An Intrusion Detection Sensor for the NetVM Virtual Processor

O. Morandi, G. Moscardi, F. Risso

Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
`{olivier.morandi, giorgio.moscardi, fulvio.risso}@polito.it`

Abstract— In the wide scenario of packet processing architectures, the development of ever sophisticated applications faces the challenge of finding a balance between different requirements: ever increasing performance, flexibility, and portability of the software across different platforms and hardware architectures. The Network Virtual Machine (NetVM) aims at responding to such defy by taking into account all these elements and by providing an abstract architecture for developing today's packet processing applications. In order to demonstrate that the NetVM platform can be profitably employed for the development of complex applications, we developed a Snort-like network intrusion detection sensor. In this paper we present its architecture and show that NetVM represents an excellent target for the dynamic generation of packet processing programs.

I. Introduction

In order to keep the pace with the everyday increasing requirements in terms of throughput and flexibility, the design of high-speed packet processing applications relies always more on network processors. However, the problem of running a packet processing application on architectures other than general-purpose processors is very complex, and it gets more difficult as the complexity of the application grows. The reasons of this issue are several and of different nature: differences between the various architectures and programming paradigms (single-threaded, multi-threaded, multi-process on a multi-core processor), difficulties in exploiting the available hardware resources, like ad-hoc instructions and coprocessors. This problem is further complicated by the several different architectures available for packet processing (network processors, multi-core, systolic processors, etc.).

The problem is only apparently simplified when working with network processors that are based on a general-purpose architecture, such as the Cavium Octeon multi-MIPS processor, because a simple recompilation and minimum changes to the application enable the production of running code, in spite of sacrificing the ability to exploit the hardware resources available on the target machine, such as the Octeon's Deterministic Finite Automata coprocessors. The problem can be solved by rewriting some parts of the application, but this solution is not general, and if the architecture changes again, the work has to be restarted from scratch. In this scenario, the NetVM virtual platform [1] represents a general solution for such problems, because it provides an abstract architecture for packet processing applications, which is able to hide the differences between physical platforms.

Even though the presented advantages of the NetVM platform look promising, it must be noted that some of them are in fact only claims that need to be further explored and demonstrated. The aim of this paper is to demonstrate the first of such claims, i.e. that the NetVM is suitable for the development of complex real-world applications, and that such development could be carried out in a relatively easy manner. To prove this statement, we have built an IDS sensor that represents one of the typical applications that could run on the NetVM architecture. This application was chosen due to its requirements in terms of processing capabilities and the necessity to deal with multiple protocol layers including deep packet inspection. In addition, IDSs are suitable for hardware acceleration because of their extensive use of regular expressions and lookup tables, which are often assisted by specialized coprocessors on physical platforms. In addition, even if this will require deeper studies, this paper will point out that the code generated for the NetVM is efficient, i.e. performances are comparable with the same application running natively on the target platform.

This work is organized as follows. Section II and III present respectively the related work and a brief recap of the technologies that are the foundation of this paper. Section IV presents the architecture of our implementation, while the evaluation of the results is given in Section V. Finally, Section VI will present some conclusive remarks.

II. Related Work

The rise of network processors generated the demand for new programming models for easing the development of packet processing applications for such highly special purpose architectures, while still allowing the performances needed for keeping the pace with ever increasing line rates and traffic loads. In particular, [2] and [3] propose different software models for building complete routers through the interconnection of simple packet processing modules. [4] and

[5] propose two different packet processing languages and compilers for automatically partitioning the code to be executed on the microengines of the Intel IXP2400 network processor. While such solutions are similar to the one proposed by NetVM, they tend to focus on a specific application (e.g. packet forwarding), or on a specific architecture (e.g. Intel NPUs).

The implementation of a complete Snort-like intrusion detection sensor on a network processor was first explored by [6] that presents a compiler for generating C code from a set of intrusion signatures to be executed on an Intel IXP1200 NPU.

Since network intrusion detection relies on deep packet inspection functionalities, such as string and regular expression matching, great effort has been directed towards solutions for optimizing and offloading such processor intensive tasks through efficient algorithms and specialized hardware modules or coprocessors [7][8][9][10][11][12]. Another approach is using optimized algorithms targeted over the physical hardware platform; for example, [13] proposes a modified version of the Aho-Corasick [14] string-matching algorithm that can be executed in parallel on several microengines of the Intel IXP1200 network processor. However, our approach aims at validating the entire application instead of speeding up specific functions such as only string and RegEx matching.

