Code Generation for Packet Header Intrusion Analysis on the IXP1200 Network Processor

I. Charitakis[1], D. Pnevmatikatos[1], E. Markatos[1], and K. Anagnostakis[2]

[1] Institute of Computer Science (ICS)
Foundation of Research and Technology - Hellas (FORTH)
P.O.Box 1385, Heraklion, Crete, GR-711-10, Greece
{haritak, pnevmati, markatos}@ics.forth.gr

[2] Distributed Systems Laboratory
CIS Department, Univ. of Pennsylvania
200 S. 33rd Street, Phila, PA 19104, USA
anagnost@dsl.cis.upenn.edu

Appears in 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003), Vienna, Austria, September 2003

Abstract. We present a software architecture that enables the use of the IXP1200 network processor in packet header analysis for network intrusion detection. The proposed work consists of a simple and efficient run-time infrastructure for managing network processor resources, along with the S2I compiler, a tool that generates efficient C code from high-level, human readable, intrusion signatures. This approach facilitates the employment of the IXP1200 in network intrusion detection systems while our experimental results demonstrate that provides performance comparable to hand-crafted code.

1 Introduction

Network processor vendors have invested considerable effort in tools for cost-effective software development, however, building an application for a network processor is still a non-trivial task. To address this difficulty, recent work has demonstrated the use of component models for simplifying development (c.f. [1, 8, 2]). This work focuses on forwarding and routing services, exploiting application modularity in a divide and conquer approach in order to map parts of the application to network processor execution resources. The main design goal is primarily flexibility and design modularity which usually comes at the price of some performance penalty.

Network monitoring and network intrusion detection are becoming increasingly important network-embedded functions[6, 5]. Network Intrusion Detection Systems improve security for organizations by monitoring in real time the traffic that crosses the border of their networks. They passively inspect traffic to determine if it matches an attack profile. The simplest and most common form of NIDS inspection is to analyze packet headers and match string patterns against

```
alert    tcp 10.0.0.0/32 any -> 10.0.0.1 80         (dsize: >512;)
alert    tcp [10.0.0.1 10.0.0.2] any -> 10.0.0.2 80 (ack: >512;)
alert    tcp any any -> 10.0.0.300 20               (dsize: >512;)
```

Fig. 1. Example of `snort` signature file. Note the use of *any*, which serves as a wild-card

the payload of packets. A popular open-source NIDS is `snort` [5], that uses signatures to describe a set of known forms of attacks. `Snort` signatures consist of three parts: the *action* (e.g. alert), the *header* (protocol + source[IP,port] + dest[IP, port]), and the *options* (ip-flags, ip-options, tcp-options, etc). Figure 1 shows a few examples of actual `snort` signatures.

The flexibility required by the dynamic nature of intrusion detection applications, along with their inherently large processing needs makes network processors an ideal implementation technology. However, these applications differ substantially from services such as packet forwarding that have been studied so far. In this paper we present the `snort` To IXP compiler (S2I), a tool to facilitate the deployment of the IXP1200 network processor in a `snort`-based NIDS. The input of the S2I compiler is a regular `snort` configuration file, which contains signatures for a collection of known intrusions (some typical signatures are shown in Figure 1). Each signature is defined in a high-level language and describes the action to be performed (e.g. alert, log, etc) when a packet satisfies a set of conditions.

S2I transforms such a set of `snort` signatures into efficient C code for the micro-engines of IXP1200. The transformation is performed using a tree-structure in order to minimize the number of required checks. The resulting code together with a general runtime environment can be compiled, optimized, and loaded on the IXP1200 using the standard tool chain. There are three main benefits from this approach. First, it offers faster execution speeds of the resulting code, which is comparable to hand-crafted code. Second it provides versatility since adding or changing the signatures involves only running tools and not hand-tuning. Finally, it offers transparent resource management.

Using cycle-accurate simulations we measure significant reduction in both the required space and execution time. Space improvements range from about 14% to 42%, with the improvement magnitude increasing with the number of signatures. In addition, execution time improves by about 20%.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the target architecture, and Section 3 describes the S2I compiler. Section 4 contains an experimental analysis, Section 5 presents related work, and Section 6 concludes and discusses our plans for future work.

