UNITE: Uniform hardware-based Network Intrusion deTection Engine

S. Yusuf and W. Luk and M. K. N. Szeto and W. Osborne

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ
{sherif.yusuf, w.luk, man.szeto, william.osborne}@imperial.ac.uk

Abstract. Current software implementations of network intrusion detection reach a maximum network connection speed of about 1Gbps (Gigabits per second). This paper analyses the Snort software network intrusion detection system to highlight the bottlenecks of such systems. It proposes a novel packet processing engine called UNITE that deploys a uniform hardware architecture to perform both header classification and payload signature extraction utilising a Content Addressable Memory (CAM) which is optimised by techniques based on Binary Decision Diagrams (BDDs). The proposed design has been implemented on an XC2VP30 FPGA, and we achieve an operating frequency of 350MHz and a processing speed in excess of 2.8Gbps. The area resource usage for UNITE is also shown to be efficient, with a Look Up Tables (LUTs) per character ratio of 0.82 for a rule set of approximately 20,000 characters.

1 Introduction

Many mechanisms have been developed which aim to address and enhance network security, of which firewalls and NIDS are the two most common examples. Many of these mechanisms have initially been developed through software implementation, but there is increasing interest in transforming these mechanisms into hardware-based implementations to obtain improved speed performance. One hardware platform for such implementations is Field Programmable Gate Arrays (FPGAs).

Network speed and flexibility are the two major concerns for network managers; networks must be adaptable enough to accommodate the enhancement of invasive technology, without allowing the financial and performance costs of network security to spiral out of control. Contemporary networks enable extremely high data rates, and any security measures used in such networks must be capable of equal or higher data rates if they are not to degrade overall network performance. In a 'Denial of Service' (DoS) attack, network security countermeasures must process packets faster than the attacker can deliver them, but current software-based security systems cannot operate at the data rates of the networks they protect.

Attackers on network security are notorious for their ability to adapt, evolve and innovate. This requires network security to be equally, or even more, adaptable in order to effectively deal with the attacks that are thrown at it. In the case of new attacks, existing systems must be able to use emerging information about the new attack to update the system to cope with the current and future such attacks. This means a system capable of almost limitless adaptability is vital, but this should not be obtained at the cost of network performance. The software implementation of network security measures provides a high level of flexibility, but cannot offer performance even approaching the speed required by networks. An alternative hardware implementation using Application Specific Integrated Circuits (ASICs), is capable of providing much faster processing speeds, but ASICs are generally inflexible and not cost effective. The aim of this project is to develop an NIDS for deployment on an FPGA platform, with the following contributions:

- The study and profiling of Snort NIDS, demonstrating that the decoding and detection phase of Snort are the best candidates for further optimisation.
- A pipelined packet processing engine called UNITE (for Uniform hardware-based Network Intrusion deTection Engine) that employs a uniform architecture to perform both header classification and payload signature extraction utilising a Content Addressable Memory (CAM) which is optimised a Binary Decision Diagram (BDD) technique.
- The implementation of UNITE is shown to be capable of achieving a high processing speed. The results show that it is capable of handling in excess of 3Gbps traffic.

2 Network Intrusion Detection Systems

Existing research into NIDS techniques explores two different directions: header classification and payload matching. The header classification of a packet has been researched for the longest period and is well-established in theory and practise, with a focus on efficient software implementation. The payload matching of the packet is more recent, and here research concentrates more on hardware implementation. This has lead to a split between the development of these two fundamental parts of NIDS solutions: although individual research for each part may yield promising results, the final result may not be as promising when attempting to integrate different parts to develop a "complete" NIDS. This is because some techniques applied to header classification may not be directly applicable to payload matching and vice versa. There is currently limited research into the area of complete hardware NIDS, that is, those that can integrate both header and packet filtering [10, 11]. In this paper, therefore, we aim to develop a complete NIDS using a uniform technique which will strengthen and improve integration.

2.1 Profiling of Network Intrusion Detection Systems

Earlier security measures were mainly concentrated on the packet header classification, but as technology has developed, such security measures, if employed on their own, are not enough to guard against evolved attacks.

Current security systems have now extended to allow the examination of packet payload content as an extra precaution, and this allows recognised patterns or attack signatures, of intrusions to be easily detected. The attack signatures are represented as strings, and are used to match against the packet payload content by the NIDS.

