# BITWISE OPTIMISED CAM FOR NETWORK INTRUSION DETECTION SYSTEMS

*Sherif Yusuf* * *and Wayne Luk*

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

## ABSTRACT

String pattern matching is a computationally expensive task, and when implemented in hardware, it can consume a large amount of resources for processing and storage. This paper presents a novel technique, based on a tree-based content addressable memory structure, for a pattern matching engine for use in a hardware-based network intrusion detection system. This technique involves hardware sharing at bit level in order to exploit powerful logic optimisations for multiple strings represented as a boolean expression. Our approach has been used to implement the entire SNORT rule set with around 12% of the area on a Xilinx XC2V8000 FPGA. The design can run at a rate of approximately 2.5 Gigabits per second, and is approximately 30% smaller in area when compared with published results. The performance of our design can be improved further by having multiple designs operating in parallel.

## 1. INTRODUCTION

Network Intrusion Detection Systems (NIDS) have been developed recently as a solution to increasingly complex security violations. Security measures previously relied solely upon the use of packet headers for filtering. However, this reliance has become vulnerable in many aspects, as system infiltrators modify their tactics in order to circumvent basic protection. Advanced security systems have to go beyond packet header information, and examine the actual payload content of packets traversing the network. The purpose of scrutinising payload content is to spot recognised patterns of intrusion: examples include patterns which relate to the acquisition of password storage locations.

NIDS search through a packet payload seeking known attack signatures. These attack signatures are represented as strings which are stored in the NIDS, and are matched against packet payload content in order to determine whether or not an unauthorised action is taking place. As the time taken to search through each packet for a matching string is computationally expensive, this can have an adverse effect on the network performance, especially when the NIDS is software-based. Software-based systems such as SNORT [1] employ lists of known intrusion signatures against which packet payloads are examined and, if found to be in infringement of security rules, subsequently rejected. These systems are constantly updated when information on new intrusion signatures or viruses emerges, ensuring that the system's intrusion detection does not lag behind the sophistication of security violators.

Software-based NIDS, however, suffer from one major drawback – speed limitations. This defect is exacerbated as packet payload is searched rather than the packet header. In general, software systems such as SNORT are incapable of operating at multi-gigabit rates, and they are unable to support current network speeds.

Recent strides have been made in developing hardware-based NIDS which are able to deal with increasingly fast network speeds. Despite this ability to deal with tremendous speeds, hardware-based systems have their own drawbacks – the storing and processing of signature strings in hardware-based systems often require a large amount of resources. As known intrusion signatures may number in the hundreds, and possibly thousands or more, sufficient resources are essential for their storage.

In order to solve this problem of size limitation while retaining the speed advantage of hardware, we develop a pattern matching engine which uses hardware sharing methods for signature storage. This hardware sharing method significantly reduces the amount of area required, since logic can be reconfigured as necessary. We use a tree-based Content Addressable Memory (CAM) structure for a pattern matching engine for use in a hardware-based network intrusion detection system. This technique involves hardware sharing at bit level in order to exploit powerful logic optimisations for multiple strings represented as a boolean expression, in the form of a Binary Decision Diagram (BDD).

The main contributions of this paper include:

- the separation of a single BDD into common and non-common BDDs, in order to reduce duplications or redundancies,

*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.

- a recursive optimisation algorithm that produces an efficient architecture for implementing a BDD, through the extraction of common and non-common BDDs,
- a specialised CAM for implementing intrusion detection signatures, using bit-level resource sharing, which gives results that are up to 30% more compact than other hardware approaches, while retaining the speed advantage of a CAM,
- an automated transformation of signature strings to a BDD structure, and automatic transformation from BDDs to an efficient and compact hardware implementation, called a BCAM (BDD-based CAM).

## 2. BDD-BASED CAM

Our BDD-based CAM (BCAM) structure is presented with the use of a simple example involving four strings: "qiya", "bebe", "spin", and "moki". Our example illustrates each stage in the procedure to implement a BDD in reconfigurable logic, showing the advantages of our method over two other approaches.

Our framework transforms a binary decision tree representation to an efficient implementation on a reconfigurable hardware platform through the five stages illustrated in Figure 1. With the exception of Section 3, we use the simple example to clarify the procedure in our approach.