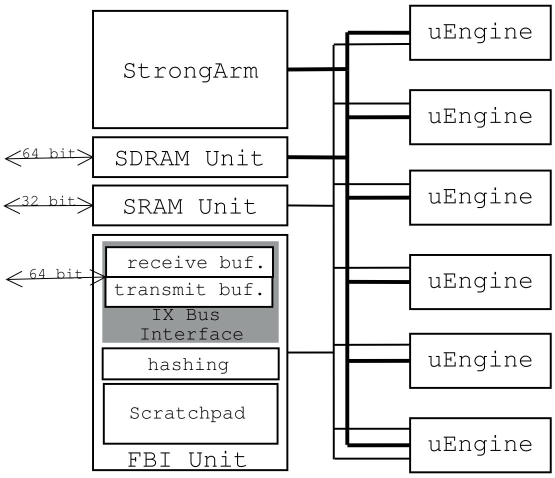

Fig. 2. Block architecture of the IXP1200 Network Processor

2 The Intel IXP1200 Network Processor

2.1 General Description

A block architecture for the IXP1200 network processor is shown in Figure 2. The IXP1200 consists of the following basic components:

- A StrongARM host processor, capable of operating at 232 MHz.
- Six micro-engines operating at the same frequency as the host processor.
- An SDRAM unit communicating to external SDRAM via a 64 bit bus at 116 MHz.
- An SRAM unit communicating to external SRAM via a 32 bit bus at 116 MHz.
- The FBI unit which provides 1 KB memory of 32 bit words (the scratchpad), a hash unit and the IX bus unit. The later interfaces to network interface cards (NICs) and provides the Receive Buffer and the Transmit Buffer.

2.2 The Micro-engines

Each micro-engine (or uEngine) is a simplified RISC processor that has hardware support for four threads of execution. Special instructions allow a thread of execution to be swapped out until a certain event occurs. Such events are "data-written", "data-read", "signaled", etc.

Each thread receives one fourth of the register space of the uEngine using relative register referencing, while absolute register referencing can be used for thread communication via shared registers. Each uEngine contains 128 32-bit general purpose registers, and 128 32-bit special purpose registers dedicated for data transfers (e.g. SRAM/SDRAM read-only and write-only registers, etc).

The uEngines can fetch data directly from SRAM, SDRAM, and the FBI (scratchpad, receive and transmit buffers, etc), but cannot exchange data directly between them. Instead, uEngines can exchange messages through shared memory accesses which are expensive: for example, reading 4 bytes from the SRAM takes 22 cycles. [3]

The uEngines are responsible for managing the buffers of the network interface ports e.g. monitoring the state of the buffers and initiating data transfers when necessary.

Each uEngine has 2 KB of instruction memory; programs are uploaded by the host processor by writing in a specific memory address region.

3 The S2I Compiler

The `snort`-to-IXP1200 (S2I) compiler generates micro-C [4] code for the uEngines from a `snort` set of signatures. The generated code consists of a static and a dynamic section. The static section is a skeleton that is independent of the set of signatures and contains the fixed run-time infrastructure needed to dispatch packets for processing. The dynamic section is produced from the `snort` set of signatures and performs the actual computation to analyze packet headers and trigger the corresponding actions.

The space and performance benefits of S2I are based on the following observation. An interpretive approach where the signatures are kept in data structures in memory is expensive both in time (e.g., executed instructions and memory references) and space since for each signature, the interpreter input can be a large structure defining which fields to check, what operation to perform and against what value. A compiled approach is faster since it avoids the interpretation cost, and allows for standard compiler optimizations. The compiled approach may also result in more compact code since many of the constants can be embedded in the instructions themselves, saving space.

An essential optimization pass performed by S2I is common-subexpression elimination using an expression evaluation tree. When several signatures share the same *prefix* conditions, these conditions are evaluated only once. Organizing the signature checks in a tree saves both space (each datum is stored once) and time (each condition is evaluated once). While this possibility is available to the programmer as well, implementing the code for a large number of signatures is error prone, reduces code readability, and is very hard to adapt to a new set of signatures. S2I provides performance close to that of hand-crafted code while offering the advantage of a standard and managable high-level input specification.