In this section, a study is carried out for two NIDS: one for software and one for hardware. The motivation behind this study is to strengthen the understanding of NIDS and to aid its development. Further, this study should prove that it is necessary to develop NIDS in hardware, rather than software, for performance considerations.

Software NIDS. Snort, which is an Open-Source Software (OSS), is one of the most popular software-based NIDS systems, deployed as a security measure on many real-world networks [1]. Snort is supplied with a database of the attack signatures of many known attacks, and matches the payload of incoming packets against this database: packets with payloads that match those of malicious packets will be rejected.

Hardware NIDS. Although the performance of a hardware implementation will, in general, exceed that of software, there are still many constraints that must be taken into account when designing an NIDS. The main constraint within hardware (FPGA) is its area resource.

Hardware NIDS has been implemented through several stages [9], starting with the Snort signature list, and culminating in a compact representation of the signatures. The implementation runs at a rate of approximately 2.5 Gbps. The entire SNORT rule set was implemented on the FPGA platform with around 12% of the area on a Xilinx XC2V8000 FPGA. Although the design in runs at high speed, it does not perform header classification.

It has been shown how FPGA based multiprocessors can be used to increase the performance of network applications [10]. The system was demonstrated performing rule processing in reconfigurable hardware. They achieved a processing rate of 2.5 Gbps, with a suggestion that 10 Gbps is also possible with the most recent FPGA chips, such as the Virtex4.

Another design [11] combines and optimises the TCAM and Bit Vector algorithms for packet header classification in NIDS. The throughput of the design is also approximately 2.5 Gbps, with a combination of Block RAMs (BRAMs) and FPGA logic used for the implementation.

2.2 Snort

Snort is profiled using a Linux-based profiling tool called GProf[3]. Snort was compiled and linked with optimisations, and profiling enabled, in order to profile it with GProf. Then the execution of Snort is carried out as a normal execution, and GProf collects the data and outputs it at the end of execution. The result of most interest to us is the call graph, which provides the number of times a function in the program is called. Snort 2.4.0 was executed in NID mode with several different tcpdump data sets; some of these data sets were obtained from [14] and contained packets of a malicious nature.

The Snort system can be divided into five parts which carry out the main functions. Here we focus mainly on the *Packet Decode* and *Detection Engine*; interested readers can consult [1] for more details.

Packet Decoder. The Packet Decoder decodes the captured packets and identifies set pointers to all the different parts of information needed for the detection phase. The packet decoder decodes the packet through the protocol stack, from the Data Link layer up through to the Application layer.

Detection Engine. The detection engine takes the packet data from the packet decoder and preprocessor, and performs the detection process. The match of signature to packet is done on the transport and application layers. The matching on the transport layer is generally for checking the source and destination IP addresses and ports, or even the flags if it is TCP protocol. The application layer is for matching the payload in the packet to the attack signatures; this matching process employs the Boyer-Moore Search Algorithm [6].

Results. Table 1 shows the results of Snort analysis of several different tcpdump data sets. These data sets contain packets with activity ranging from a distributed denial of service attack run by a novice attacker (LLDOS 1.0) and a second more stealthy DDOS attack (LLDOS 2.0.2), inside sniffing (2000 Inside) and outside sniffing (2000 Outside)[14]. There is also a data set with no malicious packets (Pack100000 & Pack1000000), were captured on a student machine connected to the ethernet network in the Department of Computing at Imperial College.

Dataset	**Processing Speed (Mbps)**
1998 DARPA	105
2000 Inside	60
2000 Outside	57
LLS(DDoS) 1.0 DMZ	38
LLS(DDoS) 1.0 Inside	21
LLS(DDoS) 2.0.2 DMZ	45
LLS(DDoS) 2.0.2 Inside	44
Pack100000	22
Pack1000000	44

Table 1. Processing Speed of Snort

There is a small variation in the processing speed of Snort, as with the different data sets, there are different characteristics, and so the behaviour of Snort in each execution is quite different. The processing speed of snort is generally below 100Mbps, regardless of whether or not the data set contains malicious packets. The average processing speed of Snort reported is approximately 48Mbps.

Table 2 shows the percentage of the total execution time for each phase in Snort. The phase that uses up most of the execution time is the payload

Operation	**Percentage of Execution Time**
Alert/logging	8
Decode Frame, Packet	25
Payload matching	51
Miscellaneous	4
Preparation for rules and pattern	4
Preprocessing of packet	8

Table 2. Result for profiling of Snort

matching phase. The decoding phase uses the second most execution time, but only half the time of the payload matching process. The preprocessing phase and alert/logging phase both took approximately 8% of the execution time, and preparation for rules/patterns and miscellaneous items took approximately 4%.