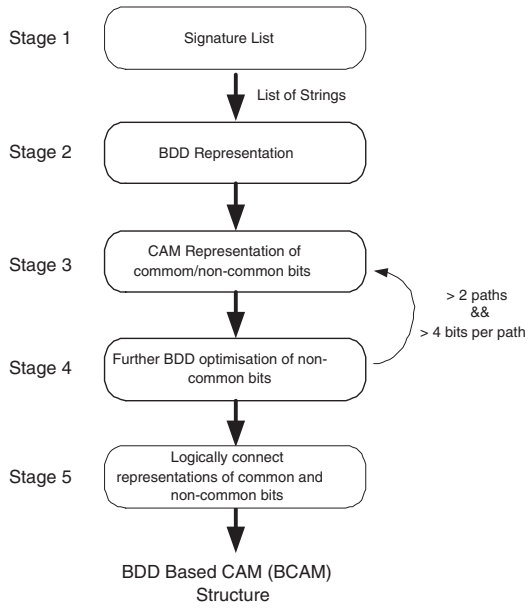



**Fig. 1**. Design flow diagram of the BCAM technique

### 2.1. Stage 1: Signature List

A signature list contains multiple signatures that indicate what patterns or sequence of bytes to look for in each packet, and based on this, an alert is generated if necessary.

The signature list for our complete experiments is obtained from the SNORT database. We start with the signature list and compile it to a simple list of strings, which contain the intrusion pattern that we are looking for in the packets. SNORT offers multiple keyword fields in the signature, such as 'distance', 'nocase' etc, which detail the number of bytes into the packet the search should be carried over, or whether the search should be case sensitive. However, we only make use of the 'content' keyword, as we are only concerned about the packet content. The other keywords are mainly there to aid the processing speed of SNORT. Our compiler extracts the strings from the 'content' keyword, and passes the new list to the second stage of our flow. For ease of explanation, we use the four simple strings listed above to explain the rest of our procedure.

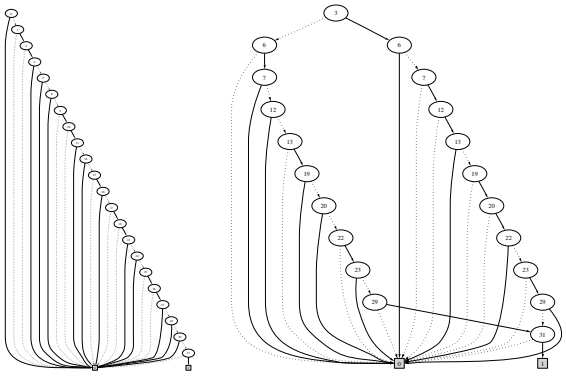
### 2.2. Stage 2: BDD Representation

Our approach converts each string representing the signature into a boolean expression, from which we generate a reduced ordered binary decision diagram. The process of generating a BDD representation of the four strings is as [2, 3]. We first create BDDs for each character in the string as above, which are then combined to construct a BDD representing the entire string. The BDD is basically a boolean representation of each character concatenated. Two values are possible from each node, a '1' or a '0', and these represent the value of that bit.

The same operations are then performed on all the given strings. Finally, all four BDDs are linked, and the end result denotes a BDD representation of the entire list of strings. For ease of expression, this final conversion of all the strings into one BDD shall subsequently be referred to as *a_list*.

With a total number of 107 nodes, the cost of implementing *a_list* would be approximately 107 look up tables (LUTs), assuming that one LUT is used to implement each BDD node or multiplexor, barring the potential optimisations mentioned in [3], such as using a karnaugh map to minimise the algebraic expressions representing the BDD.

There are redundancies that result from the fact that every time a BDD has a node with both the true and false edge leading to different nodes, then some or all of the bits from that node number are replicated on both sides. To cater for this inefficiency, we extract the common and non-common bits of *a_list* to generate two smaller BDDs, *common* and *noncommon*, where *common* has only a single path; these are illustrated in Figures 2(a) and 2(b) respectively. The following section details the course taken to implement *common* and *noncommon*.



(a) common BDD (b) non-common BDD

**Fig. 2**. BDD representation of the common and non common bits

### 2.3. Stage 3: CAM Representation

This stage of our approach produces an efficient hardware architecture for the generated BDD structures in Stage 3 above. Figure 3 demonstrates how a regular CAM implementation of the list of strings requires 4 locations. In this particular scenario, the 4 locations consist of four 8-bit characters, and hence require a 32-bit wide CAM. Using 4 input LUTs as shown in Figure 3 for the implementation of an 8-bit wide CAM comparator requires 3 LUTs per character. Hence our example requires a total of 53 LUTs, which will be compared to our BCAM usage.

