

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/228467093>

# A network intrusion detection system on IXP1200 network processors with support for large rule sets

Article · July 2004

---

CITATIONS

18

---

READS

220

2 authors, including:



[Herbert Bos](#)

Vrije Universiteit Amsterdam

224 PUBLICATIONS 9,075 CITATIONS

[SEE PROFILE](#)

# A network intrusion detection system on IXP1200 network processors with support for large rule sets

(Technical Report 2004-02)

Herbert Bos  
Leiden Universiteit / Vrije Universiteit  
2333 CA Leiden / 1081 HV Amsterdam  
The Netherlands  
H.Bos@cs.vu.nl

Kaiming Huang  
Universiteit Leiden  
2333 CA Leiden  
The Netherlands  
khuang@liacs.nl

## Abstract

In this paper we describe an network intrusion detection system implemented on the IXP1200 network processor. It is aimed at detecting worms at high speeds by matching the payload of network packets against worm signatures at the lowest possible levels of the processing hierarchy (the microengines of an IXP1200 network processor). The solution employs the Aho-Corasick algorithm in a parallel fashion, where each microengine processes a subset of the network traffic. To allow for large patterns as well as a large number of rules, the signatures are stored in off-chip memory. Using an old version of the IXP network processors (the IXP1200), the system is capable of handling close to 200 Mbps with full content scan for realistic threats.

## 1 Introduction

Intrusion detection systems (IDS) are increasingly relied upon to protect network and computing resources from attempts to gain unauthorised access to resources, e.g., by means of worms, viruses or Trojan Horses. Scanning network traffic at line speed for the occurrence of the signatures of attacks is a challenging task even with today's networks. Moreover, as it is often stated that the growth of link speed exceeds that of computational speed (and especially that of buses and memory), the problem is likely to get worse rather than better in the future.

To protect valuable computing resources on fast connections, there are no known alternatives for scanning traffic for malicious data at line rate. Worms especially are difficult to stop manually as they are self-replicating and spread fast. For exam-

ple, on the morning of January 25, 2002 a worm known as Sapphire began to infect hosts on the Internet and became (in)famous for being the fastest worm in history. Within 10 minutes 90% percent of all vulnerable hosts were infected [12].

Network processors have been proposed to cope with increasing link speeds [13]. The idea is to push packet processing code to the lowest possible level in the processing hierarchy, e.g., before the traffic even hits the host's PCI bus. These network processing units (NPUs) are explicitly designed for handling network packets at line rate. They offer parallelism and tailored instruction sets to provide the required throughput. Some NPUs, for instance the Intel IXP used in this paper, contain on-chip a number of indepent processing units known as *microengines*. High link rates can be handled for instance by configuring the NPU in such a way that each of the  $n$  microengines handles only one  $n^{th}$  of the traffic.

While NPUs have been successfully employed in many network devices, such as routers, monitors, etc., there have been few attempts to use them to implement the more computationally intensive task of intrusion detection. Often such attempts have been limited to header processing (e.g. [10]), or toy problems (e.g. [5]). To our knowledge there are no solutions that provide both high speed and scalability in terms of (a) the number of patterns to detect, and (b) the size of the patterns to detect.

In this paper, we describe NPIDS-1 an intrusion detection system on the IXP1200 network processor which is capable of scanning network traffic at a rate of roughly 170 Mbps, irrespective of the size or number of the patterns. It employs the well-known Aho-Corasick algorithm for performing

high-performance pattern searches [2]. The same algorithm is used in the latest versions of the Snort intrusion detection tool [14]. In our work, the algorithm has been fully implemented on the micro-engines of the network processor. NPIDS-1 uses the same Aho-Corasick algorithm for large rule sets. For rule sets with thousands of rules, the patterns have to be stored separate from the code in one of the memories present on the network card. As a result, they will incur a memory access each time part of the rule set is referenced.