The high execution time for decoding is a result of decoding every packet received, unlike other phases. The payload matching is not necessarily performed on all packets, but rather depends on the result of the decoding phase. The result of this profiling shows that the decoding and payload matching phases of Snort are ideal for optimisation, as the execution times are longer than other phases. In a hardware implementation, the decoding phase would be eliminated and its function would be integrated into the detection or payload matching phase.

3 A Uniform Hardware NIDS

This section presents the UNITE architecture. It adopts a uniform way of computing both header classification and payload signature detection utilising a Content Addressable Memory (CAM), which is optimised by techniques based on Binary Decision Diagrams (BDDs).

3.1 Design

In NIDS, some rules are often specified in terms of both header and payload. In the header section, it defines which IP source/destination address and port, and protocol to match against the packet's header. In the payload section, it defines the pattern that this rule is looking for in the packet's payload.

In the case of the header, the required fields will be located at the beginning of the IP packet and will be the same in every packet that arrives, with the fields of a fixed length. However, the signature being searched for in the payload will not always be in the same position, nor will it be of a fixed sized. Due to these differences, most research focuses on either header classification or payload matching.

Figure 1 shows a block diagram of the construction of the UNITE structure. The process, consisting of 5 stages, performs both header classification and payload signature extraction utilising a Content Addressable Memory (CAM) which is optimised by techniques based on Binary Decision Diagrams (BDDs).

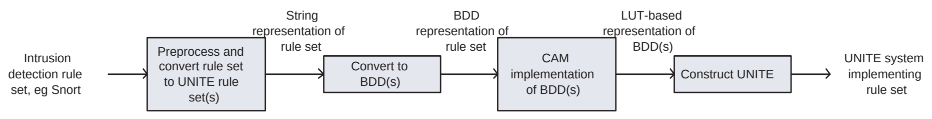

Intrusion detection rule set, eg Snort
Preprocess and convert rule set to UNITE rule set(s)
String representation of rule set
Convert to BDD(s)
BDD representation of rule set
CAM implementation of BDD(s)
LUT-based representation of BDD(s)
Construct UNITE
UNITE system implementing rule set

Fig. 1. An overview of our UNITE device

Stage 1. This stage, takes the intrusion detection (in this case Snort) signature list and parses the header and payload security requirements into a simple list of strings for the matching engine. The wildcard (or don't cares) option in the header section of the rules is a characteristic that can result in large BDDs. To combat this explosion in the BDD, we perform a preprocessing step on the rules. This preprocessing step groups the rules into smaller groups that exhibit the wildcards in the same field of the rules. In this way, the resulting BDDs with the wildcard options are shared between as many rules as possible, minimizing their impact on the overall hardware size.

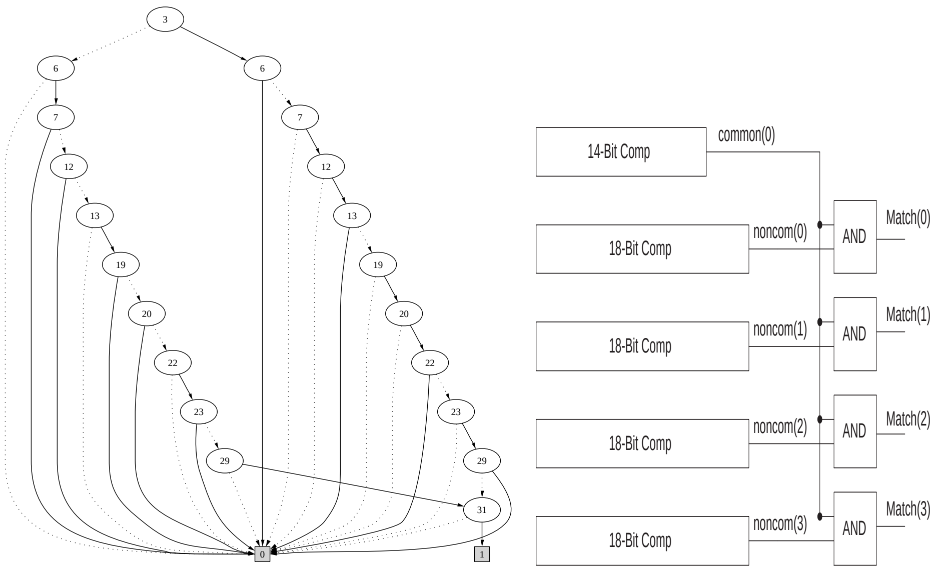
Stage 2. This stage, corresponding to the second block in Figure 1, converts the strings into a boolean expression, from which a reduced ordered binary decision diagram is generated. The string is converted to boolean expression by taking the ASCII value of each character as a parameter, then constructing an BDD representation. After all string has been converted to BDDs, the common and non-common bits of all the BDD are extracted, which are then used to build smaller BDD structures, consequently achieving a more compact hardware implementation.

