HiPAC
High Performance Packet Classification for Netfilter

Diplomarbeit

Nach einem Thema von
Prof. Anja Feldmann, Ph.D.
am Fachbereich Informatik
der Universität des Saarlandes

von

Thomas Heinz

Saarbrücken, 28. Februar 2004

This thesis is dedicated to my parents whose enduring support allowed me to fully devote myself to the fascinating aspects of high performance packet classification. It is also dedicated to Michael Bellion whose ceaseless, unswerving faith in the success of HiPAC made it possible to overcome the dispiriting dead ends during the development.

"In theory, there is no difference between theory and practice. But, in practice, there is."
Jan L.A. van de Snepscheut

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und nur die angegebenen Quellen verwendet habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Thomas Heinz
Saarbrücken, 28. Februar 2004

Contents

1 Introduction

1.1 Motivation

Packet classification is the process of dividing network packets into different flows. The precise definition of the term and the concepts behind have very much evolved over the past years. A system or algorithm providing packet classification services is called packet filter, packet classifier or flow classifier. The first proposal of a packet filter is in [MRA87] (CMU / Stanford packet filter – CSPF) which describes a programmable execution model for applying boolean operations to a stream of packets in order to select a certain subset. CSPF falls in the category of imperative packet classifiers which are represented as a sequence of instructions running on a dedicated virtual machine. Further representatives of this category are [MJ93] (BSD packet filter – BPF), [YBMM94] (Mach packet filter – MPF), [BGP+94] (PathFinder) and [BMG99] (BPF extensions and optimizations – BPF+). An alternative specification proposed by [JCSV94] defines the packet filter on the basis of a context-free grammar. The applications of early packet filters are mainly network monitoring, traffic collection, performance measurement and demultiplexing of protocols implemented in user space. The main drawback of these approaches is their generality leading to a lack of efficient algorithms meeting the ever-growing performance demands involving increasingly complex filters.

Hence, a new packet filter definition has been introduced centering around the so-called *packet classification problem (PCP)* which is also the basis of this thesis. The specification of a classifier encompasses a set of rules where each rule determines the set of packets for which it applies. For this purpose, a rule defines a set of values – called *match* – for certain packet header fields, e.g. source IP, destination IP and transport layer protocol field of the IPv4 header. The number of different packet header fields considered in a rule set is called *dimension*. This term is also used to specify the match category itself, e.g. source IP dimension. Given a rule and a packet, the rule is said to apply or match the packet if the values of the packet header fields are contained in the corresponding match. Instead of an arbitrary set, a match is usually defined as prefix, range or value / mask pair. Each rule is associated with an application dependent action which determines how the packet should be handled, e.g. drop / forward the packet or associate it with a certain service class. If a packet applies to more than one rule, a conflict resolution scheme must be used. A common approach is to assign unique priorities to the rules and select the highest priority rule matching a packet. An example for an alternative conflict resolution is the traditional IP forwarding which is an instance of the one-dimensional PCP where each rule consists of a single destination IP prefix match. Conflicts are resolved by selecting the rule with the longest prefix match among the matching rules.

Nowadays, the field of application requiring packet classification is very extensive. The main motivation is to change the original best-effort principle of the Internet towards a flexible infrastructure providing security, privacy and quality of service (QoS) along with high performance and reliability. As an example, traditional IP forwarding solely based on the destination IP is superseded by policy routing (policy based forwarding) which additionally takes into account source IP, protocol, ports etc. Other applications requiring packet classification are virtual private networks (VPN), traffic shaping (e.g. rate limiting), traffic accounting, differentiated services, network address translation (NAT) and firewalls. The latter is a collective term for a number of services which regulate the network traffic to and from a private (e.g. corporate) network. The packet filter is the basic building block of a firewall. Other elements are proxies, circuit-level and application-level gateways.