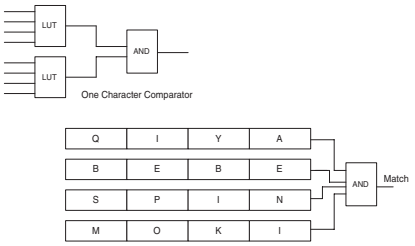



**Fig. 3**. A CAM representation of the four strings

At this stage in the design flow, we implement *common* and *noncommon*, obtained from Section 2.2, in hardware. Instead of implementing them in the traditional tree-like manner with each node indicating the path to be taken, we choose to implement them using characteristics similar to a traditional CAM, thereby eliminating the excessively large number of logic dependencies and high latency.

The *common* BDD is implemented using LUTs, similar to a traditional CAM, since there is only one path in *common*. The *noncommon* BDD generally has multiple paths, and each path leading to a terminal node is implemented using LUTs, as previously explained. Unlike the *common* CAM, the number of paths in the *noncommon* CAM is equal to or less than the number of strings in the list. The *noncommon* CAM outputs are logically ANDed with the result of the *common* CAM output. This makes up the BCAM technique. A match occurs if the input bits match the corresponding bit from *common* part and one of the paths from the *noncommon* part.

Returning to our example, Figure 4 illustrates the BCAM, showing the implementation of the *common* CAM, and how the output of the *common* is propagated to the multiple *noncommon* CAMs. If this were the end stage of our technique, we would have used a total of only 30 LUTs, a much lower cost than either the BDD [3] or CAM [4, 5] techniques on their own. However, continuing our procedure to the next stage of further BDD optimisation (Stage 4) enables us to further reduce the number of LUTs used, as described in Section 2.4.

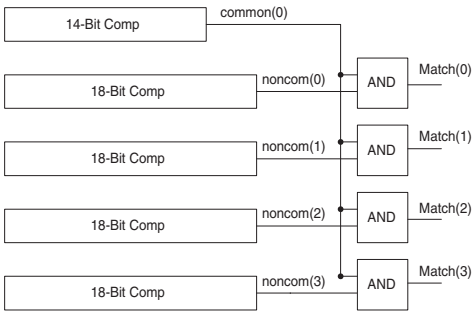



**Fig. 4**. A level 1 BCAM implementation of the simple example

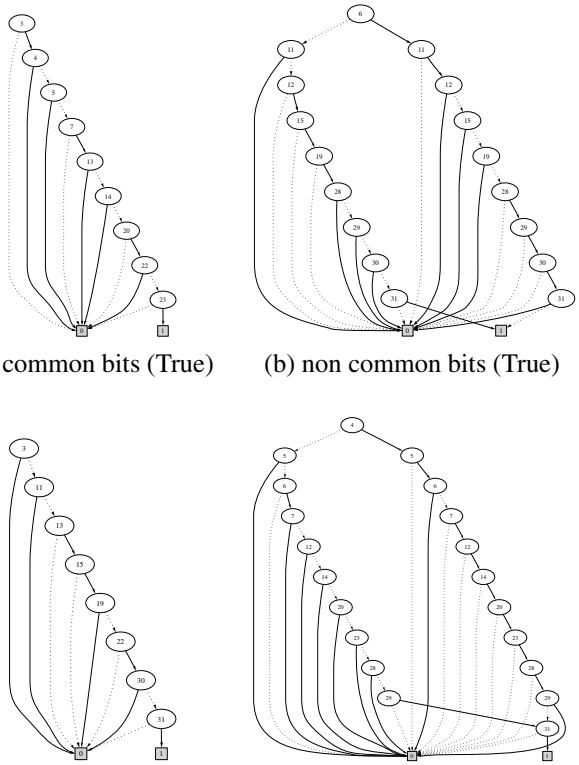
### 2.4. Stage 4: Recursive BDD Optimisation

This stage of our design flow further optimises the implementation, and is a recursive step. Figure 4 shows the first level of optimisation in the BCAM technique described in the preceding sections. The flow of design to achieve level 1 optimisations for BCAM can be repeated further on the *noncommon* BDD of level 1 to achieve further reductions.