III. Related Technologies

A. The NetVM Virtual Machine

The Network Virtual machine (*NetVM*) [1] is an abstract packet-handling engine that allows the portability of network processing applications across heterogeneous architectures.

In NetVM a packet-processing program is expressed as a set of modules called *Network Processing Elements (NetPEs)*, which represent virtual processors that execute a mid-level assembly language called *NetVM Intermediate Language* (*NetIL*). The interconnections between different modules determine the behavior of the entire application. Figure 1 shows an example on how a simple packet forwarding element can be implemented as a NetVM application.

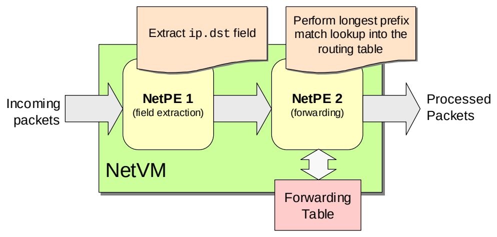

Figure 1. NetVM Based Forwarding Element.

The elementary execution engine, the NetPE, is a stack-based processor that is made up of a set of private registers (e.g. stack pointer, etc) and a memory hierarchy. The code instantiated on a single NetPE can be mapped on real processing engines on the physical hardware according to the best strategy. Execution of a NetVM program on real hardware relies on an implementation of the virtual machine, which can be an interpreter or a compiler for the translation of NetIL code to native machine code.

The NetVM has been designed to facilitate the translation of NetIL into native code: Figure 2 shows the complete architecture: a high level language (i.e. Snort rules) is used to produce NetIL code through an appropriate compiler. Then, a Just-in-Time compiler is used to produce the final binary code, which can then be executed on the target processing platform.

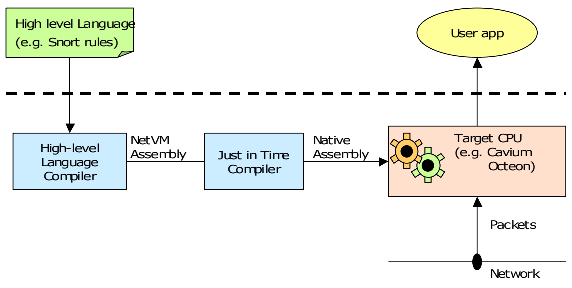
High level Language (e.g. Snort rules)
User app
High-level Language Compiler
NetVM Assembly
Just in Time Compiler
Native Assembly
Target CPU (e.g. Cavium Octeon)
Packets
Network

Figure 2. The code generation process in NetVM.

Since packet-processing applications usually rely on a subset of functionalities that are often implemented directly in hardware on many network processor architectures (e.g. Content Addressable Memories for fast table lookups, Deterministic Finite Automata coprocessors for string and regular expression matching), the NetVM architecture includes the concept of virtual coprocessors, i.e. a well-defined interface for making such features available to the programmer. An application considers coprocessors as "black boxes" providing specific operations, accessible through a well-defined interface that guarantees software portability among different platforms. On architectures that do not provide any hardware acceleration, coprocessors could be emulated by software. More details on the NetVM architecture are presented in [1].

B. Snort

Snort [15] is the implementation of a passive network IDS that is the de-facto reference in this class of applications; hence it seemed an obvious choice to design our own IDS by keeping compatibility with its rules and alerting formats. This way our IDS would get immediate benefit from the huge database of already-existing attack signatures, which would also offer an excellent testing environment.

Snort is currently capable of performing real-time traffic analysis and packet logging on IP networks. Its architecture is highly modular, and its capabilities include protocol analysis and content searching, which can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and many other security threats.

Snort uses a database of rules to describe the known attacks. Each rule describes a number of tests that should be performed on a packet, such as searching for a particular IP address or TCP port in the packet header, or matching a string or a regular expression in the payload. If all the tests specified in a rule are verified, then the corresponding action is undertaken (e.g. sending an alert and/or logging the packet). For example, the following rule:

```
log tcp any any -> 10.1.1.0/24 80 \
     (content: "GET"; msg: "HTTP GET";)
```

logs every packet coming from any host and directed to port 80 of any machine of the 10.1.1.0/24 network containing the ‘GET’ string. Such packets will be logged with a message saying “HTTP GET”.

More details on the Snort IDS can be found in [15].