Different applications impose different requirements on packet classification systems. Hence, there is a number of (performance) metrics to evaluate PCP algorithms.

classification performance: Most applications require packets to be processed at wire speed, e.g. on an OC-192 link (10 GBit/sec) more than 14 million packets per second may be transmitted.

update performance: There are static, incremental and dynamic PCP algorithms. The first category requires the data structure to be rebuilt from scratch for each update, the second allows dynamic insertions whereas deletions require a rebuild and the third allows dynamic insertions and deletions.

memory usage: Scalable space complexity leads to high cache efficiency and allows very fast SRAM or similar memory technologies to be deployed.

number of dimensions: There are specialized algorithms for the one- and two-dimensional PCP and algorithms supporting an arbitrary number of packet fields.

match specification: The match structure determines the expressiveness of a rule. Commonly, prefix, range or mask matches are offered.

implementation flexibility: Some algorithms are dedicated for software implementation, others for hardware. There are algorithms which suit both.

Unfortunately, there is no “perfect” PCP algorithm, i.e. one which performs well in all categories, supports an arbitrary number of packet fields and offers a flexible match specification (not just simple point matches). There are very efficient algorithms for both the one- and two-dimensional PCP. However, the multi-dimensional PCP is considered an inherently hard problem. The problem is at least as hard as the so-called point location problem ([GM99a]) for which the best algorithms known from computational geometry either yield $O(n)$ space along with $O(log^d(n))$ time or $O(n^d)$ space along with $O(d \cdot log(n))$ time where n is the number of rules and d the number of dimensions.

This thesis presents a novel, multi-dimensional PCP algorithm providing dynamic operations, a flexible match specification based on ranges and high classification performance ($O(d \cdot log(w))$ where w is the bit width of the largest packet field). The worst case space complexity is $O(n^d)$. Yet, this worst case is only achieved by artificial rule sets without practical relevance.

The algorithm is implemented within the scope of the packet classification software project **HiPAC** (**hi**gh **p**erformance p**a**cket **c**lassification). The implementation has been developed in equal shares by Michael Bellion ([Bel04]) and me. HiPAC provides a very efficient packet filter implemented as kernel module on top of the netfilter framework ([RBM+99]) which is included in the linux 2.4 kernel. Netfilter offers a set of so-called hooks inside the linux forwarding path which are used to intercept network packets and influence their further processing, e.g. drop them. HiPAC's feature set is almost 100% compatible with iptables, the linux 2.4 packet filter which implements a naive linear classification algorithm. In particular, HiPAC supports the iptables concepts exceeding the expressiveness of PCP which motivates the formulation of a generalized packet classification problem, namely the non-terminal packet classification problem (NPCP). Essentially, NPCP differs from PCP in that given a packet, multiple rules are allowed to apply and not only the highest priority one. HiPAC's compatibility with iptables is also reflected by the user space program nf-hipac which is used to view and modify the rule set and which is entirely syntax compatible to the iptables user space tool except for some minor aspects. Thus, HiPAC is a drop-in replacement for iptables which eliminates the performance bottlenecks of iptables without restricting its flexibility. The source code is available at `http://www.hipac.org/`.

1.2 Related work

In recent years, PCP has received a lot of attention from research community. [GM01] provides an excellent overview of the results achieved so far including a description of each algorithm along with an example. The paper distinguishes between four algorithmic categories: basic, geometric, heuristic and hardware algorithms. Additionally, the algorithms differ in the number of supported dimensions. One-dimensional PCP algorithms focus on the IP lookup problem. [WVTP97], [MB01], [SV99], [PA01] and [SVW01] are some representatives of this category. [WVTP97] describes binary search on hash tables organized by prefix lengths achieving $O(log(w))$ lookup time where w is the bit width of the packet field[1]. [MB01] and [SV99] aim to optimize the lookup by improving the hashing technique resp. reducing the number of distinct IP prefix lengths. [PA01] proposes a hardware solution based on binary decision diagrams (BDDs) which are mapped onto a pipeline of SRAM banks. The paper also outlines a solution for the multi-dimensional PCP but uses a different PCP definition which simplifies the use of BDDs. [SVW01] proposes an algorithm supporting fast updates $(O(log(n)))$ along with moderate lookup time $(O(log(n)))$ where n is the number

[1] Note that this lookup time is also achieved by the HiPAC algorithm for the one-dimensional PCP without being restricted to prefix matches.

of rules (prefixes). The following paragraphs outline some well-known two- and multi-dimensional PCP algorithms according to the aforementioned categorization. If not stated otherwise, the algorithms can be assumed to solve the multi-dimensional PCP. Time or space complexity specifications may involve the variables n, w and d where n is the number of rules, w is the bit width of the largest packet field and d is the number of dimensions.

