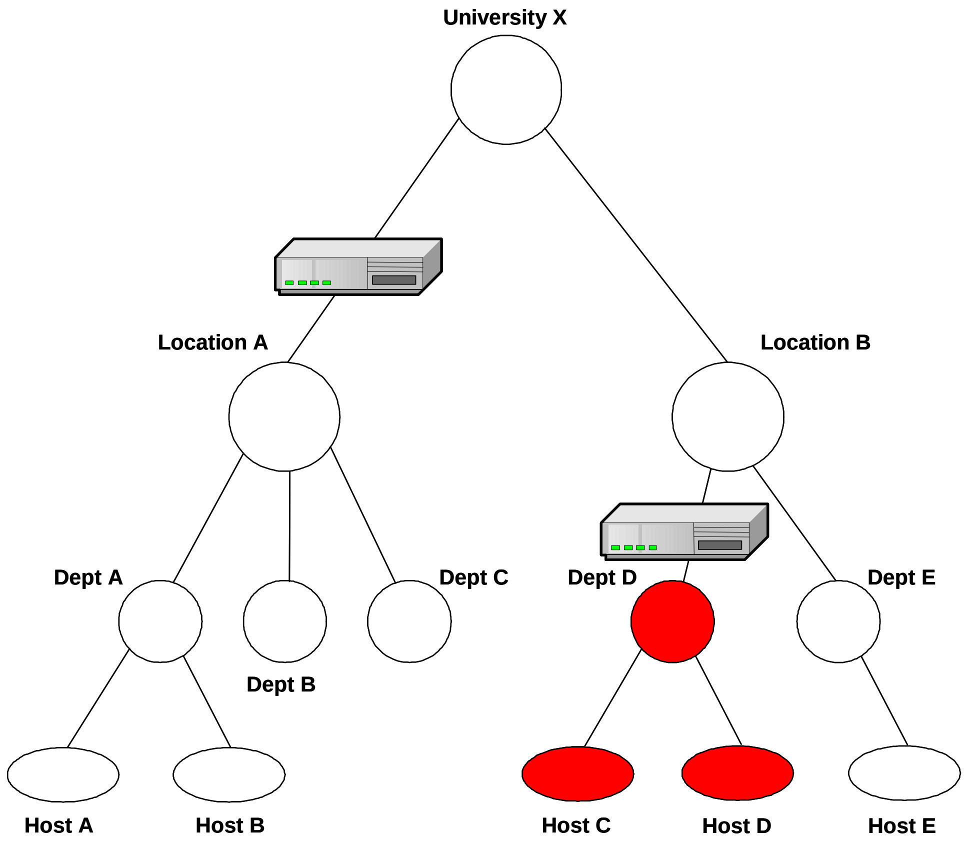


Figure 7.3: Network protected with FPgrep or FPsed

a hexadecimal string extracted from an executable virus, several lines of code from a macro virus, or some other identifying characteristic of the virus. The passive and active virus applications differ only in their behavior once a virus is detected.

7.2.1 Passive Virus

When running the passive virus application, the FPgrep module passively monitors all traffic flowing through the network. This means that if a virus is detected by the system, the virus is allowed to continue on to its destination. However, before sending the offending packet, a UDP alert packet (see section A.1 of Appendix A) is generated and sent to the same destination as the offending packet. A software agent residing on the destination machine receives the alert packet and informs the user of the machine of the possible downloaded virus via a prompt similar to the one seen in Figure 7.5. This sequence of events can be followed in Figure 7.4.

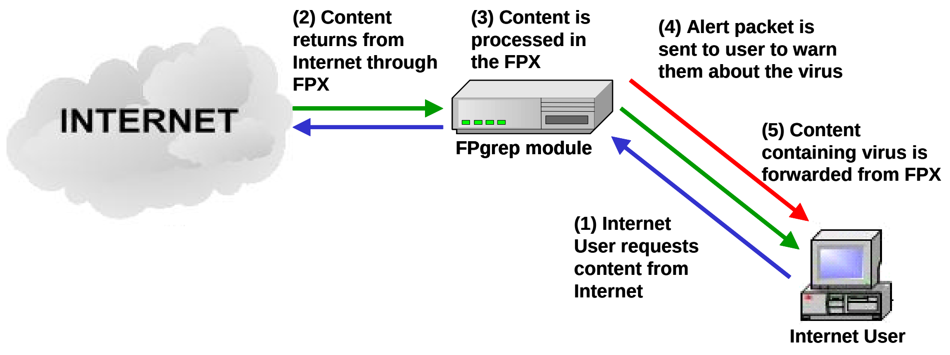
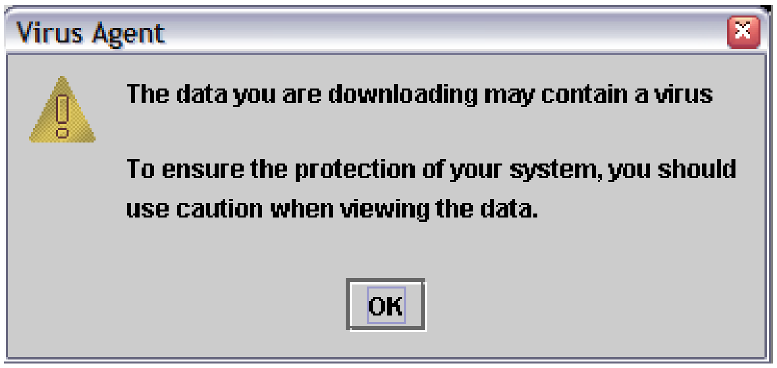

INTERNET
(2) Content returns from Internet through FPX
(3) Content is processed in the FPX
FPgrep module
(4) Alert packet is sent to user to warn them about the virus
(5) Content containing virus is forwarded from FPX
(1) Internet User requests content from Internet
Internet User

Figure 7.4: FPgrep used for passive virus protection


Virus Agent

The data you are downloading may contain a virus

To ensure the protection of your system, you should use caution when viewing the data.