[3] Newer models (the IXP2xxx series) support more efficient inter-uEngine communication, by chaining uEngines and by providing shared registers between neighboring uEngines.

[4] micro-C is an enriched version of the C language provided by Intel. It provides primitives to support the architectural advantages of the uEngines.

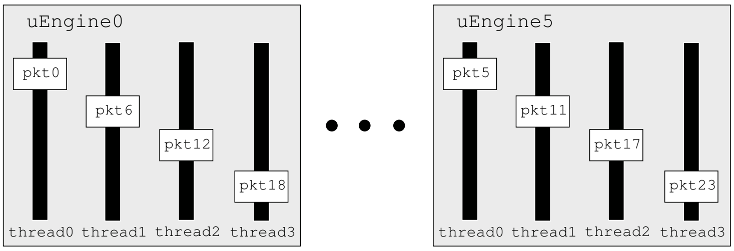

Fig. 3. Thread based scheme: packets distributed among the threads. All threads execute the same code

3.1 Static Section

We describe the structure of the static section of the generated code. It consists of the necessary minimal infrastructure for basic packet handling as well as an algorithm for distributing the packet processing load to different units of the network processor. Currently we target 100 Mbit/s ports.

For the purpose of our particular design, each packet is processed in-full by a single uEngine. Finer-grained load distribution would require inter-uEngine transfers of data as some processing to occur in one worker and the rest to be handed off to another, due to different packets requiring different amounts of processing. Since inter-uEngine transfers are not efficiently supported by hardware, such a scheme was not considered at all for this work.

The basic approach for load distribution is therefore to assign each packet to a single worker for its entire processing. This results in a reasonably balanced system, as a busy worker cannot issue requests for more work and therefore new packets will be assigned to the least busy uEngine. This approach also minimizes accesses to shared resources as the work for each packet is isolated on a single uEngine.

Because of the multi-threaded structure of the IXP1200, a worker for a packet can be either an individual thread or an entire uEngine. We therefore consider two different methods for load distribution presented below.

The **thread-based scheme**, (shown in Figure 3) assigns the entire processing of each packet to one of the four threads of a uEngine. This has the advantage of simplicity, and yields the same code for all threads. The 2 KB of instruction memory are unified and shared by all threads.

One drawback of the thread-based scheme is that the registers of each uEngine must be equally divided among the four threads. Each thread has to fetch the headers of its packet in its local registers, for processing, consuming 14 registers: 54 bytes are needed for Ethernet, IP and TCP headers. A total of 56 registers are therefore needed for all threads, corresponding to 30% of the total number of registers that can be read. Another disadvantage is that processing of the four packets inside the uEngine is done in an interleaved manner, meaning that only one thread is active and only one fourth of these registers are actually used at any given time.

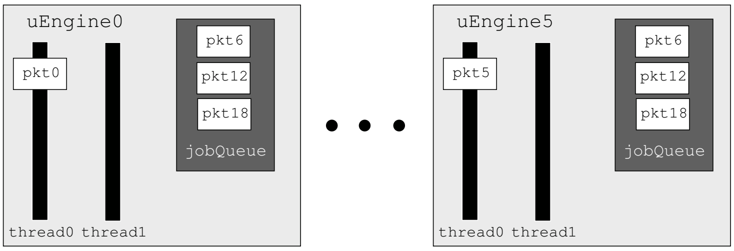

Fig. 4. uEngine based scheme: packets distributed among the uEngines. One thread does the actual header checking, while the other one maintains the jobQueue

Synchronization among the threads is accomplished in two steps. First, we allow one thread from each uEngine to start issuing willingness of receiving a packet. On the second step, up to six threads (one for each uEngine) race to lock the input port. The winning thread receives a packet, releases the acquired lock, and commences signature checking. Meanwhile, the other threads can perform the same synchronization method in order to serve the next packet.

An alternative to the thread-based scheme is the **micro-engine-based scheme**, where an entire uEngine is allocated for serving a packet. This is illustrated in Figure 4. In this case, the threads are responsible for specific jobs of packet processing, such as moving the packet between microengine and SDRAM and performing the actual header processing.