Basic algorithms: The simplest PCP algorithm uses linear search over the rules which results in $O(n)$ time and space. This algorithm is sometimes augmented with a rule cache which is only effective if the network traffic is rather homogeneous and the number of parallel flows does not exceed the cache size. The other algorithms in this class are based on tries, i.e. multi-way search trees whose edges are labeled with 0 or 1 (in general an arbitrary alphabet may be used). Tries entail the limitation that only prefix matches may be used. Hierarchical radix tries require $O(ndw)$ space and $O(w^d)$ classification time which is far from being sufficient. Set-pruning tries achieve $O(dw)$ query time using $O(n^d dw)$ space. [GM00] presents two dynamic trie algorithms, namely heap-on-trie and binarysearchtree-on-trie. The former achieves $O(log^d(n))$ classification and $O(log^{d+1}(n))$ update time while the latter provides $O(log^{d+1}(n))$ classification and $O(log^d(n))$ update time. Both approaches require $O(nlog^d(n))$ space.

Geometric algorithms: This class of algorithms interprets a d-dimensional PCP rule set as a number of potentially overlapping, d-dimensional hyper-rectangles. A packet is a d-dimensional point for which the classification algorithm returns the highest priority hyper-rectangle containing the point. [SVSW98] proposes the grid-of-tries and cross-producting algorithms. The former addresses the two-dimensional PCP. Based on hierarchical radix tries, the algorithm reduces classification time to $O(w)$ by inserting so-called switch pointers into the trie reducing the number of lookup paths to a single one. The cross-producting algorithm constructs a so-called cross-product table which contains all possible combinations of prefixes. On classification, the best matching prefix is computed for each dimension separately and the result is combined to form a key which is associated with the entry in the cross-product table containing the best matching rule. This approach is only suitable for very small rule sets due to $O(n^d)$ space complexity which does not only apply to artificial rule sets but particularly to real world rule sets. [BSW99] presents the area-based quad tree algorithm for two-dimensional PCP. The basic idea is to recursively decompose the two-dimensional space into equally sized quadrants until each quadrant contains at most one rule. The algorithm achieves $O(\alpha w)$ query and $O(\alpha \sqrt[\alpha]{n})$ update time using $O(nw)$ space where α is a tunable parameter. [FM00] presents the FIS-tree (fat, inverted, segment tree) algorithm for two-dimensional PCP. The algorithm is based on the reduction of the one-dimensional PCP to the range location problem (RLP) which is also present in [LS98] and the HiPAC algorithm. The reduction involves projecting the endpoints of the ranges onto the universe and thus creating a range partition. The range partition of the first dimension is augmented with an inverted tree of l levels where each node points to an RLP instance of the second dimension. The

algorithm achieves $O((l+1)log(w))$ query time using $O(ln^{1+1/l})$ space. An extension towards multi-dimensional PCP is sketched requiring $O(l^{d-1}log(w))$ query time which obviously does not scale well for $l > 1$.

Heuristic algorithms: This class of algorithms exploits certain properties of real world rule sets to improve the average case. [GM99a] presents an often cited statistical evaluation of real world rule sets along with the recursive flow classification (RFC) algorithm which attempts to map the $O(2^{dw})$ possible packet headers to their corresponding action. The algorithm uses several parallel lookup phases, each mapping a set of values to a smaller set and thus reducing the number of relevant bits after each phase. [SSV99] proposes the tuple space search algorithm which divides the rule set into subsets of rules, each associated with a unique tuple of prefix lengths. Each match of the rules contained in such a subset is a prefix whose length is equal to the corresponding component of the tuple associated with the subset. The subsets are represented by separate hashes which are queried linearly for a given packet. [WSV01] proposes an algorithm based on the tuple space idea for two-dimensional conflict free filters, i.e. rule sets where each packet applies to at most one rule. [GM99b] proposes the hierarchical intelligent cuttings (HiCuts) algorithm which constructs a decision tree where each node, corresponding to a certain dimension, is split into equally sized ranges using various heuristic metrics. Each leaf node of the tree contains at most c rules which have to be linearly processed. [SBVW03] proposes the hypercuts algorithm which extends HiCuts and uses heuristics to merge multiple dimensions in a single node. [Woo00] presents a heuristic, traffic adaptive tree search data structure which reflects the relative rule hit rate to attune to given traffic patterns.

