Packet Filtering for FPGA-Based Routing Accelerator

David Antoš[1,2], Vojtěch Řehák[1], and Petr Holub[2,3]

[1] Faculty of Informatics,
Masaryk University Brno,
Botanická 68a, Brno 602 00, Czech Republic
[2] CESNET, z. s. p. o.,
Zikova 4, Praha 160 00, Czech Republic
[3] Institute of Computer Science,
Masaryk University Brno,
Botanická 68a, Brno 602 00, Czech Republic
`(antos|hopet)@ics.muni.cz, rehak@fi.muni.cz`

Abstract. In this paper, we present a novel approach for Binary Decision Diagram based semantically extended representation of packet filters called Filter Decision Diagrams (FDD), used for efficient filter processing and lookup in a hardware accelerator that uses a lookup engine employing CAM and comparison instructions kept in SRAM. We present the most important operations for FDDs and also give some complexity estimate. We also analyze and compare expressing power of the most commonly available packet filters.
Keywords. Packet filtering, hardware accelerated routing, filtering rules transformation, filter decision diagrams, binary decision diagrams.

1 Introduction

PC platform has shown its suitability as mid-sized routing platform [19], its performance is however limited by bandwidth of internal PC architecture. This bottleneck can be mitigated if routing can be at least partially off-loaded to an acceleration card, so that substantial part of the traffic is confined to the card and avoids common buses inside the PC.

Such routing hardware accelerator based on programmable hardware (Field Programmable Gate Arrays, FPGA) has to perform two basic operations: *packet classification* and *filtering*. In this paper, we examine the most common packet filters and analyze them with respect to their expressive power. Then we present a novel approach to packet filter data structure representation called Filter Decision Diagrams (FDD), that is an efficient representation for deciding based on packet information and which is an important step toward building an approach which integrates routing, filtering, and MAC address lookup into a single operation suitable for hardware implementation for an engine that uses content addressable memory (CAM) combined with static RAM.

The paper is structured as follows: Section 2 summarizes related work, Section 3 presents analysis of expressive power of the most commonly available packet filters, Section 4 details the proposed data structure for packet filter representation and the basic operations on it, and Section 5 summarizes the work and gives the concluding remarks.

2 Related Work

Various attempts to find representations of packet filters can be found in the literature. Taking packets as points in k-dimensional space, algorithms based on decision trees can be understood as "cutting" the space. Grid-of-tries [9], Hierarchical Intelligent Cuttings [10], Fat Inverted Segment trees [8], and HyperCuts [22] are typical examples of cutting algorithms.

Decision diagrams are graphs that search through a series of tests resulting in a terminal node containing the result. Binary decision diagrams (BDD) [4] take a single boolean variable at each node. Hazelhurst et al. [12, 11] propose a method to convert a filtering rule set into a BDD. They represent numbers as bit vectors and convert range queries using equivalence $p \leq n \Leftrightarrow \bigvee_{i=0}^{n} p = i$. Sinnappan and Hazelhurst [23] describe an approach to convert BDD representation onto an FPGA creating directly a circuit design.

Interval Decision Diagrams (IDD) [24] have integer variables in their nodes. Each node divides possible values of the variable into intervals. Each interval directs the search to a successor node. Christiansen and Fleury [6] convert filters onto IDDs. They also give a Linux prototype implementation Compact Filter [7] that can be used as kernel module. In this approach, prefix tests are converted into ranges. No concept of filter order is introduced—rules are converted to a partitioning of packet header space in a similar manner as indicated above for BDDs.

3 Expressiveness of Existing Filters

As a preliminary step for proposing the FDD structure, we study several basic questions related to packet filtering: (1) how the filters used in operating systems can be transformed for the target device, (2) whether there are differences in their "expression abilities". We analyze a set of commonly used packet filters based on the standard way to compare expression power of formalisms—expressing one filter by the means of another filter and vice versa. We take the following filters into consideration: open source BSD filters (pf packet filter [20], IP Filter [14], and IPFIREWALL (IPFW) [16]), Linux filters (ipchains [21] and iptables/netfilter [18]), and Juniper Networks Inc. routers [15] as a sample of commercial filters.