In contrast with the thread-based scheme, the uEngine based scheme results in one packet being active per uEngine, consuming only 14 registers for packet headers. Leaving more local space available enhances the chances for the dynamic section to fit entirely within a uEngine. In other words, we want all the variables that are necessary to perform the signature checking to be mapped to registers. If there are not enough registers, some variables will have to be mapped to scratchpad or SRAM which greatly degrades performance.

Note, that in the thread-based scheme, there are four packets concurrently in each uEngine. In order to give the same processing time for each packet to the uEngine based scheme, small buffers should be kept. These buffers (jobQueues) ideally will hold pointers to three more packets, that this uEngine should process in the future. Now, the processing of these packets is done serially, rather than interleaved.

This scheme was supported further with a simpler synchronization method. Each uEngine signals the next one to start polling for arrived packets. Therefore accesses to memory (for locking/unlocking) are avoided, and the system behaves much more smoothly.

Both schemes demostrate similar performance. However, since the uEngine based scheme requires much fewer resources, it was chosen to work with.

3.2 Dynamic Section

In this section we present the dynamic section, i.e. the generated micro-C code. This code is dynamic in the sense that it can be automatically reproduced every time a new set of signatures is used. In this context, dynamic does not imply that the executed code is changed during run time. Certainly, it would be desirable to operate continuously and never having to interrupt packet monitoring. However, this flexibility would sacrifice performance if it was built-in the architecture of the monitoring system. On the other hand, there may be other ways to achieve both constant operation and flexibility without sacrificing performance (e.g. using redundancy). This can be subject of future work.

The dynamic section is the output of the S2I compiler for the particular `snort` input file. S2I does not yet support all `snort` features. More specificaly, this version of S2I does not support payload searches. In an overview the functionality of the S2I compiler is divided in to two basic tasks. Firstly it builds a tree-like representation of the signatures in the input file and secondly it produces the corresponding micro-C code.

Building the Tree. Having a complete array of signatures, i.e. the complete input file in an internal representation, the S2I compiler starts combining the signatures in a tree structure. In this tree, each level corresponds to checking a specific field. For example at the first level we check for the protocol field, while at the second level we check for the destination port. This is depicted in Figure 5 where we show the resulting tree of the signatures presented earlier in Figure 1.

The S2I compiler initializes the tree using the first signature. Then, for each next signature, it starts combining each of the fields into the tree, following a predefined order. [5] New nodes are generated if the added field of a signature checks against a value that has not been seen earlier. The algorithm that builds the tree sorts each level from the most specific values to the most general. Therefore, stricter signatures will be checked before more general.

Producing the Final Code. During this phase, the S2I compiler generates the code that will be merged and compiled together with the static infrastructure presented earlier. The final object file will be loaded in the uEngines, and monitoring can be initiated. In Figure 6 we present a snapshot of some sample code produced by the S2I compiler.

It is important to note the use of constant literals in the various checks. For example, the signature that checks for the destination port 80, will be compiled to code similar to: "if (PORT2 == 0x50)" rather than code like "if (PORT2 == ports[i])". In this way we reduce memory accesses which are expensive and may significantly degrade performance. This optimization was discussed in [3] and was found to also work well for our particular design.

[5] In the future, we plan to guide this process with the assistance of a profiler, in order to intelligently select which checks to perform first.

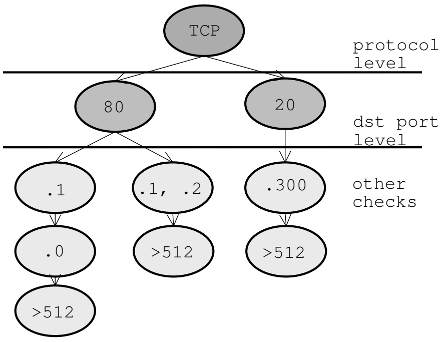

Fig. 5. Resulting tree from the signatures presented earlier in Figure 1

```
(.......................................................)
if (ETHPROTOCOL==0x0800 && PROTOCOL==0x6) {
  if (PORT2==0x50) {
     if (IP2==0xa000001) {
        if (IP1==0xa000000) {
           if (DSIZE>0x200) {
        /*Action for "tcp 10.0.0.0 any -> 10.0.0.1..." */
  }}}
  ctx_swap();
  if (IP2 == 0xa000002) {
     if (IP1==0xa000001 || IP1==0xa000002) {
        if (ACK>0x200) {
        /*Action for "tcp [10.0.0.1 10.0.0.2] any..."*/
  }}}
(.......................................................)
  }//<<<<PORT2
  if (PORT2==0x14) {
     if (IP2==0xa00012c) {
        if (DSIZE>0x200) {
        /*Action for "tcp any any -> 10.0.0.300 20..."*/
}}}}
 ctx_swap();
```

