Towards software-based signature detection for intrusion prevention on the network card

H.Bos[†] and Kaiming Huang[*]

[†]Vrije Universiteit, Amsterdam, The Netherlands,
`herbertb@cs.vu.nl`,
[*]Xiamen University, Xiamen, China,
`kmhuang@xmu.edu.cn`

Abstract. *CardGuard* is a signature detection system for intrusion detection and prevention that scans the entire payload of packets for suspicious patterns and is implemented in software on a network card equiped with an Intel IXP1200 network processor. One card can be used to protect either a single host, or a small group of machines connected to a switch. *CardGuard* is non-intrusive in the sense that no cycles of the host CPUs are used for intrusion detection and the system operates at Fast Ethernet link rate. TCP flows are first reconstructed before they are scanned with the Aho-Corasick algorithm.

Key words:distributed firewall, network processors

1   Introduction

Intrusion detection and prevention systems (IDS/IPS) are increasingly relied upon to protect network and computing resources from attempts to gain unauthorised access, e.g., by means of worms, viruses or Trojans. To protect computing resources on fast connections, it is often desirable to scan packet payloads at line rate. However, scanning traffic for the occurrence of attack signatures is a challenging task even with today's networks. Moreover, as the growth of link speed is sometimes said to exceed Moore's law, the problem is likely to get worse rather than better in the future. Worms especially are difficult to stop manually as they are self-replicating and may spread fast. For example, the Slammer worm managed to infect 90% of all vulnerable hosts on the net in just 10 minutes [1].

   In this paper, rather than performing signature scanning at a centralised firewall or on the end-host's CPU, we explore the feasibility of implementing a complete signature detection system (SDS) in software on the network card. The notion of a distributed firewall, proposed by Bellovin in 1999, has gained popularity in recent years [2, 3]. However, most of these systems do not implement payload inspection at all. Recently, Clark et al. proposed to use FPGAs for signature detection [4]. The disadvantage of FPGAs and other hardware solutions is that they are complex to modify (e.g., to change the detection algorithm). For this reason, we explore what rates can be sustained in a software-only solution running in its entirety on a network card equiped with a network processor that

was introduced half a decade ago. This is one of the extremes in the design space of where to perform signature detection and to our knowledge this option has not been explored before.

The resulting SDS, known as *CardGuard*, is intended to protect either a single end-user's host, or a small set of hosts connected to a switch. Throughout this project, our goal has been to make the SDS an inexpensive device with an eye on making it competitive with large firewalls. At the same time, the SDS should be fast enough to handle realistic loads. In this paper, we assume that the bandwidth requirements of individual users do not exceed a few hundred Mbps. We require the system to handle such loads under normal circumstances, i.e., when the number of intrusion attempts compared to regular traffic is reasonably small. In exceptional circumstances, when the system is under *heavy* attack, we consider gradual degradation in performance acceptable. In that case, the integrity of the node is more important than the ability to handle high speeds. Phrased differently, we aim to protect against unwanted content (e.g., intrusion attempts, or spam) and not against denial of service attacks. Finally, we focus on the computationally hard problem of network payload pattern matching, rather than the less compute intensive problems of header inspection and anomaly detection. The latter domain is a well-established field of research whose results can even be found in commercial network equipment [5, 6, 7, 8, 9].

The contributions in this paper fall into several categories. First, we demonstrate that network processors can be used for inspecting every single byte of the payload at realistic rates. Second, in a low-end configuration we present a software-based SDS complete with TCP stream reconstruction and an advanced scanning algorithm (primarily Aho-Corasick [10], although regular expression matching can also be catered to) that scales to thousands of signatures. Third, we employ a novel way of using the memory hierarchy of the Intel IXP network processor to exploit locality of reference in the scanning algorithm.

1.1 Distributing the firewall

Most current approaches to IDS/IPS involve a high-performance firewall/IDS at the edge of the network. All internal nodes are assumed to be safe and all external nodes are considered suspect. The firewall closes all but a few ports and in an advanced system may even scan individual packets for the occurrence of attack patterns. Compared to a distributed firewall, this approach has a number of drawbacks. First, it does not protect internal nodes from attacks originating within the intranet. Once an internal node has been compromised, by whatever means, all nodes in the intranet are at risk.

Second, as the firewall represents the intranet's link to the outside world, the volume of traffic is very large which may render payload scans difficult or even infeasible. Besides speed, managing per-flow state is an issue. Some researchers have suggested that it is difficult to keep per-flow state in the firewall [7]. Others propose to keep per-flow state, but admit that more work on state management is needed (Bro [11] and TRW [12]). Some approaches to attack detection use aggregate behaviour to get around the need to maintain per-flow state [13], or in case of signature detection, limit themselves to per-packet scans rather than full TCP streams [14]. However, attacks may span a number of packets each of which may be harmless in and off itself. Hence, flow reconstruction is a requirement for reliable signature detection in the payload. As packets arrive out of order, this

probably means that the per-flow state now also contains part of the payload. While this is rather expensive at a centralised firewall, it can be easily done at the end-host (e.g., if the firewall is pushed back to the end host).

A third drawback of a centralised firewall, is that it often protects a heterogeneous collection of machines, including webservers, mailservers, databases, workstations running hardly any services at all, etc. In principle, there is no reason to subject traffic to security checks pertaining to a particular vulnerable version of a service, when it is destined for hosts that do not run this service, or that have a patched version of the service. Firewalls at the edge of the intranet have no way of discriminating among the hosts and services that lie behind it. For instance, they don't know whether host X runs the patched or unpatched webserver (or even whether it runs a webserver at all).

A fourth drawback is that centralised firewalls tend to close all ports except a select few, such as those used for webtraffic. As a consequence, we observe that all sorts of new protocols are implemented on top of port 80, defeating the purpose. Another consequence is inconvenience to users that experience problems when using software (e.g., video-conferencing tools) with external parties. While per-host firewall configuration is possible, it is more complex, especially as IP addresses in the intra-net often are not constant.

1.2   The IXP1200 network processor

In the remainder of this paper, we describe *CardGuard* an SDS (and crude IPS) implemented on an IXP1200 network processor unit (NPU). The choice for the IXP1200 was motivated by the fact that it may now be considered yesterday's technology and, hence, potentially cheap. Still, *CardGuard* performs payload scanning at realistic rates, irrespective of the size or number of the patterns. Moreover, the presence of *CardGuard* is transparent to end-applications.

NPUs have emerged in the late 1990s to cope with increasing link speeds. The idea is to push packet processing to the lowest possible level in the processing hierarchy, e.g., before traffic even hits a host's PCI bus. The Intel IXP1200 used for this work contains on-chip one general-purpose StrongARM processor and six independent RISC cores, known as microengines. NPUs have been successfully employed in many network devices, such as routers and monitors. In addition, while previous work has shown that they can be used for other tasks as well [15], there have been few attempts to use them to implement the computationally intensive task of intrusion detection. Often such attempts have been limited to header processing (e.g. [16]). A notable exception is found in [4] which uses the IXP for port filtering and TCP stream reconstruction. The TCP streams are then fed into a string matching engine implemented in hardware (FPGA) on a separate card. In contrast, we use a single IXP1200 to handle all of the above tasks and all processing is in software. Even so, the performance is comparable to the approach with two cards and hardware-based matching. In addition, the system in [4] is not able to protect more than one host.