The analysis is split into three parts: (1) "inputs," i.e., on what fields filtering can be done, (2) "outputs," results of the process, and (3) the "way" how filters work.

3.1 Inputs, Outputs

All of the filtering schemes can use fields from packet headers and other “observable properties” (“observables”) of packets, like source and destination interface.

Resulting actions can be divided into basic actions and action modifiers. Basic actions typically determine whether the packet is accepted or not; a single action is mandatory for a rule. Action modifiers add extra processing and a rule may contain zero or more of them, e.g., `log`, `count`. For the accelerator, we focus on actions that can be performed by hardware, as otherwise the packet has to be sent for processing to the host operating system.

3.2 Processing of Filters

Execution Order Filters differ in rule evaluation order. IP Filter and pf are last-match filters, i.e., the last matching rule is taken as the result. Moreover, if a rule with keyword `quick` matches, it is considered the last matching. The remaining filters in our survey are first-matching, i.e., the first matching rule for a packet is applied. First-match filter can be converted to last-match one by reversing the order of rules or marking all of them as `quick`. Last-match filter containing `quick` rules can be converted to first-match based on analysing what packets semantically match the i-th rule. They (i) match the i-th rule, (ii) do not match any following rule, and (iii) do not match any preceding `quick` rule. It can be shown that if the original filter is total (i.e., it has a rule for any possible packet), the resulting filter is also total after applying rules (i–iii) and thus a partitioning on packet header space is obtained. Because of this equivalence, first-match semantics is used for the rest of the paper.

Blocks of rules can be named and used as subroutines (e.g., chains in iptables; similar functionality can be achieved with `skipto` in IPFW). The blocks can be expanded (the expansion is finite or otherwise the original filter would contain cycles). Similar approach can be taken for the class of “double match” features, i.e., matching rules that do not terminate the search (like `next term` statement in Juniper). The `next term` rule may be expanded into a sequence of conjunctions of the rule with the subsequent ones, followed by the rest of the filter.

Filter Position in the IP Stack Other differences can be found in position of filtering in the IP stack. Typical variants are an input and an output filter affecting all traffic (BSD style and Juniper). Other filters have INPUT chain for host-destined traffic, FORWARD chain for forwarded (routed) traffic, and OUTPUT chain for traffic generated by the host (Linux). Filters are either bound to an interface or global—applied to all interfaces. Per-interface filters can be joined to global filters just by adding “and interface i” to appropriate rules. In the opposite direction, global filters can be expanded per-interface. Similar conversion can be used for filters that consider input and output filters separately and filters with optional direction specification.

To convert input/output filters into the Linux-style INPUT/FORWARD/ OUTPUT scheme, we put the input filter into INPUT and output filter into OUTPUT. It ensures that locally destined and generated packets are correctly filtered. The FORWARD chain can be built as a Cartesian product of input and output filters. Considering only Accept and Deny actions, a resulting rule should accept if both parts were accepting. For richer set of actions, all combinations must be evaluated. In practice, we can obtain a more concise representation using chains: we put the input filter into FORWARD and rewrite accepting actions to calling a new FORWARD' chain that contains the output filter.

To convert Linux scheme into input/output, we need a way to describe addresses of the host machine. This may be supported by the filtering tool, e.g., IPFW has keyword `me` that is moreover evaluated when the rule is applied. Building the list of host's addresses by hand is not suitable; the addresses may change especially if multicast is used—membership in multicast groups can change quickly. Thus we propose using symbol `me` as the set of host's addresses, abstracting from how it is obtained.

Suppose that the output filter is able to test both input and output interfaces. In that case, we create the input filter containing INPUT rules enriched with "and destined to `me`," the output filter containing OUTPUT with "and sent by `me`" and followed by FORWARD rules enriched with "and not sent by `me`." If testing both interfaces is not possible in the output filter, another mechanism is needed to simulate that, such as tagging. If no such approach is possible, the input/output scheme would not be capable of processing FORWARD rules that test both interfaces.