The remainder of this paper is organised as follows. Section 2 provides an overview of the NPIDS-1 architecture. In Section 3, experimental results are discussed. Related work is discussed throughout the text and summarised in Section 4. Conclusions are drawn in Section 5.

## 2 Architecture

In this section, our approach to network intrusion detection is discussed. In particular, the configuration of hardware and software in NPIDS-1 will be explained.

### 2.1 Intrusion detection and Aho-Corasick

While increasing network speed is one aspect of the challenge in intrusion detection, scalability is another, equally important one. As the number of worms, viruses and Trojans increases, an IDS 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 [10]) are not sufficient. Similarly, fast scans for a small number of patterns (as described in [7]) 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, NPIDS-1 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 constructed (interested readers are referred to [2]). However, for better understanding of some of the design decisions in NPIDS-1, it is useful to consider in more detail the code that actually performs the matching.

#### 2.1.1 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 'g', the new state will be 14 and the algorithm proceeds with the next byte. In case this byte is 'e', 'g', or 'G', we will move to state 15, 14, or 1, 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 (indicated by black squares in Figure 1). Whenever an output state has been reached, we know that one of the signatures has matched. For example, should the algorithm ever reach state 34, this means that the data in the traffic contains the string 'get /NULL.ida'.

The DFA in Figure 1 is able to match four different patterns at the same time. The patterns are chosen for illustration purposes, but actually represent a small part of the exploit used by the Code Red worm that hit hundreds of thousands of Web servers in the summer of 2001 [1]. The worm exploited a buffer overflow vulnerability in the indexing service (.IDA) of Microsoft's IIS Web servers, which allowed it to execute code, as 'SYSTEM', on remote hosts.

Suppose that initially the state is 0 and that the input stream consists of the following characters: XYZGGET /NULL.ida. The algorithm proceeds as follows. The first three characters do not lead to a transition out of state 0. The fourth character, 'G' leads to a transition to state 1. The next G keeps the algorithm in state 1. Next, the state transitions for the character sequence 'E', 'T', ' ', '/', 'N', 'U', 'L' and 'L', move the algorithm to states 2, 3, 4, 5, 6, 7, 8, and 9, respectively. State 9 happens to be an output state which matches the signature 'GET /NULL'. The algorithm continues with the next character, which is a '.'. From state 9, it therefore continues with state 10. On the next three characters, 'ida' the algorithm makes transitions to states 11, 12 and 13, respectively. State 13

<b>States 0:</b> 'g' 14 'G' 1	<b>State 4:</b> '/' 5 'g' 14 'G' 1	<b>State 8:</b> 'g' 14 'L' 9 'G' 1	<b>State 12:</b> 'g' 14 'a' 13 'G' 1	<b>State 17:</b> '/' 18 'g' 14 'G' 1	<b>State 21:</b> 'g' 14 'l' 22 'G' 1	<b>State 25:</b> 'g' 14 'a' 26 'G' 1	<b>State 30:</b> '.' 31 'g' 14 'G' 1
<b>State 1:</b> 'g' 14 'E' 2 'G' 1	<b>State 5:</b> 'g' 14 'N' 6 'G' 1	<b>State 9:</b> '.' 10 'g' 14 'G' 1	<b>State 14:</b> 'e' 15 'g' 14 'G' 1	<b>State 18:</b> 'g' 14 'n' 19 'G' 1 'N' 27	<b>State 22:</b> '.' 23 'g' 14 'G' 1 'U' 28	<b>State 27:</b> 'g' 14 'G' 1 'U' 28	<b>State 31:</b> 'g' 14 'G' 1 'i' 32
<b>State 2:</b> 'g' 14 'G' 1 'T' 3	<b>State 6:</b> 'g' 14 'U' 7 'G' 1	<b>State 10:</b> 'g' 14 'G' 1 'i' 11	<b>State 15:</b> 'g' 14 't' 16 'G' 1	<b>State 19:</b> 'g' 14 'u' 20 'G' 1	<b>State 23:</b> 'g' 14 'G' 1 'i' 24	<b>State 28:</b> 'g' 14 'G' 1 'L' 29	<b>State 32:</b> 'g' 14 'd' 33 'G' 1
<b>State 3:</b> '.' 4 'g' 14 'G' 1	<b>State 7:</b> 'g' 14 'L' 8 'G' 1	<b>State 11:</b> 'g' 14 'd' 12 'G' 1	<b>State 16:</b> '.' 17 'g' 14 'G' 1	<b>State 20:</b> 'g' 14 'l' 21 'G' 1	<b>State 24:</b> 'g' 14 'd' 25 'G' 1	<b>State 29:</b> 'g' 14 'G' 1 'L' 30	<b>State 33:</b> 'g' 14 'a' 34 'G' 1