CardGuard employs the well-known Aho-Corasick algorithm for performing high-performance pattern searches [10]. The same algorithm is used in the latest versions of the Snort intrusion detection tool [14]. In our work, the algorithm runs entirely on the microengines of the network processor. Moreover, as we verified experimentally that Aho-Corasick exhibits locality of reference, *CardGuard* uses a hierarchical memory model where frequently accessed data is in faster memory.

On the surface, *CardGuard* shares some characteristics with what is known as 'TCP offload', i.e., the implementation of TCP protocol processing on the NIC. TCP Offload Engines (TOEs) have recently come under fire, mainly for being a bad match to the application domain for which they are intended, and because TCP processing need not be a very expensive task anyway [17]. While the jury may be out on the merits and demerits of TOEs, we argue that the problem domain for *CardGuard* is very different (e.g., payload scanning is much more expensive than processing TCP headers). If successful, the offloading of full payload pattern matching would be very beneficial indeed. Similarly, whereas TOEs try to alleviate the burden of host processors and in doing so may introduce scalability problems, *CardGuard* is trying to *address* scalability issues caused by performing all intrusion detection at a central point (the firewall). Also, *CardGuard* provides functionality that is not equivalent to that of a centralised firewall, as it also protects hosts from attacks originating in the intranet. Still, it resembles a stand-alone firewall in the sense that traffic is scanned *before* it arrives at a host. As such, it is potentially less dependent on the correct configuration of the end host than a solution where intrusion prevention takes place in the host OS (assuming this were possible at high speeds).

Most importantly perhaps is that this paper explores for the first time one of the extremes in the design space for in-band signature detection: a software-only solution on the NIC. Centralised solutions, implementations on the host processor and even hardware solutions on the NIC have already been studied with some success. *CardGuard* will help developers to evaluate better the different design options.

1.3   Constraints

Programming in a resource-constrained environment so close to the actual hardware is considerably harder than writing equivalent code in userspace. Before we discuss the SDS in detail, we want to point out that we envision our work as a component (albeit an important one) in a full-fledged intrusion prevention system. Although we achieved a fully functional implementation of *CardGuard*, we stress that this work is a research study that explores an extreme solution to intrusion detection rather than a production-grade IPS. Although it is clear that IDSs and IPSs may be more complex than what can be offered by a single tool like snort [14], we aim for functionality that is similar to snort's signature detection. In essence, *CardGuard* explores how much processing can be performed on packet payloads using a cheap software-only solution running entirely on the network card. To make the solution cheap[1], the card is equiped with an Intel IXP1200 which may be considered yesterday's technology. As we deliberately limited ourselves to an instruction store per microengine of just 1K instructions, we are forced to code as efficiently as possible: every instruction is precious. As a consequence, complex solutions like regular expression matching on the chip's microengines are out of the question. Instead, we try to establish (i) a bound on the link rate that can be sustained when the payload of every single packet is scanned for thousands of strings, while (ii) using hardware that is by no means state of the art. All packets corresponding to rules with regular expressions are therefore handled by the on-chip StrongARM processor (using almost the same

---

[1] 'Cheap' refers to cost of manufacturing, not necessarily retail price.

regular expression engine as used by snort). Fortunately, the vast majority of the patterns in current snort rules does not contain regular expressions[2].

Even though *CardGuard* is an SDS and not a complete IDS or IPS, we did configure it as an IPS for testing purposes. In other words, the card automatically generates alerts and drops connections for flows that contain suspect patterns. The resulting IDS/IPS is crude, but this is acceptable for our purposes, as we are interested mainly in the rates that can be sustained with full payload inspection. In the remainder of the paper we sometimes refer to *CardGuard* as an IDS/IPS.

In this paper, we consider only the SDS on the card. The control and management plane for installing and removing rules on the cards is beyond the scope of this paper. We are working on a management plane that allows sysadmins to schedule automatic updates for *CardGuard* (e.g., to load new signatures). These updates require the system to be taken offline temporarily and may therefore best be scheduled during 'quiet hours'. The system itself is modelled after the control architecture for distributed firewalls proposed in [3]. Note that since management traffic also passes through *CardGuard*, the management messages are encoded, to prevent them from triggering alarms.

1.4 Outline

The remainder of this paper is organised as follows. In Section 2 the use of Aho-Corasick in intrusion detection is discussed. Section 3 presents the hardware configuration, while Section 4 provides both an overview of the software architecture as well as implementation details. In Section 5, experimental results are discussed. Related work is discussed throughout the text and summarised in Section 6. Conclusions are drawn in Section 7.

2 SDS and Aho-Corasick

While increasing network speed is one of the challenges in intrusion detection, scalability is another, equally important one. As the number of worms, viruses and Trojans increases, an SDS must check every packet for more and more signatures. Moreover, the signature of an attack may range from a few bytes to several kilobytes and may be located anywhere in the packet payload. Existing approaches that operate at high speed, but only scan packet headers (as described in [16]) are not sufficient. Similarly, fast scans for a small number of patterns will not be good enough in the face of a growing number of threats with unique signatures. While it is crucial to process packets at high rates, it is equally imperative to be able to do so for thousands of signatures, small and large, that may be hidden anywhere in the payload.

For this purpose, *CardGuard* employs the Aho-Corasick algorithm which has the desirable property that the processing time does not depend on the size or number of patterns in a significant way. Given a set of patterns to search for in the network packets, the algorithm constructs a deterministic finite automaton (DFA), which is employed to match all patterns at once, one byte at a time. It is beyond the scope of the paper to repeat the explanation of how the DFA is

---

[2] At the time of writing, less than 300 of the snort rules contain regular expressions, while thousands of rules contain exact strings.

constructed (interested readers are referred to [10]). However, for better understanding of some of the design decisions in *CardGuard*, it is useful to consider in more detail the code that performs the matching.