Figure 7.5: Alert message generated by agent software

7.2.2 Active Virus

The active virus application uses the FPgrep module to actively monitor all traffic flowing through the network. This is different from the passive application because now when a virus is detected in a packet, the packet is not allowed to continue on to its destination. Instead, the virus is dropped by the FPgrep module, thereby protecting the destination machine. However, just as in the passive application, the FPgrep module generates a UDP alert packet and sends it to the destination machine where a software agent receives the packet and generates a screen prompt. The sequence of events for the active virus application and the screen prompt can be seen in Figures 7.6 and 7.7.

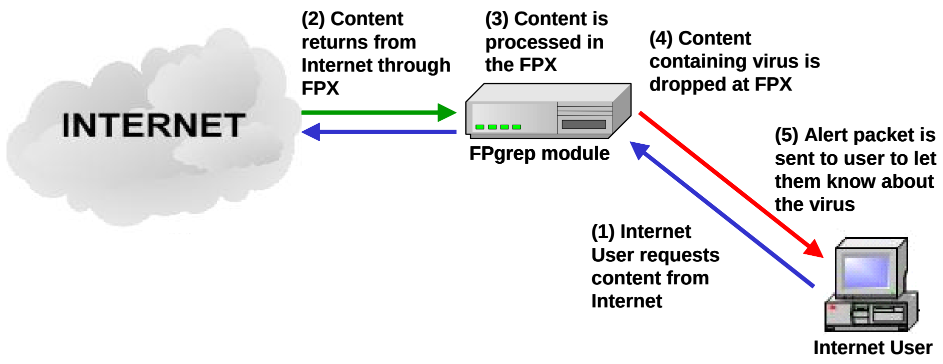
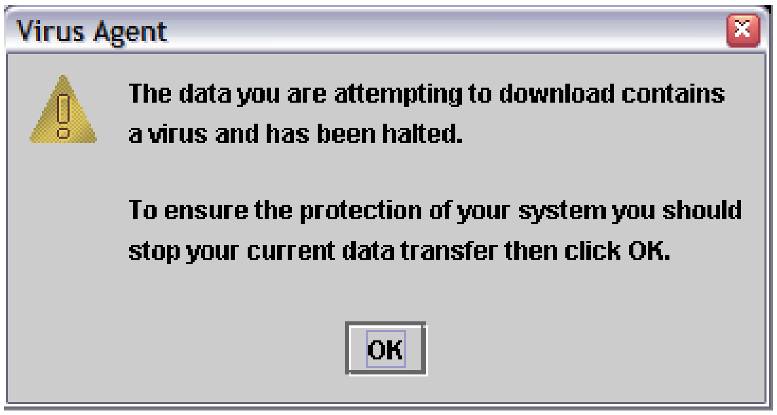

INTERNET
(2) Content returns from Internet through FPX
(3) Content is processed in the FPX
(4) Content containing virus is dropped at FPX
FPgrep module
(5) Alert packet is sent to user to let them know about the virus
(1) Internet User requests content from Internet
Internet User

Figure 7.6: FPgrep used for active virus protection



The data you are attempting to download contains a virus and has been halted.

To ensure the protection of your system you should stop your current data transfer then click OK.

Figure 7.7: Alert message generated by agent software

7.3 Information Security

Another important issue with respect to many corporate and medical networks has to do with the security of the information on the network and the information that is being accessed by machines on the network. It is crucial for both business and legal reasons that documents deemed to be *internal documents* stay internal to the network, and information deemed to be a liability or unsuitable not be allowed onto the network.

The first problem of maintaining internal documents can be easily accomplished with the FPgrep module by embedding some digital signature or watermark into each document, or by using some regular expression that is common to all internal documents such as "*CONFIDENTIAL*" or "*Do Not (Distribute|Release)*". The FPgrep hardware could then be programmed to search for this signature and drop all packets containing the signature as they attempt to exit the network.

The second problem of disallowing certain information onto the network would require an administrator to identify and create regular expressions for each piece of information that was not allowed onto the network. This sounds like an arduous task, but can be greatly simplified by finding commonalities between many pieces of information and using one regular expression to search for all of them. For instance, if administrators wanted to eliminate all traffic utilizing the KaZaA [21] peer-to-peer software, they simply need to program the FPgrep module to search for the common KaZaA application header. This would cause the FPgrep module to drop all KaZaA packets thus rendering the software unusable.

Just as with the virus protection algorithm, the security application has both an active mode that drops offending packets and a passive mode that allows offending packets to pass through (Figure 7.8). Whichever application is used, it is always important that all network security breaches be reported to some administrator. Because of this, the FPgrep security application sends a UDP alert packet (see section A.1 of Appendix A) to some predetermined administrative server whenever a match is detected.

The alert packet is received by a Java-based security application. The application logs the source and destination IP addresses of the offending packet as well as what information was matched in the packet. The security application also has the ability to access the *Property* database table displayed in Figure 6.2 to retrieve the information related to each property ID in the alert packet. This information consists of a description along with a value for each expression found. The values for each expression found can be accumulated to represent some priority level given to each alert. An example prompt for a security breach can be seen in Figure 7.9.