Stage 3. This stage (third block in Figure 1) uses look up tables to implement an CAM. The circuits that are to be implemented are based on the common and non-common tree structures extracted from the second stage above.

Stage 4. This stage uses additional BDD manipulations on the non-common BDDs obtained in Stage 2 to further reduce the logic required in the hardware implementation. This is achieved by manipulating the two branches separately and further finding common and non-common bits within each branch to further reduce the size of the BDD structure of each branch. This stage and stage 5, both correspond to fourth block in Figure 1.

Stage 5. This final stage logically connects all sub circuits built in previous stages in order to generate an BDD-based CAM structure, which then is implemented in hardware in a highly condensed form, resulting in a much smaller area resource consumption.

Figure 2 shows the BDD representation of the non-common bits of a simple rule set (Figure 2(a)) and the corresponding LUT-based architecture (Figure 2(b)). This design results from applying stages 1 to 3 and stage 5 to the original BDD representation of the rule set. Figure 3 illustrates a typical end result of applying all five stages to the simple rule set. The labels *High Branch* and *Low Branch* relate to the optimisation phase in stage 2. In Figure 2(a), each node in the BDD leads to another node or two different nodes, and in the case where a node, x (node 3 in Figure 2) leads to two different nodes, y and z (the two node 6 in Figure 2), a logic '0' (the Low Branch) from node x leads to node y, and a logic '1' (High Branch) from node y leads to node z. Therefore, the two branches stem from the further optimisation of the *non-common* BDD stated in stage 4. Note that since we adopt a uniform way in optimising header classification and payload matching, we obtain better results than using different methods in optimising them.



(a) non-common BDD (b) LUT-based architecture

Fig. 2. Optimisation of BDD representation

For the technique to work, we only need to shift the payload section of the rules, but keep the header fields fixed. To achieve this, we developed an interface (shown in Figure 4) to the UNITE device which we use to keep the header section fixed, while still shifting the payload.

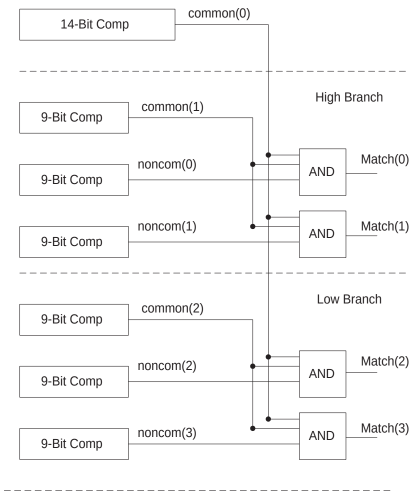


Fig. 3. A complete LUT-based architecture of a simple rule set

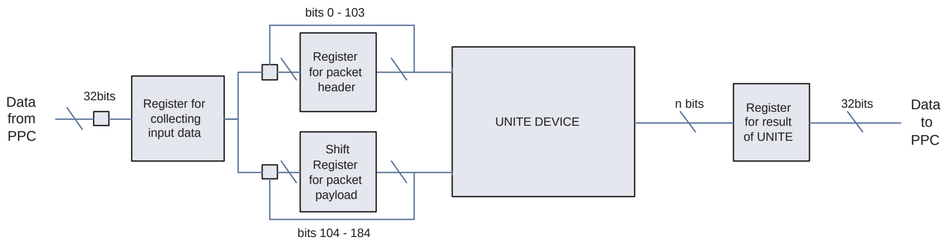

Data from PPC
32bits
Register for collecting input data
bits 0 - 103
Register for packet header
Shift Register for packet payload
bits 104 - 184
UNITE DEVICE
n bits
Register for result of UNITE
32bits
Data to PPC

Fig. 4. The interface for the UNITE device

3.2 Implementation

We develop a system capable of both header classification and payload matching, using a combination of software and hardware.