2.1 Aho-Corasick example

As an example, consider the DFA in Figure 1. Initially, the algorithm is in state 0. A state transition is made whenever a new byte is read. If the current state is 0 and the next byte in the packet is a 'Q', the new state will be 36 and the algorithm proceeds with the next byte. In case this byte is 'Q', 'h', or 't', we will move to state 36, 37, or 43, respectively. If it is none of the above, we move back to state 0. We continue in this way until the entire input is processed. For every byte in the packet, a single state transition is made (although the new state may be the same as the old state). Some states are special and represent output states. Whenever an output state has been reached, we know that one of the signatures has matched. For example, should the algorithm ever reach state 35, this means that the data in the traffic contains the string 'hws2'.

```
State 0,    State 6:      State 12:    State 18:    State 24:    State 30:    State 38:    State 45:
  32,      '.'  :  2     'Q'  : 36    'Q'  : 36    'Q'  : 36    'Q'  : 36    'Q'  : 36    'Q'  : 36
  35,      'Q'  : 36     'e'  : 13    'h'  :  1    'h'  :  1    'T'  : 31    'h'  :  1    'h'  : 46
  42,      'e'  :  7     'h'  :  1    't'  : 43    'k'  : 25    'h'  :  1    'o'  : 39    't'  : 43
  50:      'h'  :  1     't'  : 43    'u'  : 19    'o'  : 44    'h'  :  1    't'  : 43   State 46:
'Q'  : 36  't'  : 43    State 13:    State 19:    't'  : 43    't'  : 43   State 39:   '.'  :  2
'h'  :  1  'w'  : 33    'Q'  : 36    'Q'  : 36    State 25:    State 31:   'Q'  : 36   'Q'  : 36
't'  : 43  State 7:     'h'  :  1    'h'  :  1    'C'  : 26    'Q'  : 36   'c'  : 40   'h'  :  1
State 1:   'Q'  : 36    'r'  : 14    'n'  : 20    'Q'  : 36    'f'  : 32   'h'  :  1   's'  : 47
'.'  :  2  'h'  :  1    't'  : 43    't'  : 43    'h'  :  1    'h'  :  1   't'  : 43   't'  : 43
'Q'  : 36  'l'  :  8    State 14:    State 20:    'h'  :  1    't'  : 43   State 40:   'w'  : 33
'h'  :  1  't'  : 43    'Q'  : 36    'Q'  : 36    't'  : 43    State 33:   'Q'  : 36   State 47:
't'  : 43  State 8:     'h'  :  1    'h'  :  1    State 26:    'Q'  : 36   'h'  :  1   'Q'  : 36
'w'  : 33  '3'  :  9    'n'  : 15    't'  : 21    'Q'  : 36    'h'  :  1   'k'  : 41   'e'  : 48
State 2:   'Q'  : 36    't'  : 43    State 21:    'h'  : 27    's'  : 34   't'  : 43   'h'  :  1
'Q'  : 36  'h'  :  1    State 15:    'Q'  : 36    't'  : 43    't'  : 43   State 41:   'o'  : 39
'd'  :  3  't'  : 43    'Q'  : 16    'h'  :  1    State 27:    State 34:   'Q'  : 36   't'  : 43
'h'  :  1  State 9:     'h'  :  1    'o'  : 44    '.'  :  2    '2'  : 35   'f'  : 42   State 48:
't'  : 43  '2'  : 10    't'  : 43    't'  : 43    'G'  : 28    'Q'  : 36   'h'  :  1   'Q'  : 36
State 3:   'Q'  : 36    State 16:    State 22:    'Q'  : 36    'h'  :  1   't'  : 43   'h'  :  1
'Q'  : 36  'h'  :  1    'Q'  : 36    '.'  :  2    'h'  :  1    't'  : 43   State 43:   'n'  : 49
'h'  :  1  't'  : 43    'h'  : 17    'Q'  : 36    't'  : 43    State 36:   'Q'  : 36   State 49:
'l'  :  4  State 10:    't'  : 43    'h'  :  1    'w'  : 33    'Q'  : 36   'h'  :  1   'Q'  : 36
't'  : 43  'Q'  : 36    State 17:    'i'  : 23    State 28:    'h'  :  1   'o'  : 44   'd'  : 50
State 4:   'h'  : 11    '.'  :  2    't'  : 43    'Q'  : 36    't'  : 43   't'  : 43   'h'  :  1
'Q'  : 36  't'  : 43    'Q'  : 36    'w'  : 33    'h'  :  1    State 37:   State 44:   't'  : 43
'h'  :  1  State 11:    'h'  :  1    State 23:    't'  : 43    '.'  :  2   'Q'  : 45
'l'  :  5  '.'  :  2    'o'  : 18    'Q'  : 36    State 29:    'Q'  : 36   'h'  :  1
't'  : 43  'Q'  : 36    's'  : 38    'c'  : 24    'Q'  : 36    'h'  :  1   't'  : 43
State 5:   'h'  :  1    't'  : 43    'h'  :  1    'h'  :  1    's'  : 38
'Q'  : 36  'k'  : 12    'w'  : 33    't'  : 43    't'  : 30    't'  : 43
'h'  :  6  't'  : 43                                          'w'  : 33
't'  : 43  'w'  : 33
```

- Depicted above is the deterministic finite automaton for the following signatures:
  "h.dllhel32hkernQhounthickChGetTf", "hws2", "Qhsockf", "toQhsend", and "Qhsoc"
- Matches are found in the following states (indicated in the table by '▉'):
  {32,"h.dllhel32hkernQhounthickChGetTf"}, {35,"hws2"}, {40,"Qhsoc"},
  {42,"Qhsockf"}, {50,"toQhsend"}

Fig. 1. Deterministic finite automaton for Slammer worm

The DFA in Figure 1 is able to match the five different patterns at the same time. The patterns, shown beneath the figure, are chosen for illustration purposes, but the first four also represent the patterns that make up the real signature of the Slammer worm [1]. This worm was able to spread and infect practically every susceptible host in thirty minutes by using a buffer overflow

exploit in Microsoft SQL Server allowing the worm to execute code on remote hosts. The fifth pattern was only added to show what happens if patterns partly overlap and has no further meaning.

If the initial state is 0 and the input stream consists of these characters: `XYZQQhsockfA`, we will incur transitions to the following states: *0, 0, 0, 36, 36, 37, 38, 39, 40, 41, 42,* and *0.* The underlined states represent output states, so after processing the input sequence we know that we have matched the patterns `Qhsoc` and `Qhsockf`. By making a single transition per byte, all present patterns contained in the packet are found.

As an aside, we extended the Aho-Corasick algorithm in order to make it recognise rules that contain multiple strings (e.g., a rule that fires only when the data contains both strings $S1$ and $S2$). Unfortunately, there is no space in the IXP1200's instruction store to add this functionality. We recently implemented it on an IXP2400. In our view, it serves to demonstrate the advantages of a software-only approach. The port of the original code and its extension was straightforward. A similar upgrade of an FPGA-based solution would require substantially more effort.

2.2 Observations

The following observations can be made. First, the algorithm to match the patterns is extremely simple. It consists of a comparison, a state transition and possibly an action when a pattern is matched. Not much instruction memory is needed to store such a simple program. Second, the DFA, even for such a trivial search, is rather large. There are 51 states for 5 small, partly overlapping patterns, roughly the combined number of characters in the patterns. For longer scans, the memory footprint of the Aho-Corasick algorithm can grow to be fairly large. Recent work has shown how to decrease the memory footprint of the algorithm [18]. However, this approach makes the algorithm slower and is therefore not considered in this paper. Third, as far as speed is concerned, the algorithm scales well with increasing numbers of patterns and increasing pattern lengths. Indeed, the performance is hardly influenced by these two factors, except that the number of matches may increase with the number of patterns, in which case the actions corresponding to matches are executed more frequently. Fourth, parallelism can be exploited mainly by letting different processors handle different packets. There is little benefit in splitting up the set of patterns to search for and letting different processors search for different patterns in the same packet. Fifth, when a traffic scan is interrupted, we only need to store the current state number, to be able to resume at a later stage, i.e., there is no need to store per-pattern information.

3 Hardware

CardGuard is implemented entirely on a single Intel IXP1200 NPU board (shown in Figure 2(a)). The IXP1200 used in *CardGuard* runs at a clockrate of 232 MHz and is mounted on a Radisys ENP2506 board with 8 MB of SRAM and 256 MB of SDRAM. The board contains two Gigabit ports ①. Packet reception and packet transmission over these ports is handled by the code on the IXP1200 ②. The Radisys board is connected to a Linux PC via a PCI bus ③.
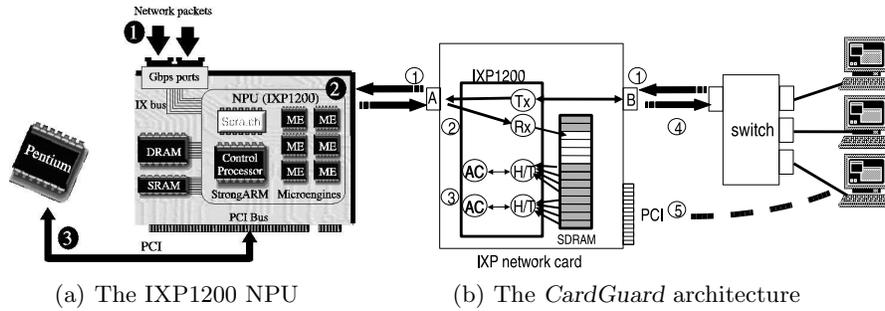
(a) The IXP1200 NPU   (b) The *CardGuard* architecture