IV. Architecture of the Intrusion Detection Sensor

The IDS sensor for the NetVM is not a direct port of Snort: the two applications share almost no lines of code. Our solution is based on a custom compiler that takes Snort rules and creates NetVM assembly. The internal architecture had to be redesigned from scratch in order to take full advantage of the NetVM paradigm, which tries to exploit the intrinsic modularization seen in packet-processing applications that are usually made up of several short and independent tasks. As the Snort rule format basically specifies tests that might involve the different protocols present in a packet, we decided to create different modules, instantiated on different Network Processing Elements (*NetPEs*). Tests on each protocol are performed in the NetPE responsible for it, with the exception of some special functions (such as packet analysis and pattern matching) that are not associated to a single protocol and that are allocated to specific NetPEs. The final architecture is shown in Figure 3.

For instance, the rule mentioned before will involve generation of code in different modules: the IP one will check that the destination address matches; the TCP module will be involved for checking the value of the TCP destination port, and so on. The rule will match only if all the tests are verified.

The NetPE abstraction offers the possibility of an excellent modularization: each module is almost independent, and performance can be incremented by simply improving the code generation for NetPEs that represent the bottleneck, implementing ad-hoc strategies to minimize the number of tests to be performed on a packet. For instance, some rarely used modules (e.g. ICMP) use a very simple algorithm (linear search), while others implement smarter strategies. Global optimizations can also be implemented in the NetVM framework to be able to reduce the size of the target code.

In the NetVM model, NetPEs communicate among themselves through *exchange buffers*, i.e. meta-packets that, besides the packet buffer, contain additional data (e.g. time stamps) and a dedicated area called *info partition*, where modules can store state information that flows through the NetVM following the same path of the packet. Each module composing the IDS exploits the info partition for keeping the matching state of every rule and for communicating it to subsequent modules. In particular, the info partition is divided in two parts: the former contains a bit-vector, in which every bit represents a rule, while the latter is further organised into several 32-bit slots, each one containing data extracted from the packet, such as source IP address, port, etc.

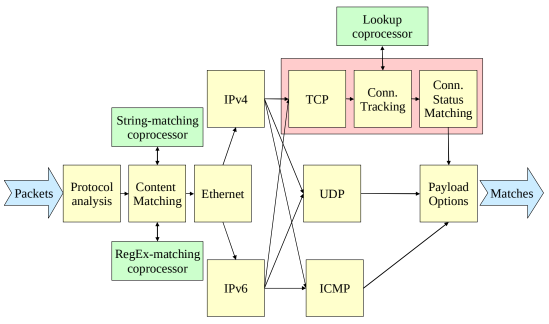

Figure 3. Architecture of the NetVM IDS sensor.

A. Packet-processing workflow

The processing of a new packet starts with the **Protocol Analysis** module that extracts information on the protocol headers present in the packet and records the starting offset of the payload. This piece of information is stored inside the “info partition” of the exchange buffer and is therefore made available to all the following modules in the chain.The next module is dedicated to **Content Matching**, which matches the payload against a set of static patterns and regular expressions specified in the source rules. This task relies on string and regular expression matching coprocessors provided by the NetVM architecture. The location of this module, almost in front of the processing chain, is due to performance reasons. In fact, the search is carried out by a modified version of the well-known Aho-Corasick algorithm [14] that allows several patterns to be searched at the same time. As a result, if a pattern is found inside the payload, only the subset of rules based on it needs to be extensively verified.

Further modules will refine the processing by performing only the tests that are required on the subset of rules that have been selected as “possibly matching” in the previous modules. For instance, the IP, TCP and UDP modules group together all the rules that have the same addresses/ports, so that they only have to check each different combination of IP and netmask once. Another optimization consists in testing the destination address/port first, and then, if it matches, the source address/port. This approach is justified by the fact that attacks come from anywhere (hence no source address is usually specified in real-world rules), while the addresses of the servers in the internal network are well-known. Testing if the packet contains a precise destination address allows discarding a large number of packets immediately, reducing the ones that need to be further processed in order to detect a match.

The **Ethernet** module only checks if the packet contains IPv4 or IPv6, and sends it to the proper module, or just discards it in case the network-layer protocol is not supported. This module does not provide any rule matching functionalities.

The **IPv4/IPv6** modules implement the tests over source and destination network addresses, while the **TCP** and **UDP** modules take care of checking the source and destination TCP/UDP ports of the packet, and the **ICMP** one checks all the possible ICMP options, which involve tests on the ICMP type, code, ID and sequence number.

The **Connection Tracking** and **Connection Status Matching** modules perform stateful TCP connection tracking, distinguishing who initiated the connection, the direction a packet is travelling in (i.e., from server to client or vice-versa) and the state of the connection (i.e., established or still in the handshake phase). This task is performed with the aid of a lookup coprocessor that acts as an associative memory holding information on the current state of active TCP connections. Finally, the **Payload** module handles the non-content payload-related options, such as tests on the payload size.