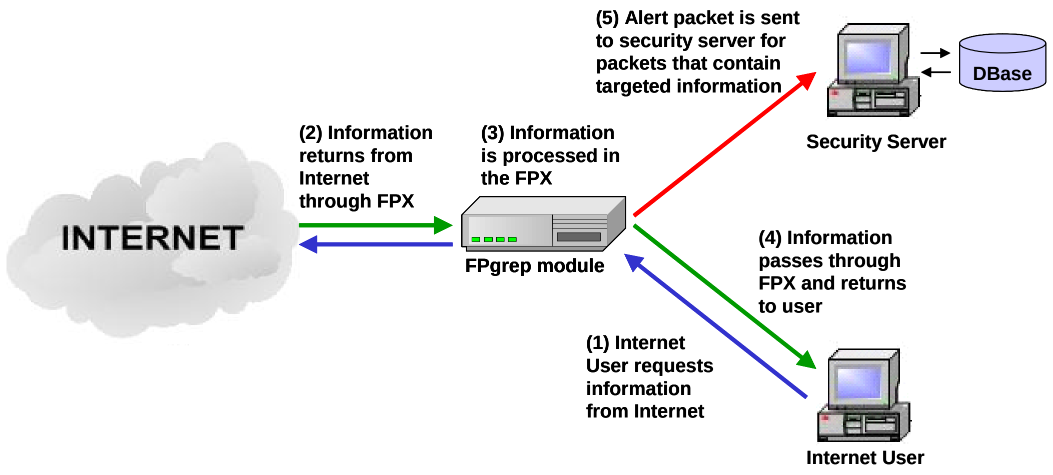
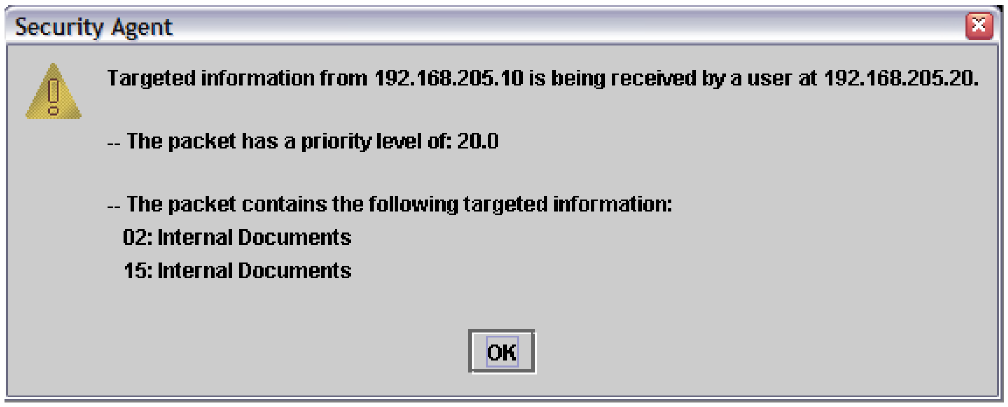

(5) Alert packet is sent to security server for packets that contain targeted information
DBase
Security Server
(2) Information returns from Internet through FPX
(3) Information is processed in the FPX
INTERNET
FPgrep module
(4) Information passes through FPX and returns to user
(1) Internet User requests information from Internet
Internet User

Figure 7.8: FPgrep used as a passive security application



Targeted information from 192.168.205.10 is being received by a user at 192.168.205.20.

-- The packet has a priority level of: 20.0

-- The packet contains the following targeted information:
02: Internal Documents
15: Internal Documents

Figure 7.9: Alert message generated by agent software

7.4 Copyright Enforcement

The final application that was developed for the FPgrep module is a method to legitimize the distribution of copyrighted digital media such as music and movies. With the widespread use of peer-to-peer networks such as KaZaA, Morpheus, BearShare and others, the illegal distribution of digital music and movie files has had a tremendous strain on their respective industries.

To alleviate this problem, a complete hardware/software solution has been developed including the FPgrep hardware, a software agent for the end systems, and a transaction server (database tables for the transaction server can be seen in Appendix B). Each software agent is registered to a specific user with the transaction server. The transaction server keeps track of all registered agents, registered users, as well as what content each user is authorized to download.

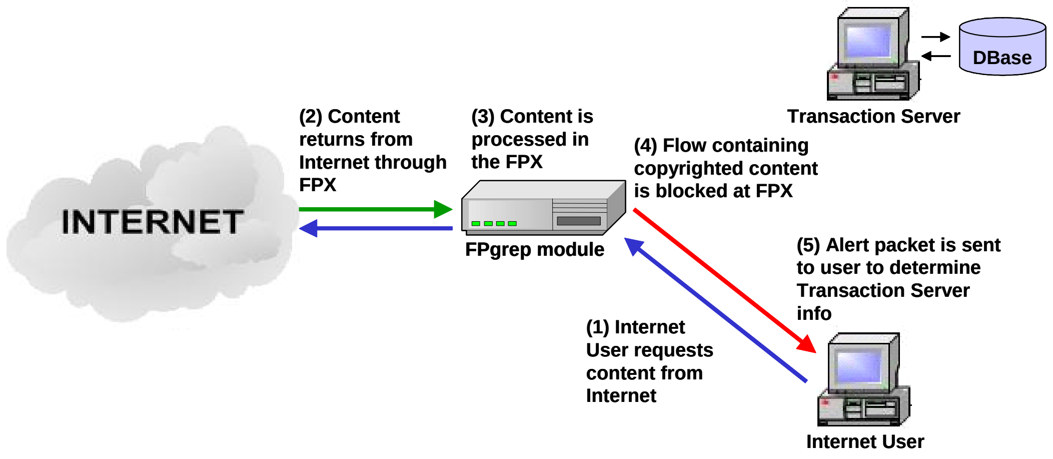

DBase
Transaction Server
(2) Content returns from Internet through FPX
(3) Content is processed in the FPX
(4) Flow containing copyrighted content is blocked at FPX
INTERNET
FPgrep module
(5) Alert packet is sent to user to determine Transaction Server info
(1) Internet User requests content from Internet
Internet User

Figure 7.10: FPgrep's copyright enforcement application (1 of 6)

For this application the FPgrep module can be programmed to search for either Digital Rights Management (DRM) tags, or just byte stings that occur in commonly traded music and movie files. When a match is detected in a packet, the FPgrep module can prevent that packet along with all subsequent packets from reaching the destination machine. Instead an alert packet (see section A.1) is sent to the destination machine with information about what match or matches occurred in the offending packet as shown in Figure 7.10.