Fig. 2. Architecture

The IXP1200 chip itself consists of a StrongARM processor running embedded Linux and 6 independent RISC processors, known as microengines. Each microengine has a 1K instruction store and 128 general-purpose registers, in addition to special purpose registers for reading from and writing to SRAM and SDRAM. On each of the microengines, the registers are partitioned between 4 hardware contexts or 'threads'. Threads on a microengine share the 1K instruction store, but have their own program counters and it is possible to context switch between threads at zero cycle overhead. On-chip the IXP has a small amount (4KB) of scratch memory. Approximate access times of scratch, SRAM and SDRAM are 12-14, 16-20 and 30-40 cycles, respectively. Instruction store and registers can be accessed within a clock cycle. The network processor is connected to the network interfaces via a fast, proprietary bus (the IX bus). Note that a newer version of the IXP architecture, the IXP2800, supports no fewer than 16 microengines (with 8 threads each), has 16KB of scratch memory and operates at 1.4 GHz. This illustrates that the results in this paper represent what can be achieved with yesterday's technology.

As shown in Figure 2(b), one of the Gigabit ports in *CardGuard* (port A) connects to the outside world, while the other (port B) is connected to the switch. The Gigabit ports are used for all data traffic between the hosts and the NPU. In addition to the Gigabit *datapath*, there also exists a *control* connection between the host processor and the IXP1200, which in the implementation on the ENP2506 consists of messages sent across the PCI bus of the machine hosting the NPU. The thick dashed line in Figure 2(b) indicates that it is quite permissible to plug the board in the PCI slot of one of the end hosts, making *CardGuard* a rather low-cost solution in terms of hardware. In the latter case, the same PCI bus is used to transport *CardGuard* control messages and user traffic.

CardGuard is designed as a plug-and-play intrusion detection system. To protect a set of hosts connected to a switch as depicted in Figure 2(b), all that is required is that *CardGuard* is placed on the datapath between the switch and the outside world. No reconfiguration of the end-systems is necessary.

4 Software architecture

In Figure 2(b), the numbering indicates the major components in *CardGuard* packet processing. Since the system is designed as a firewall, both inbound and outbound traffic must be handled. By default, outbound traffic is simply forwarded, but inbound traffic is subjected to full payload scans. In case outbound

traffic should be checked also (e.g., for containment) the performance figures of Section 5 drop by a factor of two. *CardGuard* aims to perform as much of the packet processing as possible on the lowest levels of the processing hierarchy, i.e., the microengines.

A single microengine (ME_{tx}) is dedicated to forwarding and transmission. In other words, ME_{tx} is responsible not only for forwarding all outbound traffic ①, but also for transmitting inbound traffic toward the switch ④. All four threads on ME_{tx} are used, as each of the two tasks is handled by two threads.

A second microengine, ME_{rx}, is dedicated to inbound packet reception ②. It consists of two threads that place incoming packets in the fixed-sized slots of a circular buffer. While we use fixed-sized slots, there may be more than one packet in a slot. ME_{rx} keeps placing packets in a slot, as long as the full packets fit. The motivation for this design is that a buffer with fixed-sized slots is easy to manage and partition, but may suffer from low slot utilisation for short messages. By filling slots with multiple packets resource utilisation is much better. At this microengine we also detect whether packets belong to a stream that needs to be checked by a rule that requires regular expression matching. If so, it is placed in a queue for processing by the StrongARM. The StrongARM is responsible for processing the packet and, if needed, putting it back in the queue for processing by the microengines.

The remaining four microengines (denoted by $ME_{h/t}$ and ME_{ac}, respectively) are dedicated to TCP flow handling and intrusion detection ③. The idea is to use these microengines to scan the packet payloads with the Aho-Corasick algorithm. Parallelising the scanning process in this way is in line with the fourth observation in Section 2.2. However, we now show that things are more complicated.

Attacks may span multiple packets. We should provide a means to handle the case that the signature of a worm starts in one packet and continues in the next. In order to perform meaningful intrusion detection, we cannot avoid TCP stream reconstruction. Worms may span a number of TCP segments which may or may not arrive out of order. As a consequence, we need to keep segments in memory while some earlier segments are still missing. We also need to keep track of sequence numbers and connection state. We developed a light-weight implementation of TCP stream reconstruction for microengines, which we will discuss in section 4.2. *CardGuard* handles 'fragroute'-style attacks (sending duplicate TCP segments with older TCP sequence numbers that *overwrite* previous segments) by dropping segments with sequence numbers that have already been handled. A configurable parameter determines how large the gaps may be in case of out-of-order segment arrival. If the gap grows beyond the maximum size, the connection is conservatively dropped.

Despite our attempts to minimise *CardGuard*'s footprint, the code required to handle both TCP flow reconstruction and pattern matching exceeds the size of an individual microengine's instruction store. As a consequence, we are forced to spread TCP flow hashing and reconstruction (discussed in Section 4.2) on the one hand, and Aho-Corasick pattern matching (discussed in Section 4.3) on the other, over two pairs of tightly-coupled microengines (shown in Figure 2(b) as `H/T` and `AC`, respectively). Given sufficient instruction store, `H/T` and `AC` would be combined on the same microengine (yielding four microengines to perform pattern matching rather than the two that are used in *CardGuard*). All four threads in both `AC` and `H/T` microengines are used.

The ability to 'sanitise' protocols before scanning the data for intrusion attempts is similar to the *protocol scrubber* [19] and *norm* [20], except that it was implemented in a much more resource-constrained environment.

4.1 TCP vs. UDP

By default, all traffic that is not TCP (e.g., UDP) is handled by inspecting the individual packets in isolation and is considered relatively 'easy'. As a result, signatures hidden in multiple packets ('UDP flows') will not be detected in the default configuration. If needed, however, the UDP packets may be treated in the same way as TCP flows are handled. In that case, we lose the performance advantage that UDP holds over TCP (see Section 5). As our focus is on the harder case of TCP flows, which also covers all difficulties found in UDP, we will not discuss non-TCP traffic except in the experimental evaluation.

In the current implementation, the ENP's PCI interface is used for control purposes ⑤. In other words, it is used for bootstrapping the system, loading the ME_{ac} microengines and reading results and statistics. In our test configuration, *CardGuard* is plugged in one of the PCs that it monitors. Although such a setup in which the host appears to be both 'in front of' and 'behind' the firewall may seem a little odd, it does not represent a security hole as *all* inbound traffic still traverses the packet processing code in the ME_{ac} microengines.

CardGuard attempts to execute the entire runtime part of the SDS on the microengines. The only exception is the execution of regular expression matching which takes place on the StrongARM. Each thread in the combination of $ME_{h/t}$ and ME_{ac} processes a unique and statically determined set of packet slots (as will be explained shortly). The fact that a slot may contain multiple small packets, or a single maximum-sized packet offers an additional advantage besides better buffer utilisation, namely load-balancing. Without it, a situation may arise that thread A processes a number of slots each containing just a single minimum-sized packet, while thread B finds all its slots filled with maximum-sized packets. By trying to fill all the slots to capacity, this is less likely to happen.