Connections among the various PEs are organized so that each incoming packet only traverses the subset of PEs dealing with the protocols it contains. This could be easily achieved through a scheme modelled after the TCP/IP protocol stack, as shown in Figure 3. This architecture has many advantages: first, each protocol is analysed only once. Second, the knowledge of a protocol is embedded in a single place, making debugging and improving the handling of a protocol easier. Furthermore, the addition of a new protocol simply requires a new NetPE to be inserted in the chain. Third, the number of traversed NetPEs is small, i.e. packets traverse only NetPEs responsible of protocols that are present in the packet (i.e. an UDP packet will not traverse the NetPE dedicated to TCP), with a clear advantage from the performance viewpoint. Fourth, the architecture is suitable for pipelining. At the moment, the application handles one packet at a time, but potentially it could handle more packets if the NetPEs can be instantiated on different physical execution units (e.g. in case of the Octeon multicore chip).

B. The code generation process

The traditional approach in intrusion detection applications is usually based on iterating over the rules that are represented in memory as complex data structures. For our IDS we decided to follow a different approach to the problem. In our implementation, rule checks are directly embedded in the code. In particular, instead of producing static programs that iterate over data structures in memory, the code directly implements all the checks needed for matching packets against the rules. Such a choice is based on the consideration that rules data remains constant throughout the execution of the program and such information can be exploited in order to emit checks (i.e. branch instructions) based on constant values (instead of checks based on values loaded from memory) producing more efficient code and opening the way to further optimizations.

V. PERFORMANCE EVALUATION

In order to assess the validity of our approach we made a series of tests using a real Snort rule database. We used an official ruleset provided by the Snort website in February 2007, which includes a total of 3058 rules, 1389 of them supported by our application. Such an apparent limitation is mainly due to the high number of rules requiring normalization and inspection of the URI field of HTTP headers (i.e. the *uricontent* option), which is a feature currently not supported by our application. We consider such number a fair one, because it includes all the rules needing deep packet inspection functionalities (i.e. string and regular expression matching), and it is in line with other research works [10][11][12].

Table 1 shows the number of NetIL instructions generated from the abovementioned ruleset for each module of our IDS. It is evident that the Content Matching module is the one with the highest number of instructions. The reason depends on the complexity of the rules involving content matching options.

TABLE 1
PROFILING THE CODE GENERATED FOR EACH MODULE

Module	Number of NetIL instructions	Number of x86 instructions	Code size (bytes)
Analyzer	137	163	613
Content Matching	38872	268.667	1.130.250
ethernet	10	20	104
ip	4531	2.057	13.991
icmp	5547	2.906	16.737
udp	4806	1.838	13.173
tcp	5127	2.100	14.442
Connection Tracking	141	261	1.271
Conn. Status Matching	6228	2.097	14.054
Total	65399	280.109	1.204.635

In order to evaluate the performance of our IDS sensor we measured the time needed to process a trace of 10M packets captured on a real network and we compared the results with those obtained running Snort under the same conditions. All the tests were performed on a Dual Xeon running at 3,4 GHz equipped with Linux 2.6.20-15 SMP. The NetVM application was compiled Just in Time into x86 assembly, while Snort was compiled through GCC version 4.1.2. Besides, all the features not supported by our IDS (e.g. flow reassembly) were disabled in Snort. The tests have been repeated 12 times, and results have been averaged excluding the best and the worst run. Results are shown in Table 2.

TABLE 2
THROUGHPUT OF THE TWO APPLICATIONS

Application	Packets/Second
NetVM IDS (interpreted)	5.634
NetVM IDS (with x86 JIT)	70.344
Snort (native)	97.922

Results look interesting. Performances of the IDS sensor interpreted by the virtual machine are discouraging, but this is expected: a virtual machine is not optimized for performance. Instead, performances obtained with the same code translated into native x86 code look promising, with our implementation running at 70% of the speed of the original Snort, although performance was not yet an objective at this stage. Differences in speed are due to several factors: the IDS code that does not implements all the performance-oriented tricks of Snort, because of the complexity of generating such code in NetIL assembly. In addition, the x86 JIT is still in an early stage and it implements only the most common optimizations, compared to the full set of optimizations implemented in GCC4. Our belief is that a more careful implementation of the JIT could further boost the performance, getting us closer to the original Snort.

VI. CONCLUSIONS

This paper presents the implementation of a network intrusion detection sensor for the NetVM platform. The objective of this work is to demonstrate that the NetVM abstraction is suitable for creating packet-processing applications both in terms of virtualized primitives (e.g. the NetPE abstraction, which enables an excellent modularization of the code) and in terms of performance.