When the alert packet arrives at the destination machine, the agent software appends a user ID# and an agent ID# to the alert packet (see section A.2.1) and forwards it to the transaction server. When the transaction server receives the packet, it checks a *Rights* table in the database to see if the user is authorized to download the content. If so, the transaction server sends a release packet (see section A.2.3) to the FPgrep module allowing the blocked content to pass through. This is shown in Figure 7.11.

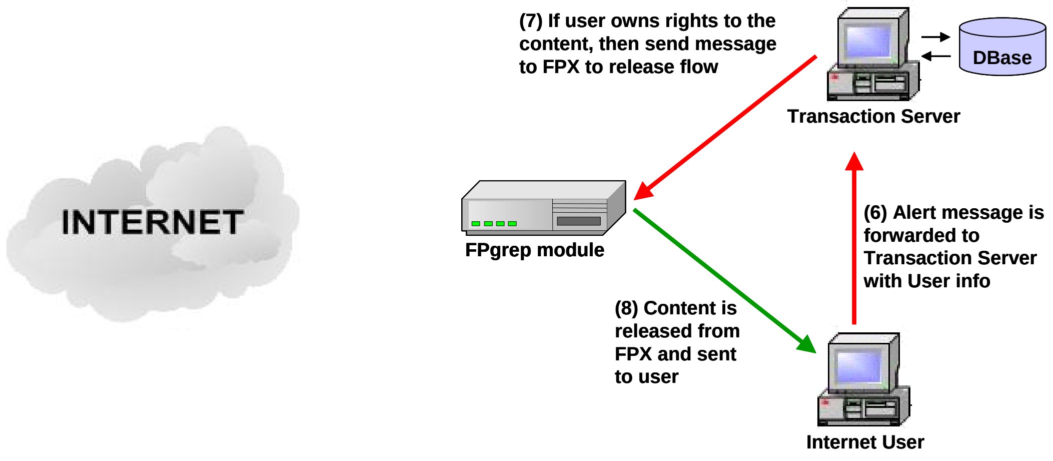

(7) If user owns rights to the content, then send message to FPX to release flow
DBase
Transaction Server
INTERNET
FPgrep module
(6) Alert message is forwarded to Transaction Server with User info
(8) Content is released from FPX and sent to user
Internet User

Figure 7.11: FPgrep's copyright enforcement application (2 of 6)

If the transaction server determines that the user is not authorized to download the content, it will send a purchase request (see section A.2.2) to the user as shown in Figure 7.12. The purchase request contains a textual description of the content that was blocked by the FPgrep module as well as a cost for the content. This enables a user to make an informative decision about purchasing the content.

Once the purchase prompt arrives at the user's machine, the agent software generates a screen prompt with all of this information giving the user a choice to purchase or not to purchase the content. An example screen prompt can be seen in Figure 7.13.

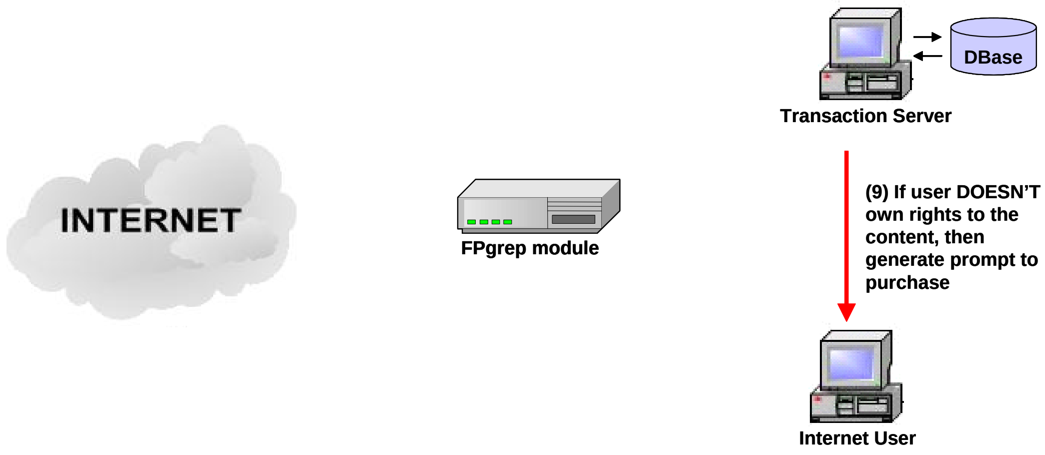
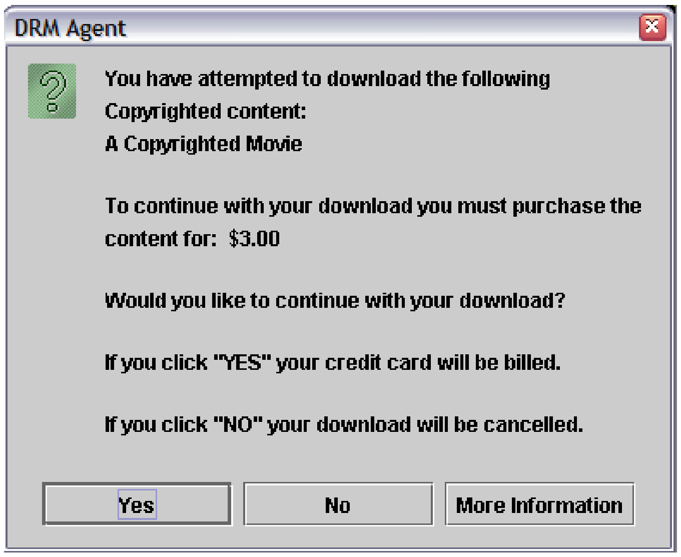

DBase
Transaction Server
INTERNET
FPgrep module
(9) If user DOESN'T own rights to the content, then generate prompt to purchase
Internet User

Figure 7.12: FPgrep's copyright enforcement application (3 of 6)



You have attempted to download the following
Copyrighted content:
A Copyrighted Movie

To continue with your download you must purchase the content for: $3.00

Would you like to continue with your download?