- Depicted above is the deterministic finite automaton for the following signatures:  
{"get /NULL.ida", "GET /NULL.ida", "GET /NULL", "GET /null.ida"}
- Matches are found in the following states (indicated in the table by '■'):  
{9, "GET /NULL"}, {13, "GET /NULL.ida"}, {26, "get /null.ida"}, {34, "get /NULL.ida"}

Figure 1: Deterministic finite automaton for part of the Code Red worm

is an output state. We have now also matched 'GET /NULL.ida'. In summary, by making a single transition per byte, all present patterns contained in the packet are found.

## 2.1.2 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 indication of a match. 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 34 states for 4 small, largely overlapping patterns, a little less than 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 [15]. 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 at all. The only exception is that if the number of patterns increases, in general the relative number of output states may also increase. This means that the actions corresponding to matches are likely to be executed more frequently. Fourth, parallelism can be exploited mainly by letting different processors handle different packets.

There is no benefit in splitting up the set of patterns to search for and letting different processors search for different patterns in the same packet. It may be worthwhile, however, to distribute the DFA over a memory hierarchy, where different types of memory have different size and different speed. For instance, placing the top of the DFA (which is accessed more frequently) in registers, while storing the rest in SRAM, is likely to lead to significant speedups. However, exploitation of the memory hierarchy is beyond the topic of this paper.

## 2.2 Hardware

The NPIDS-1 hardware architecture is sketched in Figure 2. One or more NPU boards are connected to a host processor via a PCI bus. The NPU in NPIDS-1 is an Intel IXP1200 running at 232 MHz, mounted on a Radisys ENP2506 board together with 8 MB of SRAM and 256 MB of SDRAM. The IXP consists of a StrongARM host processor running embedded Linux and 6 independent RISC processors, known as microengines. Each microengine has a 1K instruction store, 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 have their own program counters and while two threads on the same microengine cannot really execute in parallel, it is possible to context switch between threads at zero overhead. On-chip the IXP also has a small

amount (4KB) of memory, known as Scratch. Comparing Scratch memory to SRAM and SRAM to SDRAM the slow-down is roughly 1.5 to 2 times in both cases [11]. A final part of the ENP-board is that the IXP is connected to two Gigabit Ethernet ports. The unit on the processor responsible for interacting with the network is known as FBI (according to Intel, an acronym of unknown origin). The network processor is connected to the network interfaces via a fast, proprietary bus (known as the IX bus).

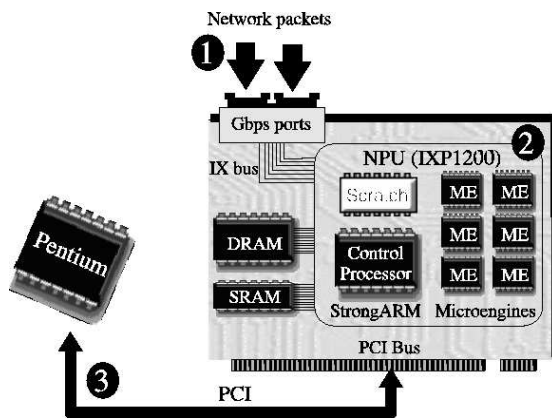


Figure 2: The NPIDS-1 hardware architecture

It should be mentioned that the IXP1200 is fairly old and no longer supported by Intel. For instance, a newer version of the IXP architecture, the IXP2800, supports no fewer than 16 microengines (with 8 threads each), has 16KB of scratch memory (in addition to a small amount of local memory at each of the microengines) and operates at 1.4 GHz. In a sense, this means that the results in this paper represent what can be achieved with yesterday's technology.