The current status of the IDS sensor is not as mature as the original Snort. For instance, some features (such as the IP defragmenter and TCP flow reassembly) are missing, and some application-layer keywords in the rule language are not supported. However, our objective was not to create a perfect clone of Snort running on the NetVM, while creating a reasonable proof-of-concept application that demonstrates the validity of the NetVM architecture. From this point of view, results are interesting: a complex application can be implemented on the NetVM, and currently it runs at a reasonable speed. The intrinsic modularization offered by NetPEs enables the creation of complex applications, while the NetVM instruction set (and coprocessors) is adequate for packet-processing software. The biggest problem encountered in creating the IDS sensor is the lack of a high-level compiler that can be used to write applications. The data-oriented approach followed by our ruleset compiler is partially a choice, but partially a necessity because of the lack of a C-like compiler for the NetVM. This may be interesting to pursue as the next step in order to improve the programmability of our platform.

Future works will include the support for more keywords (which will enable a better coverage of the rule set), a further refinement of the NetVM development tool chain (e.g. in terms of backend compilers) in order to achieve even better performance, and the integration of this code with the NetPDL language [16], which enables the dynamic generation of code for locating protocol fields.

An extended version of this paper is available in the Technical Report DAI-NTG-2008-11, available online at http://netgroup.polito.it/pubs/pdf/2008/DAI-NTG-2008-11.pdf

REFERENCES

[1] M. Baldi, F. Risso, “Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms”, in *Proceedings of IWAN 2005*, Sophia Antipolis, France, November 2005.

[2] R. Morris, E. Kohler, J. Jannotti and M. F. Kaashoek, “The Click modular router”, in *Proceedings of the 1999 Symposium on Operating Systems Principles*. December 1999.

[3] S. Karlin, L. Peterson, “VERA: an extensible router architecture”, in Computer Networks, volume 38, number 3, 2002.

[4] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, “Shangri-La: achieving high performance from compiled network applications while enabling ease of programming”, In SIGPLAN Not. Vo 40, n. 6, 2005.

[5] R. J. Ennals, R. W. Sharp, and A. Mycroft (2005), “Task partitioning for Multi-Core network processors”. In *Proceedings of the 14th International Conference on Compiler Construction*, April 2005, Edinburgh.

[6] Y. Charitakis, D. Pnevmatikatos, E. P. Markatos, and K. G. Anagnostakis, “Code generation for packet header intrusion analysis on the IXP1200 network processor,” in *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems* (SCOPES 2003), September 2003.

[7] R. Sidhu and V. K. Prasanna, “Fast Regular Expression Matching using FPGAs”, In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM01), April 2001.

[8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, “Deep packet inspection using parallel Bloom filters,” in Hot Interconnects, (Stanford, CA), pp. 44-51, August 2003.

[9] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. “Deterministic memory efficient string matching algorithms for intrusion detection”. In *Proceedings of IEEE Infocom 200*, pages 333-340.

[10] S. Dharmapurikar, and J. Lockwood, “Fast and scalable pattern matching for content filtering”, In *Proceedings of ANCS 2005,* Princeton, NJ, USA, October 26 - 28, 2005.

[11] F. Yu, Z.. Chen, Y. Diao, T. V. Lakshman, and R. H .Katz, “Fast and memory-efficient regular expression matching for deep packet inspection”, In *Proceedings of the ANCS 2006,* San Jose, CA USA, December 03 - 05, 2006.

[12] Y. H. Cho, and W. H. Mangione-Smith, “A pattern matching coprocessor for network security”, In *Proceedings of the 42nd Annual Conference on Design Automation* (DAC 05). San Diego, California, USA, June 2005.

[13] R. T. Liu, N. F. Huang, C. N. Kao; C. H. Chen, C. C. Chou, “A fast pattern-match engine for network processor-based network intrusion detection system”, in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004)*, Volume 1, pp. 97 – 101.

[14] A. V. Aho, and M. J. Corasick, “Efficient string matching: an aid to bibliographic search”, In Commun. ACM 18, 6 (Jun. 1975), pp. 333-340.

[15] M Roesch, “Snort - Lightweight Intrusion Detection for Networks”, in *Proceedings of the 13th Systems Administration Conference (LISA '99),* Seattle, WA, November 1999, pages 229-238.

[16] F. Risso, and M. Baldi, “NetPDL: an extensible XML-based language for packet header description,” In *Elsevier Computer Networks* Volume 50, Issue 5 (April 2006), pp. 688-706.