The board used is the Xilinx University Program (XUP) board[15] with a XC2VP30 [2] FPGA chip, and it consists of multiple cores which are used in this project. The cores of most interest are: PowerPC (PPC) processor [2], Ethernet controller [13], and the FPGA chip. Figure 5 illustrates the communication between the different components of the system.

The main use of the PPC processor is to run programs, written in C, to access and utilise both the FPGA logic and the ethernet controller. The PPC processor is used to receive ethernet frames from the ethernet port through the use of the

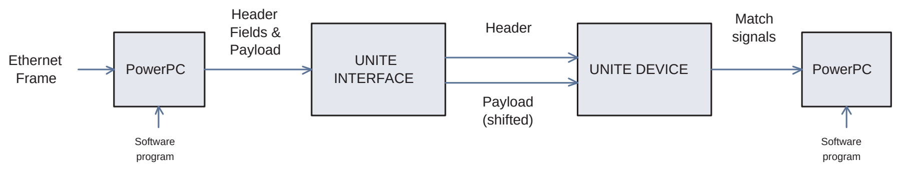


Fig. 5. Communication between component of System

EMAC core, and then to disassemble the ethernet frame and extract different fields from the frames (source/destination address and port, and protocol). These fields are of particular importance because they will be used as input to the UNITE device. The payload is then identified and extracted, and also used as input to the UNITE device.

The Xilinx library functions provide a set of functions which allow the user to manipulate the EMAC core. These functions include the ability to:

– start and stop the device,
– set the MAC address of the device,
– collect statistics of the device,
– set the receiving mode of the device(e.g broadcast, unicast).

4 Performance Evaluation

In this section, we present the results for both area resource and timing of the UNITE device.

Testing methods. The testing framework provides methods for the user to run various tests on the device. For testing, a user may wish to use "real" packet data to test the NIDS rather than reading in made up data in memory as input.

The testing framework uses a Linux program called Packit[12] to create the input data for testing. Packit is an ethernet frame construction and injection tool, and is capable of creating an ethernet frame and inject/send it down the physical ethernet interface. Packit can construct the packet from the data link layer through the network layer to the transport layer. The network layer and the transport layer are of most interest to us because network layer contains the IP addresses and the transport layer contains the source and destination ports. Users can specifies the IP addresses and ports to be the value they required as long as the value for the IP addresses and ports are of valid value within the protocol. The user can also specify the payload content for the packet to send.

One drawback of Packit is that it does not provide a function to create random payload. To overcome this drawback, a script was written to generate random or user-defined payload content for the packets and then pass the payload generated to Packit. The script can also be used for stress testing, and this is done by using loops to repeatedly generate random or user-defined packets and then send them to the NIDS using Packit. The user can specify how often the script sends user-defined packets to the UNITE device in between randomly generated packets. The user can also specify the interval between sending each packet to the UNITE device and also the total number of packets sent to the UNITE device.

Area. The UNITE implementation was compiled for a number of rule sets provided by Snort. The result for area resource usage is on average 0.82 Lut/Byte.

Snort rule set	**Number of Rules**	**Total Number of Byte**	**Number of of LUTs**	**Luts/Byte**
Finger	13	203	132	0.65
ICMP	11	488	507	1.03
Oracle	22	547	498	0.91
Porn	26	595	521	0.88
X11	2	54	34	0.63

Table 3. Area result for UNITE implementation

This is in contrast to the 0.6 Lut/Byte ratio achieved in [9], an increase of 0.22. This is acceptable, since [9] does not perform header classification. The area resource usage in [10, 11] is not comparable to UNITE as their implementation uses a combination multiple FPGA and BRAMs in order to perform the complete NIDS detection, but our UNITE only makes use of one FPGA platform.

Speed. In order to process the entire payload, it is shifted and passed to the device to perform matching again. The shifting and matching phase is repeated until the end of payload, or when the payload has an exact match to the rule set. The implementation of the UNITE device is pipelined, so this shifting and matching process will only increase the execution time by 1 clock cycle per 1 byte shifted.

An example UNITE device was implemented for the rule set of ICMP. The size of the header information is 12 bytes and the shortest payload pattern to match is 8 bytes of payload, hence needing 1455 shift operations if the 8 bytes pattern is at the very end of the maximum payload size, resulting in a total execution time of 1455 clock cycles. The clock rate for the implementation is 356MHz, and the processing speed of the device is 2.848Gbps.