## 2.3 Software configuration

The goal of the NPIDS-1 project is to perform as much of the packet processing as possible on the lowest levels of the processing hierarchy, i.e., the microengines of the IXP. The motivation for this goal is that future network speeds are expected to outgrow the speed of PCI buses and memory, so that sending all packets to the host processor (assuming it is able to cope with them) is not feasible. If the majority of packets can be processed before they reach the PCI bus, the bottleneck can be overcome.

### 2.3.1 The big picture

In Figure 2, the numbering indicates the three major steps in the way the packet travels through the system. First, packets are received from the network ① and stored in a buffer. Next, they are processed by the microengines ②. During the processing, most packets are removed from the system, as they do not match the target signature. The small number of packets that do match the signature are treated as exceptions and sent across the PCI bus to the host for further processing (e.g., to trigger alarms) ③.

NPIDS-1 attempts to execute the entire runtime part of the IDS on the microengines. One microengine ( $ME_{rx}$ ) is dedicated to packet reception, while the remaining five microengines run the Aho-Corasick to perform intrusion detection. In the remainder of this paper, these engines will be referred to as  $ME_{ac}$ . A packet is processed as follows. First,  $ME_{rx}$  polls the status register of the FBI for the presence of new packets. Next, if a new packet is detected, the microengine transfers it to SDRAM in fixed-size chunks, known as mpackets, where it is stored in a circular buffer. The buffer consists of fixed-size slots, each large enough to contain any packet, and is partitioned in disjoint sets, so that each  $ME_{ac}$  processes a different set of packets. An  $ME_{ac}$  applies the Aho-Corasick algorithm to the packets and alerts the host when a pattern is matched. Finally, when processing completes, buffers are marked as available for re-use.

### 2.3.2 Resource mapping

Taking into account the hardware limitations described in Section 2.2 and the observations about the Aho-Corasick algorithm in Section 2.1.2, this section describes in more detail how data and code are mapped on the various memories and processing units, respectively. As described above and in line with the fourth observation in Section 2.1.2, NPIDS-1 distinguishes two types of microengine: (1)  $ME_{rx}$ , used for packet reception, and (2)  $ME_{ac}$ , used for the pattern searches. Both will be discussed presently (see also Figure 3).

**Packet reception** We take the usual approach of receiving packets in SDRAM, and keeping control data 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 NPIDS-1 is less common. The moment a 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 and hence intrusion attempts may remain undetected. 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 NPIDS-1 the slowest thread should always have progressed beyond a certain threshold index in the buffer (Figure 3). NPIDS-1 conservatively considers all cases in which threads are slow as system failures, which in this case means that NPIDS-1 is not capable of handling the rate at which the traffic is sent.

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 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,  $B = 10^5$  slots, and  $A = 1$  ms, a safe value for  $T$  would be 99490.