The *noncommon* BDD is used to construct two smaller BDDs (*high* and *low*) by extracting into *high* the paths from the TRUE edge of the root node, and into *low* the paths from the FALSE edge of the root node. We then repeat the steps in Sections 2.2 and 2.3 on *high* and *low*, in order to build further CAM blocks representing our original *a_list* BDD. This process is repeated on each non-common BDD until only two paths or fewer remain in the corresponding *noncommon* BDD, or until each path only requires $x$ input bits to be implemented, where $x$ is equal to the number of input bits for a reconfigurable LUT.

The *common* and *noncommon* BDDs of *high* are shown in Figures 5(a) and 5(b) respectively, and similarly, Figures 5(c) and 5(d) illustrate the *common* and *noncommon* BDDs of *low*. Our technique uses only a total of 27 LUTs for the representation of the original list of strings, a reduc-

tion of 70% compared to the approach by [3] and 51% less than a traditional CAM approach [4, 5].



(a) common bits (True) (b) non common bits (True)

(c) common bits (False) (d) non common bits (False)

**Fig. 5**. Further optimised BDD representations

### 2.5. Stage 5: Automated Functional Block Construction

At this stage of our design process, we connect the CAM blocks obtained from extracting the common and the non-common bits, after performing Stages 1 to 3 of our BCAM technique. This end product represents our original list of string patterns as one BDD-based CAM.

We now describe the procedure to connect the CAM blocks resulting from Stages 3 to 5 of our technique. Starting from the first *common* CAM (common(0) in Figure 4), we propagate it to all subsequent CAM blocks which are obtained from the extraction process in Figure 5. The process in Figure 5 is repeated until only two possible paths remain in the *noncommon* BDD structure. Once this state has been reached, the end product represents our BDD-based CAM as applied to the original *a_list* BDD (Figure 6).

We complete the description of all five stages of our technique as applied to the original 4 strings. The end product is an automated transformation method of a BDD structure to a compact and efficient implementation in hardware as a BDD-based CAM.

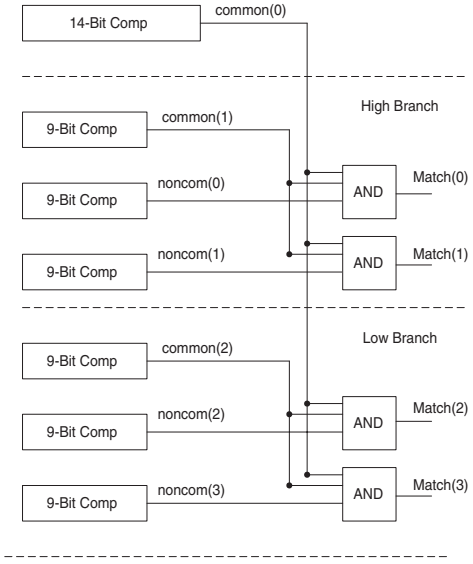



**Fig. 6**. A complete BCAM implementation of the simple example

## 3. BCAM OPTIMISED FOR HIGH THROUGHPUT

The drawback of most tree-based search systems in hardware is the high latency and also the slow speed due to long propagation delays. The main speed detriment is caused by the large fan-out from outputs of the *common* CAM blocks, which are propagated to all subsequent CAM blocks. The implementation of our BCAM structure is optimised to radically reduce long propagation delays due to fan-out, and achieve a relatively fast speed, by constructing a fan-out tree for the *common* BDDs at each level of the BCAM. From experimenting with a very large signature set, comprised of almost 2,000 strings, we find the largest fan-out to be approximately 70. Thus, we construct a fan-out tree as illustrated in Figure 7. This is done without incurring extra area resources as we use the flip flops within the configurable logic blocks. The fan-out tree for other levels can be constructed in a similar way.

The largest fan-out will always occur from the *common* BDD at level 1, and with the implementation shown in Figure 7, it can be seen that the latency is only five clock cycles. To achieve maximum operating frequency, we pipeline the BCAM structure, incurring a total latency of approximately ten clock cycles. Although we have not eliminated latency, we manage to contain it at a reasonable limit in order to achieve a satisfactory trade-off with speed. We apply this BCAM optimisation in the experiment of the large signature set above, and are able to achieve an operating frequency of 310 MHz for a Xilinx Virtex2 XC2V8000 device.