If you click "YES" your credit card will be billed.

If you click "NO" your download will be cancelled.

Figure 7.13: A prompt to purchase copyrighted content

Once the user make a decision, a response packet (see section A.2.1) is sent to the transaction server. This is shown in Figure 7.14.

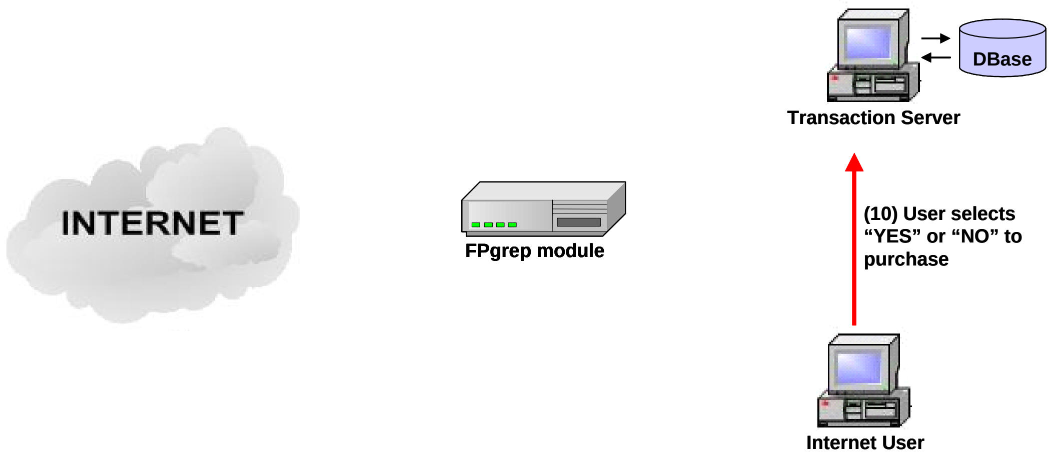
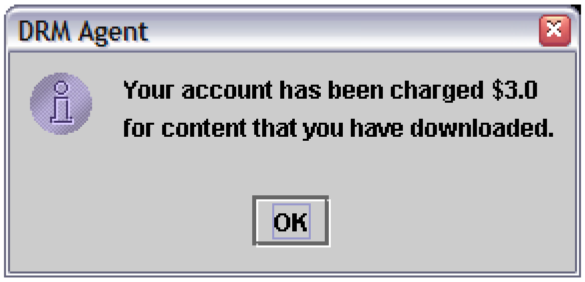


Figure 7.14: FPgrep's copyright enforcement application (4 of 6)

In the event that a user selects *YES* and decides to purchase the copyrighted content, several events follow. First, the transaction server makes an entry in the *Rights* table so that subsequent downloads of the same content are allowed to pass through the FPgrep module. At this point the transaction server can also debit a user account for the purchase. Next, the transaction server sends a release packet (see section A.2.3) to the FPgrep module allowing the blocked content to pass through. Finally, the transaction server also sends a receipt packet (see section A.2.4) to the user to let them know that their account has been debited. The receipt packet contains information about how much the user's account was debited and creates a screen prompt similar to the prompt in Figure 7.15. This whole sequence of events is shown in Figure 7.16.



Your account has been charged $3.0
for content that you have downloaded.

Figure 7.15: A receipt prompt for copyrighted content

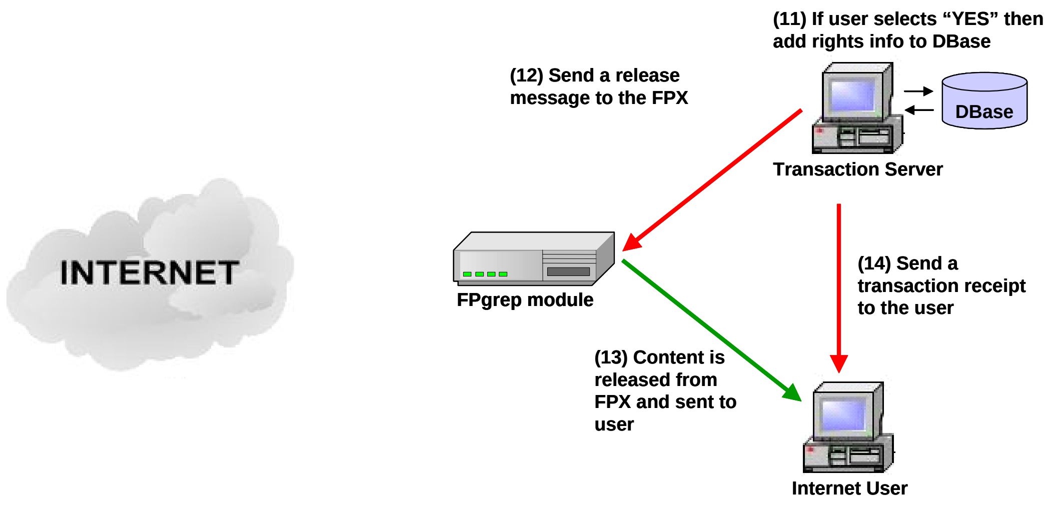

(11) If user selects "YES" then add rights info to DBase
DBase
(12) Send a release message to the FPX
Transaction Server
INTERNET
FPgrep module
(14) Send a transaction receipt to the user
(13) Content is released from FPX and sent to user
Internet User

Figure 7.16: FPgrep's copyright enforcement application (5 of 6)

If the user decides that they are not interested in downloading the copyrighted content and selects *NO* when presented with the option, the transaction server simply sends a drop packet (see section A.2.3) to the FPgrep module. This packet tells the module that it should stop buffering data associated with the copyrighted content and just drop all the data. This can be seen in Figure 7.17.