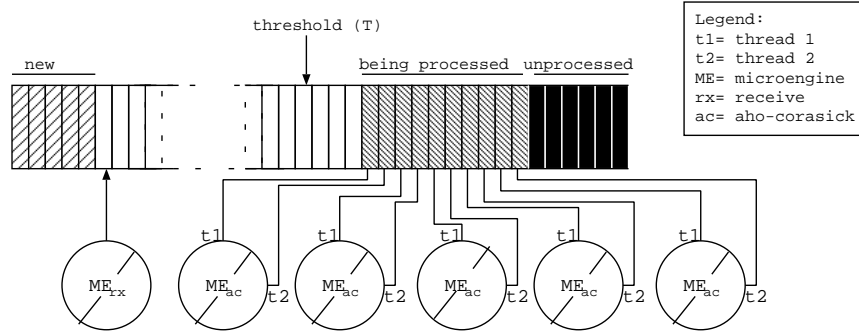
Observe that the threshold mechanism described above is overly conservative. Just because a thread has not reached the appropriate threshold at the time that  $ME_{rx}$  ‘wraps’, does not mean that it cannot catch up (for example, if the remaining packets are all minimum-sized, or new packets are big, and do not arrive at maximum rate). Moreover, it is possible to use the threads more efficiently, e.g., by not partitioning the traffic, but letting each thread process the ‘next available’ packet. We have chosen not to implement these optimisations, both for simplicity and because they require per-packet administration for such things as 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, NPIDS-1 needs a single check on 10 counters at wrap time.

**Packet processing** Each of the remaining microengines  $ME_{ac}(0)$ - $ME_{ac}(4)$  is responsible for processing one fifth of the packets. For this purpose, a thread on each microengine 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 [11]. Therefore, in order to hide latency, NPIDS-1 employs two threads. Whenever a thread stalls on a memory access, a zero-cycle context switch is made to allow the second processing thread to resume. In practice, employing more than two threads for packet processing does not improve performance and even proves detrimental, e.g., due to bus contention. As there are now ten packet processing threads in NPIDS-1, the buffer is partitioned such that thread  $t$  is responsible for packets  $t, t + 10, t + 20, \dots$ .

In order to allow for large rule sets whilst preserving reasonable access times, the DFA is stored in SRAM. The structure that is commonly used to store DFAs in Aho-Corasick is a ‘Trie’ with pointers from a source state to destination states to represent the transitions. By storing the DFA in SRAM, 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. The advantage is that thousands of rules can be stored in the 8 MB of SRAM that is available on the ENP board. Moreover, it scales to larger memories as well and can even be ported with minimal code change to SDRAM.

As there are several possibilities for storing the DFA (e.g., Scratch, SRAM and SDRAM) one may wonder why we have chosen SRAM: why not opt for memory that is significantly faster (Scratch), or memory that is significantly larger (SDRAM). The reason is that SRAM is a reasonable compromise between speed and size. While the 4KB Scratch memory is the most efficient, it is hardly larger than the microengine’s instruction store and register space. SDRAM on the other hand, while not a bad choice in and of itself, is roughly twice as slow as SRAM. For this reason, the DFA is stored fully in 8 MB SRAM with access times between 16 and 20

Figure 3: Packets are received by  $ME_{rx}$  and processed by  $ME_{ac}$ 

cycles.

### 2.3.3 Spilling

Although not very common, it is possible that a worm consists of more than one packet. In that case, a signature may straddle packet boundaries. In the current version of NPIDS-1, it is assumed that a pattern spans at most two packets, which for Ethernet implies a maximum signature length of 3000 bytes for maximum sized packets. However, it is easy to adapt the system to make it cope with signatures that stretch across more than two packets.

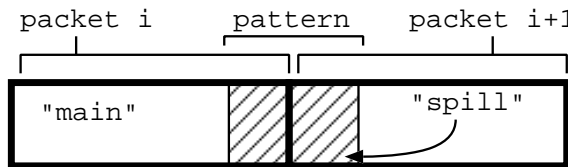


Figure 4: 'Spilling'

Given that a pattern may cross a packet boundary, it is not sufficient that a single thread on an  $ME_{ac}$  processes only one packet simultaneously. Instead, it should process the current packet and at least the first few characters in the next, as a pattern may start at the end of the current packet and finish in the beginning of the next. We refer to this process as *spilling*. Spilling is illustrated in Figure 4. The current packet that is processed by a thread is called the *main packet*, while the next packet is referred to as the *spill packet*. The number of characters to process in a spill packet is determined by the size of the patterns. An upperbound is the length of the largest pattern minus one.