Fig. 6. Generated code for the tree of Figure 5

We should note that although this paper focuses on the IXP1200, the micro-C code produced by S2I can be slighlty modified so as to be compiled on a general purpose processor. Moreover, it can be easily adapted for other embedded or network processors. (An i386-based implementation of a lightweight `snort`-like system is briefly analyzed in Section 4.)

For the IXP1200, the S2I compiler will also insert context swap directives in certain points of the code. Context swaps are needed to voluntary let the current thread swap out of execution so that other threads on the same uEngine will have a chance to execute. This is done to avoid monopolizing a uEngine for too long. If all uEngines are claimed by running threads, then the buffer of the monitored port is likely to overflow causing packet loss.

4 Evaluation

In this section we perform the evaluation of the proposed software architecture. We evaluate separately the static section and the generated dynamic code.

4.1 Evaluation of the Static Section

The evaluation of the static section was done by measuring the *headroom* [7, 2] of the system: the number of cycles that can be consumed for processing each packet (plain signature checking) without causing packet loss, using minimum-sized packets.

We produced minimum sized packets arriving from one port at 100 Mbit/s. Using the uEngine-based infrastructure, each packet was received and brought locally to a uEngine. Processing on the packet was simulated by performing some initial field extractions and then running a loop checking some values against some fields. Each loop was guaranteed to perform a fixed number of calculations and to take a constant number of cycles. We measured the number of loops that can be supported without dropping packets. By varying the number of available uEngines we measured how the headroom scales. Moreover we multiplied the number of loops that can be supported by the cycles that takes each loop. The corresponding number of cycles is the available headroom. The results are shown in Figure 7 and indicate how many cycles are available in each uEngine to perform the actual signature checks. We can see that using all the uEngines of the IXP1200, we have approximately 4920 cycles available for processing of each 64 byte packet.

The results show that the headroom offered by the static section is comparable to previous estimations [7] in which the authors used 100 Mbit/s links as well.

4.2 Evaluation of the Dynamic Section

The dynamically generated code is heavily influenced by the tree structure of the field checking. In this section we evaluate the effects of using this tree structure both in space and in performance.

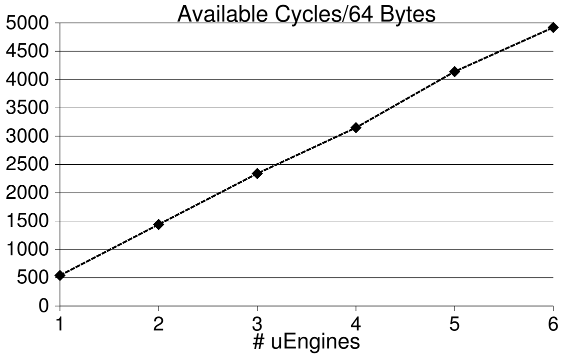

Fig. 7. Headroom in uEngine cycles (232 MHz). Simulation assumed one link at 100 Mbit/s and minimum sized packets

4.3 Evaluation of Space Requirements

Space requirements are crucial since the entire set of signatures must be loaded in the instruction memory of the uEngines in order to perform intrusion detection. Given that some space is already dedicated for the static section (approximately 476 words for the uEngine-based static scheme), the rest (1572 words) will have to be handled very efficiently. In this section we perform some simple experiments in order to measure how much space we gain by using a tree structure.

We used a modified version of the S2I compiler configured so as to produce code without using a tree structure. That is, each signature will result in one separate code-block which includes all its checks (e.g. as illustrated in Figure 8). The tree-like example of the code produced for this example was presented earlier in Figure 6.