3.3 Filtering in Hardware Accelerator

Taking into account the position of the accelerator in the system (accelerating card in a host PC acting as a common network card from the operating system point of view but switching packets by itself if possible), the card does not need to apply output filters on packets generated by the host computer, as these are filtered by the operating system (OS). Packets received by the accelerator may be forwarded by it or passed to the host OS. In the latter case, the packets are filtered by the operating system, so no hardware filtering is necessary but it is helpful in case of attacks to decrease the load on the host computer. For forwarded packets, an equivalent of the FORWARD filter must be performed by the classification engine of the accelerator. Moreover, the input filter for host-destined packets needs to be created.

We have shown that expression power of considered schemes is equal with the exception that converting INPUT/FORWARD/OUTPUT scheme into input/ output requires a way to describe interfaces of the host machine, such as term `me`. Transformations described in this paper have been employed in the Netopeer system [13] which is a configuration system for routers keeping the configuration in its repository and converting it into native languages of target devices.

4 FDD-Based Packet Filtering

The target architecture of this design is the hardware lookup engine of the hardware accelerator as available e.g., in COMBO6 [19]. The lookup engine uses a CAM for part of its search and the final resolution is finished with comparison instructions stored in static RAM. Thus designs we described in Section 2 differ from our approach in the following:

– lower granularity of data access which is more suitable for the logic of packet filtering and can be handled by CAMs (bit-level choice of variables is practically unfeasible in CAM),
– two-way branching only compared to IDDs using multi-way branching (difficult to convert to lookup instructions),
– direct rule order encoding when creating the diagram structure, while other approaches evaluate filters explicitly to the level of partitioning the header space, creating very complex rules.

Based on these properties, we define a structure called *Filtering Decision Diagram (FDD)*. The first step for the definition is the formal description of FDD variables and their classes.

Definition 1 By *FDD variables* we understand the set of all possible filtering terms of the filtering language.

We will distinguish three types of FDD variables:

1. exact match checks for protocols and interfaces, e.g., `proto tcp`,
2. prefix match checks for addresses, e.g., `saddr 147.251.54.0/24`,
3. range checks for ports, e.g., `dport 1024-65535`.[4]

The filter grammar uses the following fields: source interface `sif`, destination interface `dif`, source address `saddr`, destination address `daddr`, source port `sport`, destination port `dport`, and protocol `proto`. We assume that each filter rule is in conjunctive form; this doesn't limit generality as all "or" operations may be rewritten into sequences of rules in the conjunctive form.

All variables testing a single field will be called a *class of FDD variables.* E.g., all test of `dport` ranges belong to a single class which is distinguished from a class of `saddr` tests. □

Terminals of the diagram are actions (including modifiers) of the packet filter as described in Sec. 3.1. Moreover, we need a special terminal symbol[5] called *HSL* ("hic sunt leones"). It denotes the position of the filter that corresponds to "the remaining filter rules". During the computation of the filter, it will be overwritten step-by-step by representations of subsequent rules.

[4] For handling range queries, it might be useful to allow variables of the form `dport >= 1024`. Nevertheless, we can use tests with mandatory lower and upper bounds as all the domains are finite. We prefer not to make the theory more complex than necessary.

[5] Although *HSL* is a terminal symbol from the decision diagram point of view, it can be understood as a non-terminal symbol in the process of filter composition.

Definition 2 Multi-terminal Binary Decision Diagram (MTBDD) [2] is a rooted directed acyclic graph with two types of vertices. A nonterminal vertex u is labelled with a variable var(u) and has two successors, low(u) and high(u). Terminal vertices are labelled with elements of a finite set. □

Definition 3 A Filtering Decision Diagram (FDD) is a MTBDD over FDD variables. Its terminals are filtering actions and *HSL*.

– We say that an FDD is *finished* if it does not contain the *HSL* terminal.
– We use term *reduced* FDD (RFDD) in the same sense as for BDD (i.e., no distinct nodes with the same variable and high and low successors exist and no node has identical low and high successors).
– An FDD is *ordered* (OFDD), if all paths of the FDD respect an order of variables $<$ (variables smaller in the $<$ relation appear higher in the FDD). □

Motivation for ordering the FDD is twofold:

– it may make processing of FDDs faster by allowing to stop searching earlier in recursive procedures traversing the structure,
– it may help rewriting the structure into the format of the first-match CAM where order of columns is prescribed.