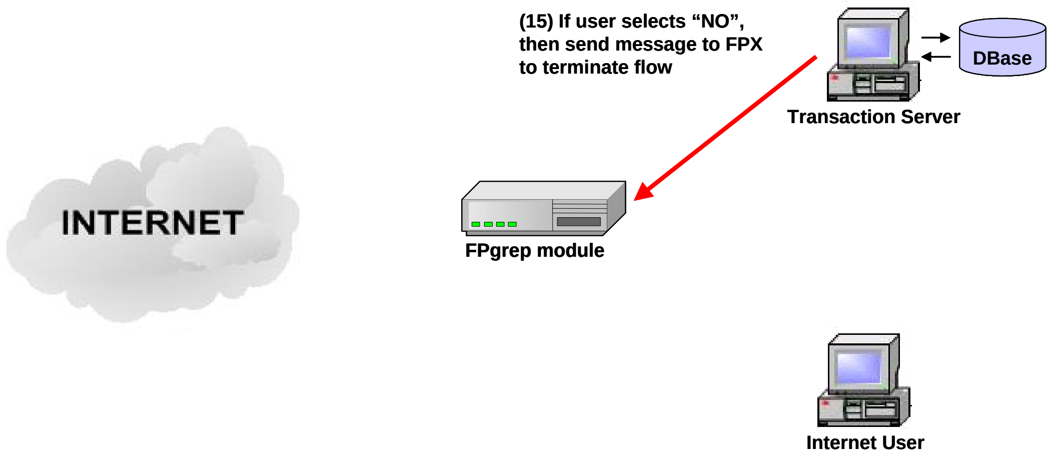

(15) If user selects "NO", then send message to FPX to terminate flow
DBase
Transaction Server
INTERNET
FPgrep module
Internet User

Figure 7.17: FPgrep's copyright enforcement application (6 of 6)

Chapter 8

Summary and Future Work

8.1 Summary

In this thesis two hardware modules, FPgrep and FPsed, have been presented that were designed to help manage the distribution of digital content.

The FPgrep module is capable of passively scanning packet payloads for a set of regular expressions, actively dropping packets that match any of the regular expressions, and generating alert packets to notify administrators of any match that occurs. The module was implemented on the FPX and tested using real TCP/IP Internet traffic on both the WUGS and in a stand-alone configuration. FPgrep is capable of operating at speeds of 1.184 Gbps for twenty-one ∼20 character regular expressions, and exceeding speeds of 2.5 Gbps for smaller numbers of similar regular expressions.

The FPsed module is capable of scanning packet payloads for a set of regular expressions and actively replacing any occurrence of a match. The module was implemented on the FPX and tested using a UDP-based chat client over the Internet.

The hardware solution was found to be about 60 times faster than similar software solutions when using parallel modules.

A simplified design flow was implemented for both modules. The design flow accepts user input in common regular expression syntax, generates the necessary VHDL, places and routes the design, and even programs the FPGA with the new bitstream. The design flow can be accessed through either command lines tools or through a web interface.

Finally, several applications were developed to utilize the capabilities of the FP-grep and FPsed modules. These include a spam filter, a virus protection application, an information security application, as well as a copyright enforcement application.

8.2 Future Work

8.2.1 FPgrep

Currently, the bottleneck in the FPgrep module is the fanout associated with sending 8-bits of data to all of the parallel DFAs. To improve the timing, a tree structure of flip-flops (as described and implemented in [7]) can be used to distribute the data to all of the DFAs to minimize the propagation delay.

The content scanner is designed to support several different behaviors as described in Chapter 4. However, the behaviors such as sending an alert message are not specified on an expression-by-expression basis. For example, if the content scanner is compiled to send an alert packet, it will send that alert packet for all the regular expressions in the module. It is more desirable to have the action rules based on which expression matched. Doing this would allow alert packets to be sent for some regular expressions in the module and not for others. An enhanced version of awk's pattern-rule syntax [25] (but with reduced instruction set) has been defined which is suitable for this task.

Finally, the content scanner currently looks for matches on a packet-by-packet basis. This means that if a string that should cause a match spans multiple packets, it will be missed by the content scanner. This behavior may be improved by utilizing the TCP-Splitter [19, 20] to process data on a stream-by-stream basis. This entails augmenting the content scanner to a multi-context design that maintains a match context for each available flow and switches contexts based on the stream that is currently being presented by the TCP-Splitter.

8.2.2 FPsed

Currently, the FPsed module does not search for multiple regular expressions in the same fashion as the FPgrep module. Each FPgrep module is capable of using parallel DFAs and searching for multiple regular expressions. In contrast, each FPsed module only searches for a single regular expression. FPsed modules are daisy-chained together when searching for more than one regular expression. This not only wastes the limited resources of the FPGA, but it also adds a great deal of latency. Future versions of the FPsed module will employ an architecture similar to that of the FPgrep module.

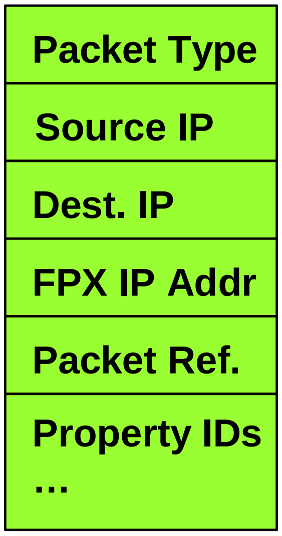
The idea of doing search-and-replace on TCP/IP streams is also something that has been considered. Preliminary plans for accomplishing this have been discussed.