Using several signature input files from the `snort` distribution site, we measured the total number of instruction words that the signature checking consists of. [6] Table 1 summarizes our findings.

S2I offers size reduction (compression) for all files, with magnitude varying from 17.24% to 69%. The S2I space benefits increase as the size of the input file increases, indicating its success to combine multiple signatures in a shallow tree. At the extreme case of icmp-info signatures, S2I manages to fit all the required code in instruction memory, while with the simple approach the signatures do not fit in the uEngine memory. These results are very encouraging, since in our tests, S2I is able to perform when needed most, i.e. for large input files.

4.4 Evaluation of Execution Time

In addition to space, S2I promises also gains is performance, since traversing the tree is a very efficient way of evaluating the signatures. In order to gain

[6] We subtracted from the total number of instructions the size of the static section (which was 476 instructions).

```
(......................................................)
if (ETHPROTOCOL==0x0800  && PROTOCOL==0x6) {
   if (PORT2==0x50) {
      if (IP2==0xa000001) {
         if (IP1==0xa000000) {
            if (DSIZE>0x200) {
         /* Action for "tcp 10.0.0.0 any -> 10.0...."*/
}}}}}
//alert tcp [10.0.0.1 10.0.0.2] any -> 10.0.0.2 80 (ack: >512;)
if (ETHPROTOCOL==0x0800 && PROTOCOL==0x6) {
   if (PORT2==0x50) {
      if (IP2==0xa000002) {
         if (IP1==0xa000001 || IP1==0xa000002) {
            if (ACK>0x200) {
         /* Action for "tcp [10.0.0.1 10.0.0.2] ..."*/
}}}}}
(......................................................)
```

Fig. 8. Linear code produced by the S2I compiler by disabling the tree structure optimizations. Each signature is implemented in an independent code block

Table 1. Space Savings using Tree structure

Signature File	Signatures	Plain Code inst/ions	Tree Code inst/ions	Reduction
icmp-info	79	did not fit	479	>69.00%
backdoor	44	1531	886	42.13%
web-misc	18	401	277	30.92%
virus	6	173	149	13.87%
web-cgi	4	145	120	17.24%

Table 2. Cycles(232 MHz) spent on field checking

Scenario	Plain Code	Tree Code	Reduction
signature0+signature4	75	60	20.00%
signature1+signature4	74	62	16.22%
signature2+signature4	74	59	20.27%
signature3+signature4	74	61	17.57%
signature4 only	47	29	38.30%
Average	68.8	54.2	21.22%

intuition on the speed improvements, we contacted the following experiments in the IXP1200 Simulator.

Artificial signatures and artificial traffic. We used five different signatures compiled using both the tree and without the tree. Then, we produced traffic with interleaved packets so as the signatures are matched sequentially: first packets matches first signature, second matches the second signature, etc. The fifth signature was a wild-card and therefore all packets matched it.

For this setting we measured the number of cycles spent on checking fields for the two compiled sources. In Table 2 we provide details of our findings. For each scenario, (packet matches signature 1, packet matches signature 2,...) we present the total number of cycles that were spent on performing checks. This time includes the time needed to perform an action when a match is found. (An action was simply to increment the value of an address in scratchpad).

As it can be seen the number of field checks is decreased by 21.2% on average. More interesting however, is that the performance gains are larger where there are fewer matches in the input (as in the "signature 4 only" case). The reason for this behavior is that if there is a match, the linear search of the simple implementation will stop quickly (for the few signatures we evaluated here). However, if the signatures do not match, the search will continue for longer. A tree structure allows the search to stop even in intermediate branches of the tree if the prefix does not match.

Using artificial signatures and real traffic. This experiment is conducted to increase our confidence in the previous evaluation and to indicate that the inputs we used are not skewed in favor of S2I. In this scenario, we conducted experiments using a small set of artificial signatures similar to the above. These signatures count packets based on protocol, source host, target host and payload size. However, unlike the previous case, we used real network traffic trace. This trace primarily consists of web traffic which was taken at ics.forth.gr, during a work day. Again we measure 20% on average reduction in the time spent on checking fields.