We define the functions on FDDs as order-independent as possible and the differences are commented. Three types of ordering are considered:

1. Total order. Relation $<$ is a linear order over FDD variables.
2. Class order. We prescribe the order of variable classes, e.g., we require that all `dport` tests precede `saddr` etc. The variables inside a class are incomparable.
3. No order at all, all the variables are incomparable. For the purposes of the algorithms we present, we define the $<$ relation to never hold. It allows to write the algorithms in a uniform manner. Such an FDD may not be call ordered.

The basic functions on FDDs are derived directly from standard BDD operations [1] extended with semantic improvements. Valuable notes on efficient BDD implementation can be found in [3]. The FDD procedures have been implemented by Minaříková [17].

4.1 Creating and Testing FDD Nodes

Function `FDDCreate(`n, l, h`)` is a standard procedure for creating nodes in restricted BDD. It returns a node u with var(u) $= n$, low(u) $= l$, and high(u) $= h$. The nodes are stored in a hash table and provided a suitable hashing scheme is employed, the complexity of this function is constant on average [1].

The function first tests if $l = h$. In that case, l is returned immediately in order to preserve non-redundancy. Otherwise, if the desired node is already present in the hash table, it is returned; if not, it is newly created ensuring the uniqueness property. Using `FDDCreate(`n, l, h`)` in all cases when new nodes are created guarantees that the structure remains reduced.

When total variable ordering is used, using `FDDCreate(`n, l, h`)` ensures that the FDD representation is canonical [5]. When we relax such a strict requirement, we can obtain a pair of semantically equivalent structures that cannot be unified because of they are not equal syntactically, so their equivalence cannot be recognised. This behaviour is completely identical to BDDs.

We define function `FDDIsTerminal(`n`)` that returns true if and only if the FDD rooted by node n is terminal.

4.2 Restriction

Given an FDD u (i.e., the root of an FDD) and a variable j (e.g., `saddr 147.251.54.0/0`) assigned to v (high or low), restriction `FDDRestrict(`u, j, v`)` computes an FDD based on that assignment. Intuitively, the result of the restriction function is as follows: requiring $v = \text{high}$, variable j is assumed to be satisfied and the FDD is modified so that j is replaced by its high-successor; all further tests their result follows from j assignment are eliminated. Vice versa, if $v = \text{low}$, then the FDD is modified based on j not being satisfied. The algorithm pseudocode is shown in Fig. 1.

In a standard BDD, no relationship among variables exists. This is not the case of FDDs. Knowledge about a variable may also allow restricting other variables belonging to the same class. E.g., if we know that `dport 1024-65535` condition holds, we may also restrict a condition `dport 0-25` as it is obviously false. This principle is expressed in lines 4–18 of the algorithm and illustrated in Fig. 2 where a case of interval comparison is shown.

We understand the set relations and operations as performed over sets of packets matching appropriate FDD variables. If the variable j is assigned to hold (i.e., $v = \text{high}$) and if condition $j \subseteq \text{var}(u)$ holds then testing $\text{var}(u)$ is not necessary, e.g., for $j =$ `dport 25-80` and $\text{var}(u) =$ `dport 0-1023`. In that case, the node is restricted to its high value (Fig. 2a).

If variables j and $\text{var}(u)$ are disjoint and test j holds, $\text{var}(u)$ does not need to be tested as it does not hold. Thus node u is restricted to low (Fig. 2b). In other cases, no restriction can be performed—the relationship of the variables can not be determined.

The other possibility is that $v = \text{low}$, i.e., variable j is not satisfied. Then, tests on variables represented by j may be satisfied only in the complement of j. E.g., if $j =$ `dport 0-1023`, the `dport` test may hold for values `1024-65535`. Therefore, if $\text{var}(u)$ covers the whole complement of j, which is expressed as "$j \cup \text{var}(u) =$ the whole domain of the variable," then u may be restricted to the high value as it brings no new information (Fig. 2c).

In the final case j does not hold, therefore its subset cannot hold either, so it can be restricted to $\text{low}(u)$ (Fig. 2d).