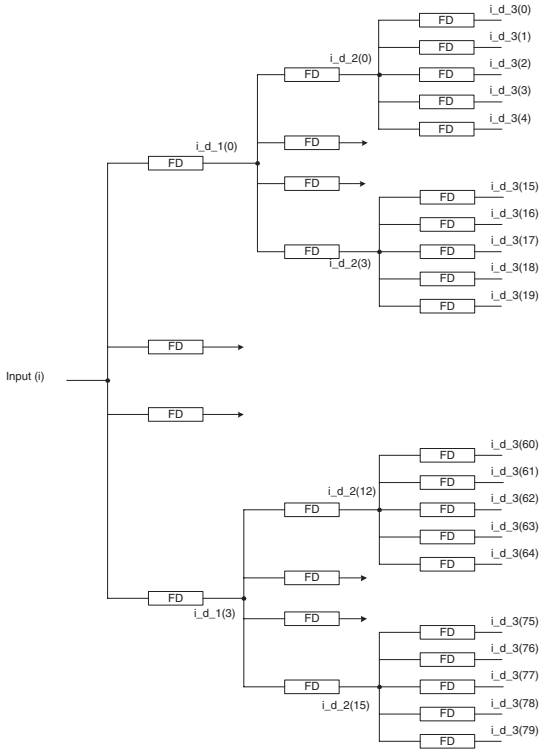



**Fig. 7**. A fan-out tree used to eliminate excessive propagation delays – only some leaves are shown

## 4. RESULTS

We use an open-source BDD package, BuDDy [6], to transform each string to its BDD equivalent. From each string, we extract a character and use its ASCII value as a parameter to a function provided by the BDD package in order to construct a BDD representation of that character. Our simple example shows that the representation of the strings using our BCAM technique is done at a cost of approximately 27 LUTs (Figure 6). Normally, where the string list is particularly large, the CAM method uses the largest amount of hardware area resource. The BDD method in general uses less area resources than the CAM, but our BCAM technique uses less area resources than the BDD approach, although according to [3], the standard BDD approach can still be optimised.

Although the standard BDD approach has a latency dependent on the location of the matching rule, there is still potential for optimising throughput by pipelining. However, the BCAM technique does not have as deep a latency, and has a significantly smaller area usage than the BDD implementation (in our simple example approximately 70% less area), even where the BDD approach is less effective than a CAM approach.

To illustrate the effectiveness of our system, we implement a realistic rule set. We use the entire Snort rule set, as of August 2004, to do so, and this clearly illustrates the gap between using our technique and using other published techniques, since some others use a subset of SNORT (Table 1). We compare our results with the nearest best result [7], LUTs/byte ratio is approximately 0.825 (this figure is obtained using the average unit size for their 16 character pattern block). In contrast, our method has a ratio of 0.6, a result which indicates area resource usage of close to 30% less than that of [7]. Furthermore, we calculate that our approach equates to an approximate unit size of 4.7, which is less than 6.6 [7].

In terms of the speed levels obtained, we compare our results to that which attained the best speed levels [8]. Whilst [8] reports speeds of up to 7.3 Gbps, at a LUTs/byte ratio of 5.3, we achieve a speed of approximately 2.5 Gbps at our ratio of 0.6 per unit. When multiple units are used (i.e. 6 units), we obtain a calculated rate of approximately 12-14 Gbps at a LUTs/byte ratio of 3.6, assuming we can maintain an operating frequency between 250-300 MHz. This throughput is achieved with a smaller resource ratio compared to [8], although there is still some indication of inevitable trade-offs between speed and resources.

Table 1 shows the comparison of our results against other FPGA-based techniques. It is clear from this table that our BCAM is the most efficient in terms of area usage. Although initial design objectives are for area efficiency, we pipeline our system to achieve speed results comparable to other techniques, and by making use of multiple BCAM engines, we estimate an improvement in speed. The main obstacle in optimising for speed is the large number of fan outs from the *common* CAMs to all *noncommon* CAMs, especially from the root CAM (i.e. *common(0)*). Thus, this is where we focus our pipelining efforts. Finally, when our BCAM structure is compared to all the other techniques discussed, on average ours is up to 30% more area-efficient than the most efficient of them [7].

To determine the effectiveness of the system as a whole, we use a metric of the throughput per LUTs/byte ratio (efficiency), and in this case the larger the ratio, the more effective the system. By using this ratio, we determine that our pattern matching engine is the most balanced system, with the closest rival being the system in [12].