Using real signatures and real traffic. Finally, to get a feeling of the actual impact on real applications with real traces, we used the same trace, and the `snort` "backdoor" set of signatures. We ran this trace with the simple and the S2I tree structure, and measured total cycles spent on one packet. The results show that using the simple, sequential code, the field checking of the 44 signatures takes about 280 cycles. When compressing the field checks using the tree, the number drops to about 180 cycles, corresponding to a reduction of 35%.

Summarizing, the use of the tree is beneficial both for space and time reasons. Regarding space, we observe a minimum compression of 17.3% in instruction memory. Regarding time, we observe a significant reduction of around 20% in the time spent to apply the signatures, using some simple scenarios.

4.5 Lightweight `snort` for i386 systems

The output of the dynamic section of the S2I compiler can be used as a base to program any kind of processor. In this section we present experiments with the S2I output C code on an Intel Pentium processor. We compare the user time of executing the original `snort` and the lightweight version produced using the S2I tool.

We extracted from the default `snort` signature set all the signatures that do not require payload search. Then we used the S2I tool to produce a lightweight `snort` based on the remaining signatures. We run `snort` and lightweight `snort` over a trace taken from the NLANR archive [4]. While the user time of the original `snort` is about 12 seconds, our lightweight `snort` takes less than 5 seconds – an improvement of more than 50%.

5 Related Work

Research in tools and methodologies for network processors have focused mainly on routing-like applications and on modularity, re-usability and ease of programming.

In [8], Spalink et al. use the IXP1200 to build a software-based router. They propose a two-part architecture, which consists of a fixed infrastructure and a dynamically re-programmable part. The use of a network processor in software routers is also discussed in [2]. The authors present a tool supporting the dynamic binding of different components to form a fully-fledged router. The tool provides a basic infrastructure for controlling program flow and the data from one component to another, and a way for binding the components before uploading the code on the uEngines.

Dynamic code generation for packet filtering has been studied by Engler et al. in [3], with focus on efficient message demultiplexing in a general purpose OS. They present a tool that generates code based on a filter description language. Each filter is embodied at runtime in a filter-trie in a way that takes advantage of the known values the filter checks for.

6 Summary and Future Work

Hand coding hundreds of signatures in micro-C or assembly is a painful and error-prone task. In this paper we have proposed a software architecture and a tool for generating IXP1200 code from NIDS signatures. Using the S2I compiler, this task is being highly automated, translating a high-level signature specification into high-performance code. Therefore implementing intrusion analysis on the IXP1200 becomes a process that does not require knowledge of architecture internals and the micro-C programming language. Overall, the S2I compiler is able to produce fast and efficient code. while offering development speed and versatility.

There are several directions for future work that we are pursuing. First, we are working on tuning the S2I infrastructure. For instance, we consider improving the tree structure by adapting the field order for each sub-tree in order to minimize space, and execution profiles to reorder fields for minimizing processing time. Second, we are investigating the applicability of our design to higher-speed ports (e.g. 1 Gbit/s on the IXP1200). Finally, we are interested in applying the same general design principles of application-specific code generation to content matching, which is of great practical interest in intrusion detection.

Acknowledgments

This work is funded by the IST project SCAMPI (IST-2001-32404) of the European Union. It is also supported by Intel through equipment donation.

References

1. Intel IXA SDK ACE programming framework developer's guide, June 2001. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.
2. A. Campbell, S. Chou, M. Kounavis, V. Stachtos, and J. Vicente. Netbind: A binding tool for constructing data paths in network processor-based routers. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH' 02)*, June 2002.
3. D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *In Proceedings of ACM SIGCOMM '96*, pages 53–59, August 1996.
4. MRA traffic archive, September 2002. *http://pma.nlanr.net/PMA/Sites/MRA.html.*
5. M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 1997 USENIX Systems Administration Conference (LISA)*, November 1999. (software available from *http://www.snort.org/*).
6. M. Sobirey. Intrusion detection systems. *http://www-rnks.informatik.tu-cottbus.de/ sobirey/ids.html.*
7. T. Spalink, S. Karlin, and L. Peterson. Evaluating Network Processors in IP Forwarding. Technical report, Computer Science dep, Princeton University, Nov 15 2000.

8. T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 216–229, October 2001.