Hardware algorithms: This class of algorithms is implemented in dedicated hardware devices attacking PCP by massive parallelism. Ternary content addressable memory (TCAM) for example matches n rules in parallel. For this purpose, each rule is stored as value / mask pair concatenating the prefix matches of each dimension. The classification result is stored in a n bit vector where each bit, corresponding to a single rule, is 1 if the rule applies. A priority encoder is then used to index the rule action in memory. [LS98] proposes a bitmap intersection scheme which treats each dimension independently by constructing d RLP instances, one for each dimension. On classification, d parallel lookups are issued, each yielding a n bit vector stating the matching rules for the corresponding dimension. The vectors are then intersected by applying bit-wise AND operations and the resulting bit vector is further processed like the TCAM bit vector. This algorithm is extended by [BV01] which describes the aggregated bit vector (ABV) algorithm. ABV assumes sparse vectors, i.e. vectors with a very small number of 1's, to reduce both memory usage and number of parallel operations.

In recent years, a scheme called stateful packet filtering or connection tracking has been proposed to disburden PCP algorithms in the context of firewalling. The approach manages a table of active "connections" where the *connection* term is extended to sup-

port connectionless protocols like UDP or ICMP. For each packet, the table is firstly queried to determine whether the packet belongs to an active connection. In that case, the packet is allowed to pass the firewall. If the packet initiates a new connection, it is processed by a PCP algorithm and a new (half-open) connection is added to the table if the packet is allowed to pass the firewall. Table lookups and updates can be achieved in $O(1)$ using efficient hashing techniques. Additionally, a kind of garbage collection must be implemented which removes stale, timed out connections. Obviously, stateful packet filtering does not supersede packet classification. Yet, it is sometimes used to legitimate naive PCP algorithms since the packet rate relevant for the packet filter is significantly reduced. This reasoning is shortsighted because it does not take into account (distributed) denial of service ((D)DOS) attacks which may lead to high rates of packets not belonging to active connections. The main drawback of connection tracking is that the memory usage depends on the number of parallel connections which can be easily several hundreds of thousands even for edge networks.

1.3 Outline

Chapter 2 formally defines the multi-dimensional packet classification problem using range matches. It describes the reduction of the one-dimensional PCP to the simpler range location problem (RLP) exploiting some basic geometric insights which are also used in [LS98] and [FM00]. The multi-dimensional PCP is then reduced to a generalization of RLP called multi range location problem (MRLP) which is the basis of the novel classification algorithm. Finally, the insert, delete and classification procedures are stated along with their space and time complexities.

Chapter 3 covers design and implementation of HiPAC and introduces the non-terminal packet classification problem (NPCP). It outlines the linux kernel basics relevant to understand the integration of HiPAC as linux kernel module. For this purpose, the linux 2.4 forwarding path with focus on the netfilter and iptables framework [RBM+99] is described. The information on this is primarily extracted from the linux kernel sources ([Org97]) with the aid of lxr ([GG01]), an excellent html based cross-referencing tool. Other useful resources are [BBD+01], [WPR+02] and [kt00].

Chapter 4 presents design and implementation of the verification framework which has been used to test the implementation and to ensure both correctness and robustness. The framework does not involve strictly formal verification methods but instead uses a practical approach which relies on randomized test case generators and a well-behaved execution environment provided by valgrind ([Sew02]).

Chapter 5 presents a preliminary performance evaluation of HiPAC. The performance test compares the time complexity worst case of HiPAC and iptables as a representative of naive linear packet filters.

Chapter 6 brings up a number of topics regarding future optimizations and investigations around HiPAC and PCP in general.