## 5. SUMMARY

We illustrate our BCAM approach in a NIDS application setting, utilising a simple example of a signature set. Our multi-staged technique allows us to vastly decrease the hardware area resource needed to implement a pattern-matching engine. We are able to implement the entire SNORT rule set with around 12% of the area on a Xilinx XC2V8000 FPGA; the design can run at a rate of approximately 2.5 Gbps, and is up to 30% smaller in area when compared with published

**Table 1**. A comparison of LUTs/byte ratios amongst published techniques – Multiple PBCAM is an estimated result. NFA/DFA: Non-deterministic/Deterministic Finite Automaton; CAM: Content Addressable Memory.

| Technique | Device | Throughput (Gbps) | # Bytes | # LUTs | LUTs/Byte (LpB) | Efficiency Gbps/(LpB) |
|---|---|---|---|---|---|---|
| BCAM | XC2V8000 | 1 | 19,715 | 11,780 | 0.6 | 1.67 |
| Pipelined BCAM | XC2V8000 | 2.5 | 19,715 | 11,780 | 0.6 | 4.17 |
| Multiple PBCAM (x6) | XC2V8000 | 12-14 | 19,715 | 70,680 | 3.6 | 3.33 |
| Clark et al. (NFA) [8] | XC2V8000 | 7.3 | 17,537 | 93,180 | 5.3 | 1.38 |
| Franklin et al. (NFA) [9] | XCV2000E | 0.4 | 8,003 | 20,618 | 2.58 | 0.16 |
| Sidhu et al. (NFA) [10] | XCV1000 | 0.46 | 29 | 1,920 | 66.2 | 0.01 |
| Moscola et al. (DFA) [11] | XCV2000E | 2.5 | 420 | 14,660 | 34.9 | 0.7 |
| Cho et al. (CAM) [12] | XC3S2000 | 3.2 | 6805 | 6,136 | 0.9 | 3.56 |
| Gokhale et al. (CAM) [13] | XCV1000 | 2 | 640 | 9,722 | 15.2 | 0.13 |
| Sourdis et al. (CAM) [14] | XC2V6000 | 2.5 | 6,000 | 7,200 | 1.2 | 2.08 |
| Baker et al. [7] | XC2VP100 | 2 | 19,584 | per 16 char unit: 13.2 | 0.825 | 2.42 |

results. This performance can be further improved by having multiple designs operating in parallel.

In our technique, we also introduce a hardware sharing method which eliminates the need to work at the character-sharing level. Our approach instead focuses on the lowest possible level of sharing at bit level, an idea which is based on BDD structures. The strength of our BDD-based technique is its ability to remove replications in tree-based structures which may contribute to resource wastage, resulting in a much smaller BDD structure and a more compact hardware implementation.

## 6. REFERENCES


[1] Snort, "The Open Source Network Intrusion Detection System," 2004, www.snort.org.

[2] S. Hazelhurst, A. Fatti, and A. Henwood, "Binary Decision Diagram Representations of Firewall and Router Access Lists," University of the Witwatersrand, Johannesburg, South Africa, Tech. Rep., 1998.

[3] R. Sinnappan and S. Hazelhurst, "A Reconfigurable Approach to Packet Filtering," in *Field Programmable Logic and Applications*, Aug. 2001, pp. 638–642.

[4] J. Ditmar, K. Torkelsson, and A. Jantsch, "A Dynamically Reconfigurable FPGA-Based Content Addressable Memory for Internet Protocol Characterization," in *Field Programmable Logic and Applications*, Aug. 2000, pp. 19–28.

[5] P. B. James-Roxby and D. J. Downs, "An Efficient Content-Addressable Memory Implementation Using Dynamic Routing," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2001, pp. 638–642.

[6] J. Lind-Nielsen, "BuDDy: Binary Decision Diagram Package Release 2.0," IT-University of Copenhagen (ITU), Tech. Rep., 2001.

[7] Z. K. Baker and V. K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *Field Programmable Logic and Applications*, Aug. 2004, pp. 311–321.

[8] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2004, pp. 249–257.

[9] R. Franklin, D. Carver, and B. L. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2002, pp. 404–413.

[10] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2001.

[11] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2003.

[12] Y. H. Cho and W. H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," in *Field Programmable Logic and Applications*, Aug. 2004, pp. 125–134.

[13] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," in *Field Programmable Logic and Applications*, Sept. 2002, pp. 404–413.

[14] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2004, pp. 258–267.