The function restricts all occurrences of variables that can be restricted under the assignment. This is the reason why we continue the recursion (returning re-

```
function FDDRestrict($u$, $j$, $v$)
   if FDDIsTerminal($u$) then return $u$
   fi
   if $v =$ high then /* $j$ holds */
      if $j \subseteq \text{var}(u)$ then
         return FDDRestrict(high($u$), $j$, $v$)
      fi
      if $j \cap \text{var}(u) = \emptyset$ then
         return FDDRestrict(low($u$), $j$, $v$)
      fi
   else /* $v$ is low, we know $j$ does not hold */
      if $j \cup \text{var}(u) =$ whole domain of variable var($u$) then
         return FDDRestrict(high($u$), $j$, $v$)
      fi
      if $\text{var}(u) \subseteq j$ then
         return FDDRestrict(low($u$), $j$, $v$)
      fi
   fi
   if $\text{var}(u) < j$ then /* the variable is surely not present any more */
      return $u$
   else
      return FDDCreate(var($u$),
         FDDRestrict(high($u$), $j$, $v$), FDDRestrict(low($u$), $j$, $v$))
   fi
end function
```

Fig. 1. FDDRestrict(u, j, v)

sults of the FDDRestrict(high(u) or low(u), j, v)) in the 4–18 block. Successors of the node can contain a variable that can be further restricted.

The important problem is when the search may be terminated. The commands on lines 19–20 serve to optimise the run of the procedure using the properties of variable order. The procedure works correctly even if no order is defined—the $<$ relation is defined to never hold then. If class order is used, the recursion may stop when we leave the relevant class. When total order is defined, this condition acts completely as used for ordered BDDs [1]. In all cases, the order of variables is preserved as this procedure may only delete nodes and it never creates new ones.

4.3 Converting a Filtering Rule to an FDD

Let us have packet filter F consisting of rules F_i for $1 \leq i \leq \text{Size}(F)$. By Action($F_i$) we denote the action related with rule F_i.

Converting a filtering term from F_i into an FDD is straightforward. Each term is rewritten from the rule into an FDD variable with the same test. The high branch of the test leads to terminal Action(F_i), the low branch to *HSL*, cf. Fig. 3. In its upper part, components of filtering rule "`dport 0-1023 saddr 147.251.54.0/24 sif 1 accept`" are shown.