After flow reconstruction, ME_{ac} applies the Aho-Corasick algorithm to its packets while taking care to preserve the flow order. As we configured *CardGuard* as an IPS, the microengine raises an alarm and drops the packet as soon as a pattern is matched. Otherwise, a reference to the packet is placed in the transmit FIFO and transmitted by ME_{tx}. When processing completes, buffers are marked as available for re-use.

4.2 Resource mapping

Taking into account the hardware limitations described in Section 3 and the observations about the Aho-Corasick algorithm in Section 2.2, we now describe how data and code are mapped on the memories and processing units, respectively.

Packet transmission Two threads on ME_{tx} are dedicated to the task of forwarding outbound traffic from port B to port A. The other two threads transmit packets from SDRAM to port B by monitoring a circular FIFO containing references to packets that passed the ME_{ac} checks. The FIFO is filled by the processing threads on the ME_{ac} microengines.
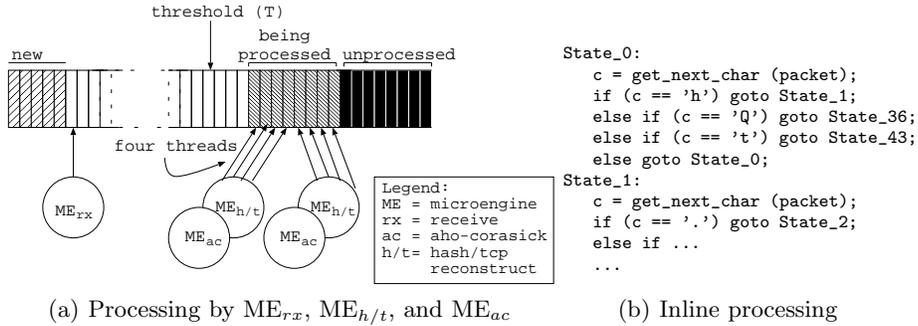
```
State_0:
    c = get_next_char (packet);
    if (c == 'h') goto State_1;
    else if (c == 'Q') goto State_36;
    else if (c == 't') goto State_43;
    else goto State_0;
State_1:
    c = get_next_char (packet);
    if (c == '.') goto State_2;
    else if ...
    ...
```

(a) Processing by ME_{rx}, $ME_{h/t}$, and ME_{ac} (b) Inline processing

Fig. 3. Packet processing

Packet reception Packet processing of inbound traffic is illustrated in Figure 3(a). We take the usual approach of receiving packets in SDRAM, and keeping control structures in SRAM and Scratch. Assuming there is enough space, ME_{rx} transfers the packets to a circular buffer, and keeps a record of the read and write position, as well as a structure indicating the validity of the contents of the buffer in SRAM. Using this structure, an ME_{ac} processing packets may indicate that it is done with specific buffers, enabling ME_{rx} to reuse them.

The exact way in which the buffers are used in *CardGuard* is less common. The moment an in-sequence packet is received and stored in full in SDRAM by ME_{rx}, it can be processed by the processing threads. However, the processing has to be sufficiently fast to prevent buffer overflow. A buffer overflow is not acceptable, as it means that packets are dropped. We have designed the system in such a way that the number of per-packet checks is minimised, possibly at the expense of efficient buffer usage. Whenever ME_{rx} reaches the end of the circular buffer and the write index is about to wrap, ME_{rx} checks to see how far the packet processing microengines have progressed through the buffer. In *CardGuard* the slowest thread should always have progressed beyond a certain threshold index in the buffer (T in Figure 3(a)). *CardGuard* conservatively considers all cases in which threads are slow as system failures, which in this case means that *CardGuard* is not capable of handling the traffic rate.

As both the worst-case execution time for the Aho-Corasick algorithm (the maximum time it takes to process a packet), and the worst-case time for receiving packets (the minimum time to receive and store a packet) are known, it is not difficult to estimate a safe value for the threshold T for a specific rate R and a buffer size of B slots. For simplicity, and without loss of generality, assume that a slot contains at most one packet. For the slowest thread, the maximum number of packets in the buffer at wrap time that is still acceptable is $(B - T)$. If the worst-case execution time for a packet is A, it may take $A(B-T)$ seconds to finish processing these packets. The time it takes to receive a minimum-size packet of length L at rate R is (L/R), assuming ME_{rx} is able to handle rate R. An overflow occurs if $(TL/R) \leq A(B-T)$, so $T = (RAB)/(L+RA)$. For $L = 64$ bytes, $R = 100$ Mbps, $B = 1000$ slots, and $A = 10\mu s$, a safe value for T would be 661.

The threshold mechanism described above is overly conservative. Threads that have not reached the appropriate threshold when ME_{rx} wraps may still

catch up, e.g., if the remaining packets are all minimum-sized, or new packets are big, and do not arrive at maximum rate). Moreover, it is possible to use threads more efficiently, e.g., by not partitioning the traffic, but letting each thread process the 'next available' packet. We have chosen not to do so, because these methods require per-packet administration for such things as checking whether (a) a packet is valid, (b) a packet is processed by a thread, and (c) a buffer slot is no longer in use and may be overwritten. Each of these checks incurs additional overhead. Instead, *CardGuard* needs a single check on eight counters at wrap time.

Packet processing Each of the two $\text{ME}_{h/t}$-ME_{ac} microengine pairs is responsible for processing half of the packets. $\text{ME}_{h/t}$ is responsible for sanitising the TCP stream, while ME_{ac} handles pattern matching.

For TCP flow identification, we use a hash table. The hash table contains a unique entry for each TCP flow, which is generated by employing the IXP's hardware assist to calculate a hash over the segment's source and destination addresses and the TCP ports. A new entry is made whenever a TCP SYN packet is received. The number of flows that may hash to the same hash value is a configurable parameter `HashDim`. If a new flow hashes to an index in the table which already contains `HashDim` flows, the new flow is conservatively dropped.

As a result, every live TCP flow has a hash table entry, which records the following information about the flow: source IP address, destination IP address, source port, destination port, next sequence number, and current DFA state. As logically contiguous segments might be dispatched to different packet processing threads, the next sequence number ensures that segments are pattern-matched in order, while keeping track of the current DFA state facilitates the resumption of pattern matching of a subsequent packet at a later stage (e.g., by another pkt-processing thread). As explained in the fifth observation of Section 2.2, we only need the current DFA state to resume scanning exactly where we left off.

When a non-SYN packet is received, the corresponding hash entry is found and the sequence number of the packet is compared to the sequence number in the table. As explained earlier, we do not permit segments to overwrite segments that were received earlier. Any packet that is not the immediate successor to the stored sequence number is put 'on-hold'. There are two possible schemes for dealing with such segments with which we have experimented. The first, and simplest, is to wait until all missing segments have arrived and only then perform pattern matching. The second is to scan the segment for worms as an individual packet and if it is considered safe, forward it to its destination, while also keeping a copy in memory. Then, when the missing segments arrive, (part of) the segment is scanned again for all signatures that may have started in the segments preceding it and overlap with this segment. This is safe, even if the segments that were forwarded were part of a worm attack. The reason is that these packets *by themselves* do not constitute an attack. Only the addition of the preceding packets would render the 'worm' complete. However, as the attack is detected when the preceding packets arrive, these segments are never forwarded.

In the current implementation, a hash table entry is removed only as a result of an explicit tear-down. The assumption that motivates this choice is that the FIN and RST messages coming from the downstream host are never lost. How-

ever, in the future we expect to incorporate a time-out mechanism that frees up the hash-table entry while dropping the connection.

Also note that when *CardGuard* is started, all flows that are currently active are by necessity dropped, as they will not have hash entries in the new configuration. Recently we have started implementing a mechanism that preserves the original hash table.