As a result, all packets except the first have to be processed by two threads: the 'main' thread and the

'spill' thread, before the buffer slot can be reused. It is not necessary to add extra checks to verify this, however, as the threshold mechanism implies that when the write position 'wraps' the progress of all ten threads is checked. The index of the main packet of the slowest thread should be greater than the threshold. If this is the case, the index of the spill packet is certainly greater than the threshold. hence, the slot will not be reused until it is safe to do so.

### 2.3.4 Reporting matches

When a signature was 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 and signals code running on the StrongARM. The StrongARM code has several options: it may take all necessary actions itself, or it may defer the processing to the host processor. In NPIDS-1, the latter solution is the default one. The packet is sent to the host as an Ethernet packet, and a copy of the report is also made available to code running on the host processor. When done, the  $ME_{ac}$  marks the buffer slot as available for reuse. On the host, both the packet and the report are sent to a userspace application for further processing.

## 2.4 Implementation details

Microengines on IXPs can be programmed either in assembly, or in a specific dialect of C, known as 'microengineC'. We have used the latter for the code running on the microengines.

The construction of the Aho-Corasick DFA is done offline on the host connected to the IXP board. The new DFA can be loaded in SRAM by the control code, which subsequently signals all threads to

764 byte pkt	1464 byte pkt
192.6 Mbps	194.3 Mbps

Table 1: Sustainable rates for different packet sizes

packet size (bytes)	Cycles
64	1821
764	56905
1464	112011

Table 2: Clock cycles per packet

restart in state 0, with the new DFA. For this purpose, we allocated two areas in SRAM where DFAs may be stored. Whenever a thread receives the ‘change DFA’ signal it will automatically read from the area 1, if the previous area was area 2, and from area 2 otherwise. In this way, the downtime can be kept to a minimum.

### 3 Results

One of the problem of evaluating the NPIDS-1 implementation is generating realistic traffic at a sufficient rate. In the following, all experiments involve the deterministic finite automaton shown in Figure 1. In other words, they are aimed at detecting part of the Code Red worm. 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 could be generated with this tool was very limited, in the order of 50Mbps. At this rate, both implementations of NPIDS-1 could easily handle the traffic.

Preliminary results are shown in Table 1. Here the UDP packet generator `rude` was used to send traffic at maximum rates. Note that only the results for fairly big packets are included in the figure. The reason is that for small packets, `rude` was not able to send at a sufficiently high rate to exceed the capability of NPIDS-1. As a result, we do not currently know the maximum sustainable rate for small packet. In fact, `rude` generated less than 50Mbps for small packets, which could easily be handled.

As a second experiment, we examined the number of cycles that were needed to process packet of various packet sizes. The results are shown in Table 2.

As the signature is rather small, NPIDS-1 was tested with larger signatures as well. However, as

argued before the size and number of the patterns does not significantly contribute to the overhead. As a result, NPIDS-1 was able to sustain a rate close to 190 Mbps, depending on the sizes of the packets.

## 4 Related work

Aho-Corasick is used in several modern ‘general-purpose’ network intrusion detection systems, such as the latest version of Snort [14]. However, to our knowledge, this is the first implementation of the algorithm on an NPU. We are aware of one commercial implementation of the Boyer-Moore algorithm on a board that carries an IXP2400 with a special-purpose daughtercard [6, 7]. IXPs have also been applied to intrusion detection in [10]. However, the intrusion detection in this case is limited to packet headers only. Boyer-Moore only matches a single string at a time and is therefore less interesting for intrusion detection systems that search for many attack signatures. It was improved by Horspool which also provided as simpler and more efficient implementation [9].