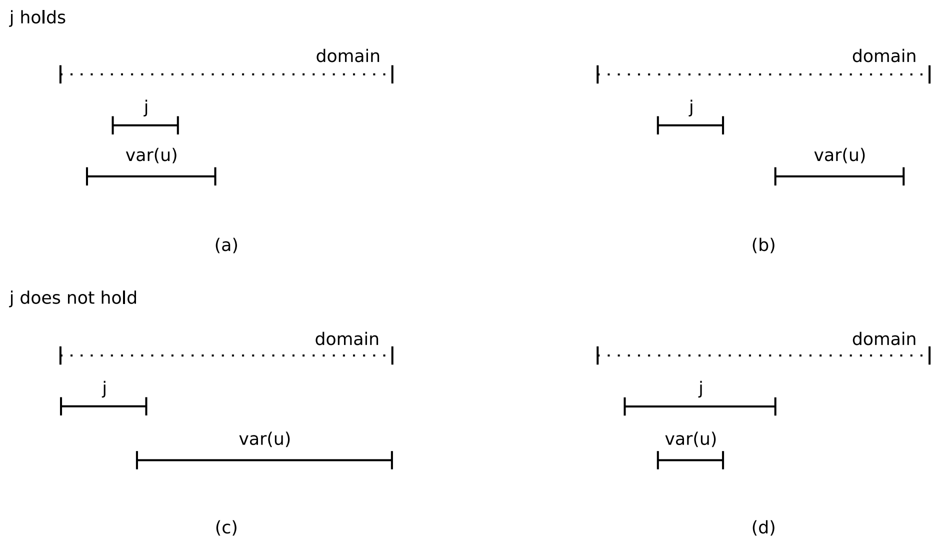

Fig. 2. Principles of FDD Restriction

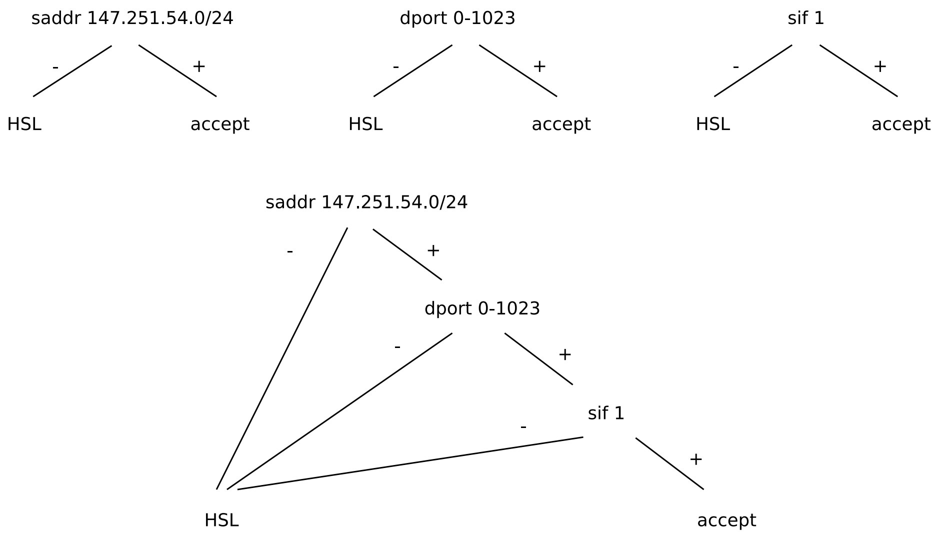

Fig. 3. Converting a filtering rule to FDD

```
function FDDAnd($u_1$, $u_2$)
    /* solve terminal cases */
    if $u_1 = HSL$ or $u_2 = HSL$ then return $HSL$
    fi
    if FDDIsTerminal($u_1$) then return $u_2$
    fi
    if FDDIsTerminal($u_2$) then return $u_1$
    fi
    /* perform Shannon expansion on the smallest variable */
    $h$ = smallest of nodes $u_1, u_2$ in relation $<$
    $u_1^h$ = FDDRestrict($u_1$, var($h$), high)
    $u_1^l$ = FDDRestrict($u_1$, var($h$), low)
    $u_2^h$ = FDDRestrict($u_2$, var($h$), high)
    $u_2^l$ = FDDRestrict($u_2$, var($h$), low)
    return FDDCreate($h$, FDDAnd($u_1^h$, $u_2^h$), FDDAnd($u_1^l$, $u_2^l$))
end function
```

Fig. 4. FDDAnd(u_1, u_2)

The terms are combined together by FDD function FDDAnd(u_1, u_2) in Fig. 4. It computes an FDD equivalent to the conjunction of the terms. This function is based on standard "Apply" BDD procedure [5] for computing a logical function of a BDD pair. Refinements to this function are related to the fact it is only used to combine FDDs where all terminals are equal or *HSL*—the terminals are taken from a single filtering rule. The advantage of this approach over using the standard and more general Apply function is that relationship of all terminal symbols does not need to be solved.

Note that supposing non-*HSL* terminals in u_1 and u_2 are identical, this function is commutative.

Terminal cases are tested first. If one of the terminals is *HSL*, the result is *HSL*, too. If a terminal is reached, the remainder of the other structure may be simply appended. If a variable order is prescribed and the parameters are ordered, the order is preserved by this step.

We shall precise meaning of line 10 of the algorithm. If no variable order is used, let us understand the choice as "choose the first available variable, say, from u_1." For class variable order, a variable is chosen from the lowest class available. For total variable order, the possibility of choice is abandoned completely— the lowest available variable has to be taken. Moreover, as all new nodes are created with the FDDCreate(n, l, h) function, the resulting structure is reduced. Regardless of variable choice, Shannon expansion is used to propagate the computation to child nodes.