Table 4 shows the results of profiling UNITE. However, only the packet capture and detection phase has been implemented in hardware, with software being used as an aid to provide the extraction of the header fields and the payload to the UNITE device.

The packet matching phase only used up 4% of the total execution time; this is a very good result because it shows that the hardware element of the system uses much less time than other software phases. This provides a major motivation in migrating the design for other software parts into hardware imple-

Operation	**Average Number of Clock Cycles**	**Percentage of Execution Time**
Frame Capture	3311	66
Frame Decoding	1147	23
Data Transfer through Bus	336	7
Packet Matching	182	4

Table 4. Result for profiling of UNITE device

mentation. The migration of software parts to hardware implementation will not only improve the speed of execution, but will also eliminate some of the communication phase, e.g. data transfer through bus, between hardware and software if all software elements are eliminated. In Table 5, we show the merits between different designs of NIDS.

Table 5. Comparison between different designs of NIDS

	Header & Payload	Area (Logic/Byte)	Throughput (Gbps)
UNITE	Both	0.82	2.85
BCAM[9]	Payload only	0.6	2.5
WashU Rule Processor[10]	Both	3.16	2.5
TCAM & BV[11]	Both	2.16	2.5
Snort[1]	Both	-	<1

5 Conclusion

This paper describes UNITE, a novel network intrusion detection engine which adopts a uniform hardware architecture to perform both header classification and payload and payload signature extraction. Both CAM and BDD techniques are used in optimising the sharing of resources in this architecture.

UNITE achieves higher processing speeds than Snort, and also shows comparable performance to the designs in [10, 11], which also support header classification and payload matching. However, UNITE is developed on a single FPGA, whereas the designs in[10, 11] uses multiple FPGAs and BRAMs for their implmentation.

We have shown that the UNITE architecture, with its simplicity and scalability, has significant potential. Its performance can be further enhanced by two means: first, arranging for multiple engines to process packets in parallel, since currently each engine only takes around 10% on an advanced FPGA; second, to migrate functions currently implemented on the processor to the FPGA, so that both the software processing speed and the hardware/software interface will no longer have an impact on performance for the UNITE approach.

Acknowledgment

The support of UK EPSRC (Grant numbers GR/R 31409, GR/R 55931 and GR/N 66599), EU Diadem Firewall project, Celoxica and Xilinx is gratefully acknowledged.

References

1. Sourcefire, "Snort - The Open Source Network Intrusion Detection System", http://www.snort.org, 2005.
2. Xilinx Inc, "Virtex II Pro Platform FPGA", http://www.xilinx.com/products/silicon_solutions/fpgas/virtex, 2005.
3. J. Fenlason and R. Stallman, "The GNU Profiler", http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono, 2005.
4. W. R. Cheswick and S. M. Bellovin, "Firewalls and Internet Security", *Addison-Wesley Professional*, 1994.
5. E. D. Zwicky, S. Cooper and B. D. Chapman, "Building Internet Firewalls (2nd Edition)", *O'Reilly*, 2000.
6. S. R. Boyer and S. J. Moore, "A Fast String Searching Algorithm", *ACM Press*, pp. 762–772, 1977.
7. D. Knuth, J. Morris and V. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Compting*, pp. 323–350, 1977.
8. G. A. Stephen, "String Searching Algorithms", *World Scientific Publishing Co., Inc.*, 1974.
9. S. Yusuf and W. Luk, "Bitwise Optimised CAM for Network Intrusion Detection Systems", *Field Programmable Logic Conference Proceedings*, pp. 444–449, 2005.
10. M. E. Attig and J. Lockwood, "A Framework for Rule Processing in Reconfigurable Network Systems", *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2005.
11. H. Song and J. Lockwood, "Efficient Packet Classification for Network Intrusion Detection using FPGA", *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2005.
12. D. Bounds, "Packit - Network Injection and Capture", http://packit.sourceforge.net/, 2005.
13. Xilinx Inc, "OPB EMAC", http://www.xilinx.com, 2005.
14. Massachusetts Institute of Technology Lincoln Laboratory, "DARPA Intrusion Detection Evaluations", http://www.ll.mit.edu/IST/ideval/data/data_index.html.
15. Xilinx Inc, "Xilinx University Program", http://www.xilinx.com/univ/, 2005.