Other research projects have looked at speeding up pattern searches by improving the search algorithms even more. An example is found in [8] which proposes a set-wise Boyer-Moore-Horspool algorithm to make Boyer-Moore match a set of patterns at the same time [8]. It is a hybrid approach that employs a different algorithm, depending on the number of patterns that should be matched. While it employs Aho-Corasick in case the number of patterns to match is greater than 100, it is faster than Aho-Corasick when the number of patterns is small. Recent work in intrusion detection algorithms, known as E2xB, achieves a speedup over existing algorithms between 10 and 35 percent [3]. While important work, we did not use any of these improvements in NPIDS-1, as it would be very hard indeed to fit them in the small instruction store of the IXP1200. The reason why we did not consider a more memory-efficient implementation of Aho-Corasick (such as [15]) in NPIDS-1 is that it less efficient in performance.

The work described in this paper builds on our own previous work in which Aho-Corasick on network processors was employed to scan DNA databases for the occurrence of DNA sequences that were similar (but not necessarily equal) to a given query [4].



## 5 Conclusions

This paper shows how intrusion detection can be performed on a network processor, *before* the packets hit the host's memory and PCI bus. As network speed is expected to outgrow bus and memory speeds, such an implementation is a requirement for dealing with the line rates of the future. While the hardware that that was used in NPIDS-1 is rather old, the principles remain valid for newer hardware also. As modern NPUs offer a clock rate that is much higher than the clock rate of the IXP1200 that was used for the results in this paper, and also support more microengines (each with more threads, larger instruction stores, better synchronisation primitives, etc.), we are confident that a much higher rate is achievable. However, one should be cautious when predicting *how* much faster NPIDS-1 would run on an equivalent NPU with a clock rate that is an order of magnitude higher, as faster code may also lead to bottlenecks in accessing on-board memories. In particular, it would be premature to say that such code would also run an order of magnitude faster and the only way to find out the new performance is by measuring it. We therefore cautiously conclude that NPIDS-1 represents a first step towards providing intrusion detection at multi-gigabit speeds.

## Acknowledgements

This work was supported by the EU IST SCAMPI Project. Our gratitude goes to Intel for providing us with IXP12EB boards and to the University of Pennsylvania for letting us use one of its ENP2506 boards.

## References

- [1] Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, 2002.
- [2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.
- [3] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis. E2xb: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of the 18th IFIP International Information Security Conference (SEC2003)*, May 2003.
- [4] Herbert Bos and Kaiming Huang. On the feasibility of using network processors for DNA queries. In *Proceedings of the Third Workshop on Network Processors & Applications - NP3. Submitted.*, Madrid, Spain, February 2004.
- [5] Herbert Bos, Bart Samwel, Mihai Cristea, and Kostas Anagnostakis. Safe execution of untrusted applications on embedded network processors. In *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, Marcel Dekker, Inc.
- [6] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
- [7] White Paper DeCanio Engineering, Consys-tant. The snort network intrusion detection system on the intel ixp2400 network processor. <http://www.consys-tant.com/technology/>, February 2003.
- [8] M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection, technical report cs2001-0670 (updated version). Technical report, University of California, San Diego, 2002.
- [9] R.N. Horspool. Practical fast searching in strings. *Software - Practice and experience*, 10:501–506, 1980.
- [10] I.Charitakis, D.Pnevmatikatos, E.Markatos, and K.Anagnostakis. S2I: a tool for automatic rule match compilation for the IXP network processor. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2003*, pages 226–239, Vienna, Austria, September 2003.
- [11] Erik J. Johnson and Aaron R. Kunze. *IXP1200 Programming*. Intel Press, 2002.
- [12] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the Sapphire/Slammer worm, technical report. Technical report, CAIDA, 2003. <http://www.caida.org/outreach/papers/2003/sapphire/>.
- [13] Kurt Keutzer Niraj Shah. Network processors: Origin of species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, October 2002.
- [14] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, 1999. Available from <http://www.snort.org/>.
- [15] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the IEEE Infocom Conference* [2], pages 333–340.