In higher-level procedures, we will use notation FDDConvertRule(f) for the function that converts a firewall rule f into an FDD by means of applying methods described above. It returns the root of the FDD representing rule f.

```
function FDDAppend($u_1$, $u_2$)
    if $u_1 = HSL$ then return $u_2$
    fi
    if FDDIsTerminal($u_1$) then return $u_1$
    fi
    /* perform Shannon expansion on the smallest variable */
    $h =$ smallest of nodes $u_1, u_2$ in relation $<$
    $u_1^h =$ FDDRestrict($u_1$, var($h$), high)
    $u_1^l =$ FDDRestrict($u_1$, var($h$), low)
    $u_2^h =$ FDDRestrict($u_2$, var($h$), high)
    $u_2^l =$ FDDRestrict($u_2$, var($h$), low)
    return FDDCreate($h$, FDDAppend($u_1^h$, $u_2^h$), FDDAppend($u_1^l$, $u_2^l$))
end function
```

Fig. 5. `FDDAppend(`u_1, u_2`)`

4.4 Converting a Rule Set to an FDD

Suppose we have an FDD representation of filtering rules from the first up to rule i. Now, we present a procedure to add rule $i + 1$ to the FDD.

Function `FDDAppend(`u_1, u_2`)` shown in Fig. 5 searches for the *HSL* terminal in the u_1 FDD and replaces it with u_2. It is also the principle of handling terminal cases in the algorithm. If *HSL* is found in u_1, it is rewritten to u_2. When another terminal is reached, it is returned.

The propagation through the structure is again done using Shannon expansion. Discussion of relationship of variable order to the expansion is completely identical as for function `FDDAnd(`u_1, u_2`)` described in Section 4.3.

Converting a whole filter to its FDD representation is shown in Fig. 6. The algorithm starts with the *HSL* terminal and rules are applied one by one. The resulting FDD is finished (i.e., it does not contain the *HSL* terminal) since the last filtering rule is the default rule. Adding the last filtering rule, the *HSL* terminal of so-far-processed filters is rewritten into the default action (it may have been rewritten earlier if some of the preceding rules was default—in that case the computation could have been stopped at that point as all subsequent rules are unreachable anyway).

```
$u = HSL$
for $i = 1$ to Size($F$) do
    $f =$ FDDConvertRule($F_i$)
    $u =$ FDDAppend($u$, $f$)
done
```

Fig. 6. Converting a filter to the FDD representation

4.5 Complexity of Operations on FDD

We supposed that the filter is a sequence of filters containing only conjuncts of terms. Let m be the number of filters and n maximal number of terms in a rule. Then evaluation of the filter in software needs $O(mn)$ tests in the worst case. The number of tests in the FDD is limited from above by the same expression. We can easily see this if the FDD is not ordered nor reduced: then its structure corresponds directly to lazy evaluation of packet filter in software. Properties of reduction and ordering of nodes find identical nodes and unify them, decreasing the number of tests needed.

4.6 Implementation Notes

In a practical implementation, the length of addresses is too large to treat the addresses as single entities in hardware, mostly in case of IPv6. Therefore we propose the addresses are split into sequences of registers and the registers to act as FDD classes in FDD processing. It has no effect on the functions themselves so we decided to hide this detail in the theory in order not to add extra complexity to the text.

The structure can be rewritten into CAM and comparison instructions as follows. Each column of CAM can test a class of variables. The algorithm searches for such a variable, restricts the FDD based on its value and fills it into CAM row. It recurses until all columns of CAM are filled, creating first-match representation in CAM. The remainder of the structure is directly rewritten into SRAM comparison instructions. The main advantage of this design is that searching CAM and evaluating the instructions can be pipelined.

5 Conclusions

In this paper, we have presented an analysis of expressive power of various commonly available packet filtering systems and proposed a novel data structure for packet filtering. The expression power of considered schemes has beed shown to be equivalent with the exception that converting INPUT/FORWARD/OUTPUT scheme into input/output requires features mentioned in Section 3. Based on this knowledge, we have designed a semantically enriched BDD-based structure called FDD, which allows efficient lookup of the packet filtering decision. The basic operations have been described that are necessary for constructing the FDDs from the common packet filtering systems. We also comment on implementing the FDDs into the hardware implementation based on CAM and static RAM. We consider the FDDs as a very important first step toward unified ARP-lookup/routing/filtering operation to increase the performance of routing platforms that support both packet routing and filtering.

Acknowledgements

This project has been kindly supported by a research intent "Optical Network of National Research and Its New Applications" (MŠM 6383917201).