Pattern matching For pattern matching purposes, a thread on each ME_{ac} reads data from SDRAM in 8-byte chunks and feeds it, one byte at a time, to the Aho-Corasick algorithm. However, as the memory latency to SDRAM is in the range of 33 to 40 cycles, such a naive implementation would be prohibitively slow [21]. Therefore, in order to hide latency, *CardGuard* employs four threads. Whenever a thread stalls on a memory access, a zero-cycle context switch is made to allow the next processing thread to resume. As there are now eight packet processing threads in *CardGuard*, the buffer is partitioned such that thread t is responsible for slots $t, t + 8, t + 16, \ldots$

4.3 The memory hierarchy

We have explained in Section 3 that the IXP1200 has various types of memories with different speeds and sizes: registers, instruction store, scratch, SRAM and SDRAM. Optimising the use of these memories proved to be key to *CardGuard*'s performance. For instance, as *CardGuard* needs to access the DFA for every byte in every packet, we would like the DFA to be stored in fast memory. However, there are relatively few general purpose registers (GPRs) and scratch is both small and relatively slow. Moreover, these resources are used by the compiler for local variables as well.

For this reason, we make the following design decisions: (1) GPRs and scratch are not used for storing the DFA, (2) instead, we exploit unused space in the *instruction store* for storing a small part of the DFA, (3) another, fairly large, part is stored in the 8 MB of SRAM, and (4) the remainder of the DFA is stored in the 256 MB of slow SDRAM.

The idea is that, analogous to caching, a select number of frequently accessed states are stored in fast memory, while the remainder is stored in increasingly slow memory[3]. A premise for this to work, is that the Aho-Corasick algorithm exhibits strong locality of reference. Whether this is the case depends both on the patterns and on the traffic. Defining level n in the DFA as all states that are n transitions away from state 0, we *assume* for now that the top few levels in the DFA, (e.g., states 0, 1, 36 and 43 in Figure 1) are accessed much more frequently than the lower levels. In Section 5, we present empirical evidence to support this.

Using the instruction store and 'normal memories' for storing the DFA, leads to two distinct implementations of the Aho-Corasick algorithm itself, which we refer to as 'inline' and 'in-memory'. In an inline implementation, a DFA like the one sketched in Figure 1 is implemented in the instruction store of a ME_{ac}, e.g., as a set of comparisons and jumps as illustrated in pseudo-code in Figure 3(b).

In-memory implementations, on the other hand, keep the DFA itself separate from the code by storing it in one of the memories. The data structure that is

---

[3] It is not a real cache, as there is no replacement.

commonly used to store DFAs in Aho-Corasick is a 'trie' with pointers from a source state to destination states to represent the transitions. In this case, a state transition is expensive since memory needs to be accessed to find the next state. The overhead consists not only of the 'normal' memory latency, as additional overhead may be incurred if these memory accesses lead to congestion on the memory bus. This will slow down *all* memory accesses.

Note that each state in the inline implementation consists of several instructions and hence costs several cycles. We are able to optimise the number of conditional statements a little by means of using the equivalent of 'binary search' to find the appropriate action, but we still spend at least a few tens of cycles at each state (depending on the exact configuration of the DFA and the traffic). However, this is still far better than the implementation that uses SRAM, as this requires several slow reads (across a congested bus), in addition to the instructions that are needed to execute the state transitions.

In spite of the obvious advantages, the inline version can only be used for a small portion of the DFA, because of the limited instruction store of the microengines. *CardGuard* is designed to deal with possibly thousands of signatures, and the instruction store is just 1K instructions in size, so locality of reference is crucial. In practice, we are able to store a few tens of states in the unused instruction store, depending on the number of outgoing links. In many cases, this is sufficient to store the most commonly visited nodes. For instance, we are able to store in their entirety levels 0 and 1 of the 2025 states of snort's web IIS rules. In our experiments these levels offer hit rates of the order of 99.9%. In Section 5, we will analyse the locality of reference in Aho-Corasick in detail.

One may wonder whether, given 8 MB of SRAM, SDRAM is ever needed for storing the DFA. Surprisingly, the answer is yes. The reason is that we sacrifice memory efficiency for speed. For instance, if we combine all of snort's rules that scan traffic for signatures of at least ten bytes, the number of states in the DFA is roughly 15k. For each of these states, we store an array of 256 words, corresponding to the 256 characters that may be read in the next step. The array element for a character c contains the next state to which we should make a transition if c is encountered. The advantage is that we can look up the next state by performing an offset in an array, which is fast. The drawback is that some states are pushed to slow memory. Whether this is serious again depends on how often reads from SDRAM are needed, i.e., on the locality of reference.

The partitioning of the DFA over the memory hierarchy is the responsibility of *CardGuard*. The amount of SRAM and SDRAM space dedicated to DFA storage is a configurable parameter. For the instruction store, there is no easy way to determine how many states it can hold *a priori*. As a consequence, we are forced to use iterative compilation of the microengine code. At each iteration, we increase the number of states, and we continue until the compilation fails because of 'insufficient memory'.

4.4 Alerts and intrusion prevention

When a signature is found in a packet, it needs to be reported to the higher-levels in the processing hierarchy. For this purpose, the code on the microengines writes details about the match (what pattern was matched, in which packet), in a special buffer in scratch and signals code running on the StrongARM. In addition, it will drop the packet and possibly the connection. The StrongARM

code has several options: it may take all necessary actions itself, or it may defer the processing to the host processor. The latter solution is the default one.

4.5   Control and management

The construction of the Aho-Corasick DFA is done offline, e.g., on the host connected to the IXP board. The DFA is subsequently loaded on the IXP. If the inline part of the Aho-Corasick algorithm changes, this process is fairly involved as it includes stopping the microengine, loading new code in the instruction store, and restarting the microengine. All of this is performed by control code running on the StrongARM processor. In the current implementation, this involves restarting all microengines, and hence a short period of downtime.

5   Evaluation

CardGuard's use of the memory hierarchy only makes sense if there is sufficient locality of reference in the Aho-Corasick algorithm when applied to actual traffic. Figure 4(a) shows how many times the different levels in the Aho-Corasick DFA are visited for a large number of different rule sets for a 40 minute trace obtained from a set of 6 hosts in Xiamen University. We deliberately show a short trace to avoid losing in the noise short-lived fluctuations in locality. More traces (including longer-lived ones) are maintained at `www.cs.vu.nl/~herbertb/papers/ac_locality`. Every class of snort rules of which at least one member applied pattern matching (with a signature length of at least ten characters, to make it interesting) was taken as a separate rule set. We used the current snapshot of snort rules available at the time of writing (September 2004). In total there were 22 levels in the DFA, but the number of hits at levels 6-22 is insignificant and has been left out for clarity's sake. The figure shows the results for hundreds of rule sets with thousands of rules. The line for the combination of all of snort's rules is explicitly shown. The remaining lines show the locality for each of snort's rule types (e.g., web, viruses, etc.). Since there are a great many categories, we do not name each separately. To provide a thorough evaluation of Aho-Corasick in signature detection, we have performed this experiment in networks of different sizes (e.g., one user, tens, hundreds, and thousands of users), for different types of users (small department, university campus, nationwide ISP) and in three different countries. The results show clear evidence of locality. The plot in Figure 4(a) is typical for all our results.

It may be countered that these results were not obtained while the network was 'under heavy attack' and that the plots may look very different then. While true, this is precisely the situation that we want to cater to. When the network is so much under attack that locality of reference no longer holds, degrading network performance is considered acceptable. Probably the network is degrading anyway, and we would rather have a network that is somewhat slower than an infected machine.