Appendix A

Application Packet Formats

A.1 Common Packet Formats for All Applications

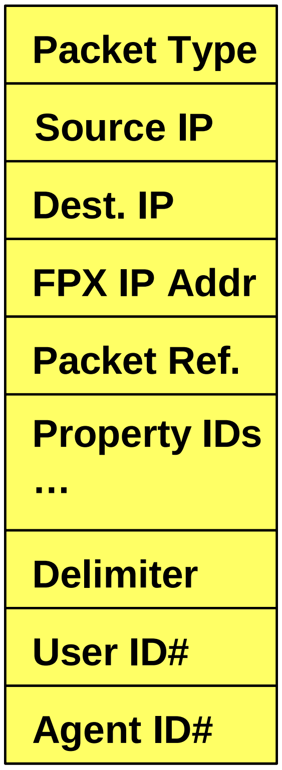
A.1.1 Alert Packet Format



- Packet Type:
  - byte 0:
    - opcode = $0x00$ (Information Security)
    - opcode = $0x01$ (Passive Virus Protection)
    - opcode = $0x02$ (Active Virus Protection)
    - opcode = $0x03$ (Copyright Enforcement)
    - opcode = $0x04$ (Spam Filter)
  - byte 1-3: unused
- Source IP address of packet that caused this alert packet
- Destination IP address of packet that caused this alert packet to be generated
- FPX IP address is the address of the FPX device that found the match and generated this alert packet
- Packet Reference ID# is a 32-bit identifier used by the hardware to identify this alert packet
- Property IDs is a list of 32-bit integers that identifies which content matched in the original packet

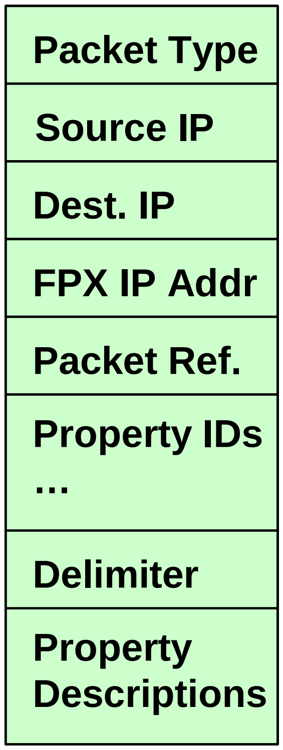
A.2 Other Copyright Enforcement Application Packet Formats

A.2.1 Forwarded Alert Packet and Purchase Prompt Response Formats



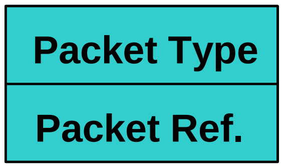
- Packet Type:
  - Forwarded Alert Packet
    - byte 0: opcode = $0x03$
    - byte 1-3: unused
  - Purchase Prompt Response
    - byte 0:
      - opcode = $0x16$ if user selects *YES*
      - opcode = $0x18$ if user selects *NO*
    - byte 1-3: unused
- Source IP address of packet that caused this alert packet to be generated
- Destination IP address of packet that caused this alert packet to be generated
- FPX IP address is the address of the FPX device that found the match and generated this alert packet
- Packet Reference ID# is a 32-bit identifier used by the hardware to identify this alert packet
- Property IDs is a list of 32-bit integers that identifies which content matched in the original packet
- The Delimiter is used to separate the User ID# from the Property IDs. This delimiter is 32-bits of zeros. (NOTE: This means that when assigning property IDs, the database must begin at 1 instead of 0 to prevent confusion).
- The User ID# is a 32-bit number that identifies each user that has registered with the system
- The Agent ID# is a 32-bit number that identifies an Agent that has been registered with the system.

A.2.2 Purchase Prompt Request Format



- Packet Type:
  - byte 0: opcode = $0x10$
  - byte 1-3: a representation of the value of the sum of *all* property IDs the user doesn't currently have rights to download
- Source IP address of packet that caused this alert packet to be generated
- Destination IP address of packet that caused this alert packet to be generated
- FPX IP address is the address of the FPX device that found the match and generated this alert packet
- Packet Reference ID# is a 32-bit identifier used by the hardware to identify this alert packet
- Property IDs is a list of 32-bit integers that identifies which content matched in the original packet
- The Delimiter is used to separate the Property Descriptions from the Property IDs. This delimiter is 32-bits of zeros. (NOTE: This means that when assigning property IDs, the database must begin at 1 instead of 0 to prevent confusion).
- Property Description is a string which identifies to the user what was matched by the hardware.

A.2.3 Release and Drop Packet Formats



- Packet Type:
  - Release Packet
    - byte 0: opcode = $0x12$
    - byte 1-3: unused
  - Drop Packet
    - byte 0: opcode = $0x14$
    - byte 1-3: unused
- Packet Reference ID# is a 32-bit identifier used by the hardware to identify this alert packet

A.2.4 Receipt and Not Registered Packet Formats

Packet Type

- Packet Type:
  - Receipt Packet
    - byte 0: opcode = $0x22$
    - byte 1-3: unused
  - Not Registered Packet
    - byte 0: opcode = $0x21$
    - byte 1-3: unused

Appendix B

Database Tables

This appendix describes the MySQL tables that are used for the applications described in Chapter 7 as well as the table described in section 6.2 that is used with the web interface.

Property Table (used with web interface)

```
+---------------+--------------+------+-----+---------+----------------+
| Field         | Type         | Null | Key | Default | Extra          |
+---------------+--------------+------+-----+---------+----------------+
| id            | int(11)      |      | PRI | NULL    | auto_increment |
| search_string | varchar(50)  |      |     |         |                |
| description   | varchar(30)  |      |     |         |                |
| owners_id     | int(11)      |      |     | 0       |                |
| value         | decimal(9,2) |      |     | 0.00    |                |
+---------------+--------------+------+-----+---------+----------------+
```

Agents Table

```
+----------+---------+------+-----+---------+-------+
| Field    | Type    | Null | Key | Default | Extra |
+----------+---------+------+-----+---------+-------+
| users_id | int(11) |      |     | 0       |       |
| agent_id | int(11) |      |     | 0       |       |
+----------+---------+------+-----+---------+-------+
```

Owners Table

```
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     |      | PRI | NULL    | auto_increment |
| name  | varchar(30) |      |     |         |                |
+-------+-------------+------+-----+---------+----------------+
```