References

1. Henrik Reif Andersen. An Introduction to Binary Decision Diagrams, April 1998. Department of Information Technology, Technical University of Denmark, Lyngby, `http://www.it.dtu.dk/tilde-hra/bdd97.ps`.
2. C. Baier and E. Clarke. The Algebraic Mu-Calculus and MTBDDs. In *5th Workshop on Logic, Language, Information and Computation (WoLLIC'98)*, pages 27–38, 1998.
3. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE conference on Design autamation*, pages 40–45, Orlando, Florida, USA, 1991. IEEE/ACM, ACM Press, New York, NY, USA. ISBN 0-89791-363-9.
4. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986. ISSN 0018-9340.
5. Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
6. Mikkel Christiansen and Emmanuel Fleury. An Interval Decision Diagram Based Firewall. In *Proceedings of 3rd IEEE International Conference on Networking (ICN '04)*, Gosier, Guadeloupe, French Caribbean, 2004. University of Haute Alsace, Colmar, France. ISBN 0-86341-325-0.
7. Compact Filter: An IDD Based Packet Filter for Linux, August 2005. `http://www.cs.aau.dk/tilde-mixxel/cf/`.
8. Anja Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification. In *Proceedings of INFOCOM*, volume 3, pages 1193–1202. IEEE, 2000.
9. Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 147–160, Cambridge, Massachusetts, United States, 1999. ACM Press, New York, NY, USA. ISBN 1-58113-135-6.
10. Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, January/February 2000.
11. Scott Hazelhurst, Adi Attar, and Raymond Sinnappan. Algorithms for Improving the Dependability of Firewall and Filter Rule Lists. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 576–585, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0707-7.
12. Scott Hazelhurst, Anton Fatti, and Andrew Henwood. Binary Decision Diagram Representations of Firewall and Router Access Lists. Technical Report TR-Wits-CS-1998-3, University of the Witwatersrand, Johannesburg, South Africa, October 1998.
13. Petr Holub. XML Router Configuration Specifications and Architecture Document. Technical Report 7/2002, CESNET, 2002.
14. IP Filter, March 2005. `http://coombs.anu.edu.au/~avalon/`.
15. Juniper Networks, Inc. JUNOS Internet Software for J-series, M-series, and T-series Routing Platforms: Policy Framework Configuration Guide, February 2005. `http://www.juniper.net/techpubs/software/junos/junos71/index.html`.
16. Kurt J. Lidl, Deborah G. Lidl, and Paul R. Borman. Flexible Packet Filtering: Providing a Rich Toolbox. In *Proceedings of the BSDCon '02 Conference on File and Storage Technologies*, pages 99–110, Cathedral Hill Hotel, San Francisco, California, USA, February 11–14 2002. USENIX.

17. Kateřina Minaříková. Computing Look-up Programs of Routing Accelerator. Master's thesis, Faculty of Informatics, Masaryk University Brno, 2005.
18. The netfilter/iptables project, March 2005. `http://www.netfilter.org/`.
19. Jiří Novotný, Otto Fučík, and David Antoš. Project of IPv6 Router with FPGA Hardware Accelerator. In Peter Y.K. Cheung, George A. Constantinides, and Jose T. de Sousa, editors, *Field-Programmable Logic and Applications, 13th International Conference FPL 2003*, volume 2778, pages 964–967. Springer Verlag, September 2003.
20. PF: The OpenBSD Packet Filter, March 2005. `http://www.openbsd.org/faq/pf/`.
21. Rusty Russel. Linux IP Firewalling Chains, March 2005. `http://people.netfilter.org/~rusty/ipchains/`.
22. Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet Classification Using Multidimensional Cutting. In *SIGCOMM'03: Proceedings of the Applications, Technologies, Architectures, and Protocols for Computer Communication Conference*, pages 213–224, Karlsruhe, Germany, 2003. ACM Press, New York, NY, USA.
23. Raymond Sinnappan and Scott Hazelhurst. A Reconfigurable Approach to Packet Filtering. In Gordon J. Brebner and Roger Woods, editors, *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, volume 2147 of *LNCS*, pages 638–642, Belfast, Northern Ireland, UK, August 2001. Springer Verlag. ISBN 3-540-42499-7.
24. Karsten Strehl and Lothar Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 686–692, San Jose, California, United States, 1998. ACM Press, New York, NY, USA. ISBN 1-58113-008-2.