One of the problems of evaluating the *CardGuard* implementation is generating realistic traffic at a sufficient rate. In the following, all experiments involve a DFA that is stored both inline and in-memory. As a first experiment we used `tcpreplay` to generate traffic from a trace file that was previously recorded on our network. Unfortunately, the maximum rate that can be generated with

`tcpreplay` is very limited, in the order of 50Mbps. At this rate, *CardGuard* could easily handle the traffic (even when we did not store any states in instruction-store whatsoever).

As a second experiment, we examined the number of cycles that were needed to process packets of various sizes. The results are shown in Table 1. These speeds suggest that a single thread could process approximately 52.5 Mbps for maximum-sized non-TCP packets. By gross approximation, we estimate that with eight processing threads this leads to a throughput of roughly 400 Mbps (accounting neither for adverse effects of memory stalls, or beneficial effects from latency hiding). We show that in reality we perform a little better.

The penalty for in-memory DFA transitions, compared to inline transitions is shown in Table 2. The table lists the number of cycles needed for ten state transitions in the DFA. For inline and in-memory we measure the results both when the packet is still in SDRAM, and when the packet data is already in read registers on-chip. The difference is 70-80 cycles. The table shows that in-memory state transitions are approximately twice as expensive as inline ones. One might expect that this also results in a maximum sustainable rate for a completely in-memory implementation that is half of that of a completely inline implementation, but this is not the case, as memory latency hiding techniques with multiple threads is quite effective.

packet size (bytes)	cycles
64	976
300	9570
600	20426
900	31238
1200	42156
1500	53018

Table 1. Cycles required to process a packet

Our final experiment is a stress-test in which we blast *CardGuard* with packets sent by `iperf` (running for 3 minutes) from a 1.8GHz P4 running a Linux 2.4 kernel equiped with a SysKonnect GigE interface. We evaluate the throughput that *CardGuard* achieves under worst case conditions. Worst case means that the payload of every single packet needs to be checked from start to finish. This is not a realistic scenario, as snort rules tend to apply to a single protocol and one destination port only. For example, it makes no sense to check web rules for non-webtraffic. In this experiment, we deliberately send traffic of which each packet is checked in its entirety. The assumption is that if we are able to achieve realistic network speeds under these circumstances, we will surely have met the requirements defined in Section 1.

Figure 4(b) shows the throughput achieved for various types of traffic and actions (median values over a series of runs). Note that as long as the network is not under heavy attack, the results are hardly influenced by which rule sets and traces are used, due to

memory type used	cycles
instruction store, pkt access in register	330
same, with pkt access in SDRAM	410
SRAM, pkt access in register	760
same, with pkt access in SDRAM	830

Table 2. Cycles to make ten transitions

the locality of reference observed earlier. The top line shows the throughput when all UDP packets are checked and forwarded (with the two $ME_{h/t}$ microengines turned into ME_{ac} microengines), so for UDP we achieve maximum throughput. The line below UDP shows the throughput of TCP, when TCP segments are simply forwarded, but not checked. The reason why it performs fairly poorly, compared to UDP is that the traffic generator was the bottleneck, not *CardGuard* (`rude` was used as UDP generator). Finally, the most important line

is the bottom line, which shows the maximum throughput for TCP when the full streams are reconstructed and the entire stream is scanned. We conclude that *CardGuard* meets the requirement of handling 100 Mbps under worst case assumptions.

CardGuard adds one additional feature: it is able to limit the number of incoming and outgoing connections. The default configuration is that ten inbound and ten outbound connections are premitted. With this configuration, we are able to sustain the maximum rates at worst-case conditions. More flows are possible, but in that case the aggregate rate drops (the system tops out at 100 Mbps for ≤ 10 connections). This is unlikely to be a problem for most applications, but some (e.g., some peer-to-peer clients) may suffer from reduced bandwidth. In our view, ten is a reasonable choice for most end-user set-ups, because end-user systems are not expected to have many connections open at the same time. For servers these numbers may be increased. As systems in practice do not work under such extreme conditions where every packet needs a full scan, we expect to sustain high rates even with larger numbers of connections. Note that *Card-Guard* always remains on the conservative side. If the rate cannot be supported, the packets are dropped. In other words, it does not suffer from false negatives.
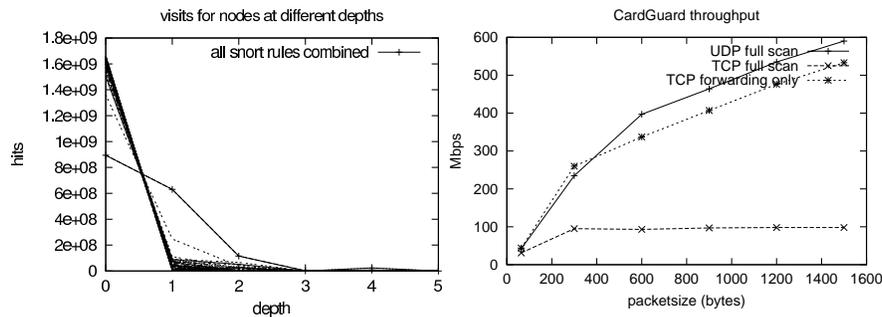

Fig. 4. (a) locality in Aho-Corasick, and (b) *CardGuard*' UDP and TCP throughput

Discussion: network processing on the card

While we have shown that *CardGuard* performs well, despite its five year old network processor, we now return to two questions that were touched upon in Section 1: (1) whether it is necessary to perform signature detection on the card, and if so, (2) whether a network processor is the most appropriate choice. Alternative approaches may be a centralised firewall with payload scanning, signature detection on the end node's main processor, and detection on the card with a different processor (e.g., an ASIC, FPGA, or full-blown CPU).

Technically, it may be difficult to perform payload scanning at high rates in a centralised firewall. While the CPU of a modern PC probably *is* powerful enough to scan an end user's traffic, we should bear in mind that network speed grows harder than Moore's law. Furthermore, the first question has as much to do with policies and politics as with technology. A programmable network card is remote from the user and hence simpler to protect from manipulation. Moreover, while host processors may be fast enough to perform signature detection,

doing so consumes many cycles. When we evaluated the same signature detection algorithm on a 1.8 GHz P4 (Linux 2.4 kernel) equiped with a SysKonnect GigE card, we were unable to achieve rates greater than 69 Mbps. Note that this is at a clock rate that is 8 times higher than that of the IXP1200. Over the loopback device we were able to achieve significantly more than 100 Mbps, but only at the cost of high CPU loads that leave few cycles for useful work. Breaking down the overhead, we found that the signature detection algorithm consumes over 90% of the processing time, suggesting that perhaps it is a better candidate for off-loading than TCP is for TOEs.

The second question concerns whether a network processor is the best choice on the card. ASICs and FPGAs are attractive alternatives in terms of speed. On the other hand, they are more complex to modify. Additionally, compared to C programmers, VHDL/Verilog programmers are scarce. In contrast, we experienced at first hand how simple it was to modify an older version of *CardGuard* for students with only experience in C programming. While the same would be true *a fortiori* for general purpose processors, installing such a processor on a NIC is probably overkill and requires more extensive cooling as it needs to run at higher frequencies to keep up.

More importantly, this paper is meant to explore the design space by studying the feasibility of one the extremes in the design space: a software-only solution on the NIC. While the other approaches have been studied previously, to the best of our knowledge, this is the first time anyone has explored this extreme.