Rights Table

```
+---------------+---------+------+-----+------------+----------------+
| Field         | Type    | Null | Key | Default    | Extra          |
+---------------+---------+------+-----+------------+----------------+
| invoice_num   | int(11) |      | PRI | NULL       | auto_increment |
| users_id      | int(11) |      |     | 0          |                |
| property_id   | int(11) |      |     | 0          |                |
| purchase_date | date    |      |     | 0000-00-00 |                |
+---------------+---------+------+-----+------------+----------------+
```

Users Table

```
+--------------+--------------+------+-----+------------+----------------+
| Field        | Type         | Null | Key | Default    | Extra          |
+--------------+--------------+------+-----+------------+----------------+
| id           | int(11)      |      | PRI | NULL       | auto_increment |
| login        | varchar(16)  |      |     |            |                |
| password     | varchar(16)  |      |     |            |                |
| name         | varchar(30)  |      |     |            |                |
| address      | varchar(30)  |      |     |            |                |
| city         | varchar(30)  |      |     |            |                |
| state        | char(2)      |      |     |            |                |
| zip          | varchar(5)   |      |     |            |                |
| credit_card  | varchar(16)  |      |     |            |                |
| exp_date     | date         |      |     | 0000-00-00 |                |
| account_bal  | decimal(9,2) |      |     | 0.00       |                |
+--------------+--------------+------+-----+------------+----------------+
```

References

[1] E. Berk and C. Ananian. JLex: A lexical analyzer generator for Java `http://www.cs.princeton.edu/∼appel/modern/java/JLex/`, 2000.

[2] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[3] Florian Braun, John Lockwood, and Marcel Waldvogel. Reconfigurable Router Modules Using Network Protocol Wrappers, booktitle = Proceedings of Field-Programmable Logic and Applications, address = Belfast, Northern Ireland, month = aug, year = 2001.

[4] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware. Technical Report WU-CS-01-10, Washington University in Saint Louis, Department of Computer Science, June 2001.

[5] Tom Chaney, J. Andrew Fingerhut, Margaret Flucke, and Jonathan S. Turner. Design of a Gigabit ATM Switch. Technical Report WU-CS-96-07, Washington University in Saint Louis, 1996.

[6] G. Davies and S. Bowsher. Algorithms for Pattern Matching. *Software Practice and Experience*, 16(6):575–601, 1986.

[7] R. Franklin, D. Carver, and B. L. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, April 2002.

[8] Roger L. Haskin and Lee A. Hollaar. Operational Characteristics of a Hardware-Based Pattern Matcher. *ACM Transactions on Database Systems*, 8(1):15–40, March 1983.

[9] Edson Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Washington University in Saint Louis, Department of Computer Science, July 6, 2001.

[10] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration. In *Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[11] MessageLabs Intelligence. MessageLabs Monthly Intelligencec Report-May 2003. http://messagelabs.com/binaries/May031.pdf, May 2003.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.

[13] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[14] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, February 2001.

[15] John W. Lockwood, Jon S. Turner, and David E. Taylor. Field Programmable Port Extender (FPX) for Distributed Routing and Queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137–144, Monterey, CA, USA, February 2000.

[16] James Moscola, John Lockwood, Michael Pachos, and Ronald Loui. Implementaion of a Content-Scanning Module for an Internet Firewall. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, April 2003.

[17] MySQL AB. MySQL. `http://www.mysql.com/`.

[18] Martin Roesch. Snort - lightweight intrusion detection for networks. In *13th Administration Conference, LISA'99*, Seattle, WA, November 1999.

[19] David V. Schuehler and John Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware. In *Proceedings of Hot Interconnects 10 (HotI-10)*, Stanford, CA, USA, August 2002.

[20] David V. Schuehler, James Moscola, and John Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System. In *Proceeding of Hot Interconnects 11 (HotI-11)*, Stanford, CA, USA, August 2003.

[21] Sharman Networks. KaZaA. `http://www.kazaa.com/`.

[22] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, CA, USA, April 2001.

[23] SpamAssassin. `http://www.spamassassin.org`.

[24] Todd Sproull, John W. Lockwood, and David E. Taylor. Control and Configuration Software for a Reconfigurable Networking Hardware Platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, April 2002.

[25] R. Stallman. *The Gawk Manual for Gawk Version 2.15*. GNU Press, 1993.

[26] Synplicity. Synplify. `http://www.synplicity.com/`.

[27] David E. Taylor, John W. Lockwood, and Naji Naufel. RAD Module Infrastructure of the Field-programmable Port eXtender (FPX). Technical report, WUCS-01-16, Washington University, Department of Computer Science, July 2001.

[28] The PHP Group. PHP. `http://www.php.net/`.

[29] Ken Thompson. Regular Expression Search Algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[30] Wallace Westfeldt. Internet Reconfigurable Logic for Creating Web-enabled Devices. Xilinx Xcell, Q1 1999.

Vita

James Moscola

Date of Birth	December 14, 1977
Place of Birth	Staten Island, New York
Education	M.S. Computer Science, Washington Univ., August 2003 B.S. Computer Engineering, Washington Univ., May 2001 B.S. Physical Science, Muhlenberg College, May 2000

Publications

James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (Napa, CA, USA), Apr. 2003.

James Moscola, Michael Pachos, John Lockwood, and Ronald P. Loui. FPsed: A Streaming Content Search-and-Replace Module for an Internet Firewall. In *Proceeding of Hot Interconnects 11 (HotI-11)*, (Stanford, CA, USA), Aug. 2003.

David V. Schuehler, James Moscola, and John Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System. In *Proceeding of Hot Interconnects 11 (HotI-11)*, (Stanford, CA, USA), Aug. 2003.

John Lockwood, Chris Neely, Chris Zuver, Dave Lim, and James Moscola. An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall. In *Proceedings of Field-Programmable Logic and Applications (FPL)*, (Lisbon, Portugal), September. 2003.

August 2003