6   Related work

According to the taxonomy in [22], our work would be categorised as a knowledge-based IDS with an active response based on the packet scanning using continuous monitoring with state-based detection. As such it differs from (a) passive systems like HayStack [23], (b) approaches that use network traffic statistics like GrIDS [24], (c) transition-based approaches like Netranger [25], (d) periodic analyzers like Satan [26]. In the terminology of [27], *CardGuard* is a 'containment' solution, which the authors identify as the most promising approach to stop self-propagating code. Unlike passive systems, *CardGuard* does not exhibit the 'fail-open' flaw identified in most existing IDSs in [28]. Since the IDS/IPS *is* the forwarding engine, there is no way to bypass it.

CardGuard is more static than the IDS approach advocated in [29] which suggests that the IDS should be adaptive to the environment. In *CardGuard* this is not an option, as all capacity is fully used.

The use of sensors in the OS kernel for detecting intrusion attempts [30] also adds a light-weight intrusion detection system in the datapath. An important difference with *CardGuard* is that it requires a reconfiguration of the kernel and is therefore OS-specific.

A well-known IDS is Paxson's Bro [11]. Compared to *CardGuard*, Bro gives more attention to event handling and policy implementation. On the other hand it counts over 27.000 lines of C++ code and is designed to operate at a very high level (e.g., on top of `libpcap`). It relies on policy script interpreters to take the necessary precautions whenever an unusual event occurs. In contrast, *CardGuard* sits at a very low-level and takes simple, but high-speed actions whenever it detects a suspicious pattern.

The Aho-Corasick algorithm is used in several modern 'general-purpose' network intrusion detection systems, such as the latest version of Snort [14]. To our knowledge, ours is the first implementation of the algorithm on an NPU. Recent work at Georgia Tech uses IXP1200s for TCP stream reconstruction in an IDS for an individual host [4]. In this approach, a completely separate FPGA board was used to perform the pattern matching. IXPs have also been applied to intrusion detection in [16]. Detection in this case is limited to packet headers and uses a simpler matching algorithm.

The ability to 'sanitise' protocols before scanning the data for intrusion attempts is similar to the protocol scrubber [19] and *norm* [20], except that it was implemented in a much more resource-constrained environment. As a result, the mechanisms in *CardGuard* are considerably simpler (but possibly faster).

7 Conclusions

This paper demonstrates that signature detection can be performed in software on a NIC equiped with a network processor, *before* the packets hit the host's PCI and memory bus. While the hardware that was used in *CardGuard* is rather old, the principles remain valid for newer hardware. As modern NPUs offer higher clock rates and support more (and more poweful) microengines we are confident that much higher rates are possible. Perhaps that makes *CardGuard* amenable to implementation on edge routers also. We conclude that *CardGuard* represents a first step towards providing intrusion detection on a NIC in software and the evaluation of an unexplored corner of the design space.

Acknowledgments

References

[1] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: The spread of the Sapphire/Slammer worm, technical report. Technical report, CAIDA (2003) http://www.caida.org/outreach/papers/2003/sapphire/.

[2] Bellovin, S.M.: Distributed firewalls. Usenix ;login:, Special issue on Security (1999) 37–39

[3] Ioannidis, S., Keromytis, A.D., Bellovin, S.M., Smith, J.M.: Implementing a distributed firewall. In: CCS '00: Proceedings of the 7th ACM conference on Computer and communications security, ACM Press (2000) 190–199

[4] Clark, C., Lee, W., Schimmel, D., Contis, D., Koné, M., Thomas, A.: A hardware platform for network intrusion detection and prevention. In: Third Workshop on Network Processors and Applications, Madrid, Spain (2004)

[5] Toelle, J., Niggemann, O.: Supporting intrusion detection by graph clustering and graph drawing. In: Proc. RAID'00, Toulouse, France (2000)

[6] Barford, P., Kline, J., Plonka, D., Ron, A.: A signal analysis of network traffic anomalies. In: SIGCOMM Internet Measurement Workshop, Miami, FLA (2003)

 [7] Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y.: Sketch-based change detection: Methods, evaluation, and applications. In: SIGCOMM Internet Measurement Workshop, Miami, FLA (2003)

 [8] Yegneswaran, V., Barford, P., Ullrich, J.: Internet intrusions: Global characteristics and prevalence. In: Proc. of ACM SIGMETRICS. (2003)

 [9] Estan, C., Savage, S., Varghese, G.: Automatically inferring patterns of resource consumption in network traffic. In: Proc. of SIGCOMM'03. (2003)

[10] Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM **18** (1975) 333–340

[11] Paxson, V.: Bro: A system for detecting network intruders in real-time. Computer Networks **31(23-24)** (1999) 2435–2463

[12] Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast Portscan Detection Using Sequential Hypothesis Testing. In: IEEE SP'04, Oakland, CA (2004)

[13] Kompella, R.R., Singh, S., Varghese, G.: On scalable attack detection in the network. In: SIGCOMM Internet measurement conference. (2004) 187–200

[14] Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of the 1999 USENIX LISA Systems Adminstration Conference. (1999)

[15] N.Shalaby, L.Peterson, A.Bavier, Y.Gottlieb, S.Karlin, A.Nakao, X.Qie, T.Spalink, M.Wawrzoniak: Extensible routers for active networks. In: DANCE'02. (2002)

[16] I.Charitakis, D.Pnevmatikatos, E.Markatos, K.Anagnostakis: S2I: a tool for automatic rule match compilation for the IXP network processor. In: SCOPES 2003, Vienna, Austria (2003) 226–239

[17] Mogul, J.: TCP offload is a bad idea whose time has come. In: Proc. of HotOS IX, Lihue. Hawaii, USA (2003)

[18] Tuck, N., Sherwood, T., Calder, B., Varghese, G..: Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of IEEE Infocom, Hong Kong, China (2004)

[19] Malan, R., Watson, D., Jahanian, F., Howell, P.: Transport and application protocol scrubbing. In: Infocom'2000, Tel-Aviv, Israel (2000)

[20] Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: USENIX-Sec'2001, Washington, D.C., USA (2001)

[21] Johnson, E.J., Kunze, A.R.: IXP1200 Programming. Intel Press (2002)

[22] Debar, H., Dacier, M., Wepsi, A.: A revised taxonomy for intrusion-detection systems. Technical report, IBM Research, Zurich (1999)

[23] Smaha, S.E.: Haystack: An intrusion detection system. In: IEEE Fourth Aerospace Computer Security Applications Conference, Orlando, FL, USA (1988)

[24] Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Rowe, J., Staniford, S., Yip, R., Zerkle, D.: The design of GrIDS: A graph-based intrusion detection system. Technical Report CSE-99-2, UC Davis (1999)

[25] Cisco: Cisco secure intrusion detection system version 2.2.0 (netranger) (2002)

[26] Farmer, D., Venema, W.: Improving the security of your site by breaking into it. Technical report, Internet White Paper (1993) `http://www.fish.com/security/`.

[27] Moore, D., Shannon, C., Voelker, G., Savage, S.: Internet quarantine: Requirements for containing self-propagating code. In: Infocom, San Francisco, CA (2003)

[28] Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc. (1998)

[29] Lee, W., Cabrera, J.B.D., Thomas, A., Balwalli, N., Saluja, S., Zhang, Y.: Performance adaptation in real-time intrusion detection systems. In: RAID'02, Zurich, Switzerland (2002)

[30] Kerschbaum, F., Spafford, E.H., Zamboni, D.: Using embedded sensors for detecting network attack. Technical report, Purdue University (2000)