2 Algorithmic foundations

This chapter provides a formal foundation of the multi-dimensional packet classification problem (PCP) using range matches and a novel, dynamic packet filter algorithm. The algorithm relies on the reduction of PCP to another problem embodying the geometric view on packet classification. The reduction is first established for one-dimensional PCP and later generalized towards multiple dimensions.

2.1 Notations

Notation 2.1 *Let $\mathbb{N}^+ = \mathbb{N}\backslash\{0\}$ be the set of positive natural numbers.*

Notation 2.2 *Let $[l,r],\ l,r \in \mathbb{N},\ l \leq r$ denote the set $\{i \in \mathbb{N} \mid l \leq i \leq r\}$.*

Notation 2.3 *Let M be an arbitrary set and $K = (k_1,\ldots,k_n) \in M^n,\ n \in \mathbb{N}^+$. The "subtuple" $(k_i,\ldots,k_j) \in M^{j-i+1},\ 1 \leq i \leq j \leq n$ is denoted by $K^{[i;j]}$. For $1 \leq j < i \leq n$, $K^{[i;j]}$ is defined as the empty tuple $()$. For $n = 0$, M^n is defined as $\{()\}$.*
Note that there is no notational difference between the elements of M and M^1, i.e. for $m \in M^1$ the expressions $m(1)$ and m may be used interchangeably.

Notation 2.4 *Let $S = \langle s_{k-1}\ldots s_0\rangle,\ k \in \mathbb{N}^+,\ s_i \in \{0,1\}\ \forall 0 \leq i < k$ denote a k-bit string representing the value $n = \sum_{i=0}^{k-1} s_i 2^i$. Note that S refers to n rather than to its representation as a sequence of 0's and 1's which is stated as $\langle S\rangle$. The bit width of S is $|\langle S\rangle| = k$. The substring $\langle s_j\ldots s_i\rangle,\ 0 \leq i \leq j < k$ of S is denoted by $\langle S\rangle^{[i;j]}$. By $\overline{S} = \langle\overline{s}_{k-1}\ldots\overline{s}_0\rangle$ the inverted bit string of S is denoted.*

2.2 Packet classification problem (PCP)

In this section, the packet classification problem (PCP) is defined precisely. Intuitively, packet classification means the process of selecting an appropriate rule out of a given set of rules – also called *rule set* – for a certain packet. Each rule consists of a match part that specifies the packets to which the rule applies, an action that is executed if the rule is selected by PCP and a unique priority that is used to choose the *best* rule if more than one applies. A packet is a sequence of fixed width bit strings, also called bit fields. These bit fields don't necessarily have to be of the same width. In connexion with network packets, the bit fields are usually certain parts of the network packet headers, e.g. source IP, destination IP or protocol field of the IPv4 header.

Definition 2.1 *Let $\mathcal{U}_k = [0,\ 2^k - 1],\ k \in \mathbb{N}^+$ be the **universe**. The subscript k is omitted if the size of the universe does not need to be specified.*
*A **packet** P is a d-tuple $(S_1, \ldots, S_d) \in \mathcal{U}^d$ of bit strings. $P(i) = S_i,\ 1 \leq i \leq d$ is called the i-th **packet field** of P or simply the i-th field of P.*

The match part of a rule is specified by a tuple of ranges representing a d-dimensional hyper-rectangle. Each range corresponds to a certain packet field.

Definition 2.2 *Let $d \in \mathbb{N}^+$. A **rule** R is the triple*

$$(p, M, a),\ p, a \in \mathbb{N},\ M \in \{[l, r] \mid 0 \leq l \leq r \leq max(\mathcal{U})\}^d$$

*The **priority** of R is $prio(R) = p$, the **match part** of R is $mat(R) = M$ and the **action** associated with R is $act(R) = a$.*
*$M(i),\ 1 \leq i \leq d$ is called the i-th **match** of R which consists of totalling d matches.*

A rule applies to a packet if the point represented by the packet is contained in the hyper-rectangle represented by the rule, i.e. if the value of each packet field is contained in the corresponding range match. More formally:

Definition 2.3 *Let R be a rule with $d \in \mathbb{N}^+$ matches and P be a packet with d fields. Then R is said to apply to P and vice versa if and only if $P(i) \in (mat(R))(i)\ \forall 1 \leq i \leq d$. This is denoted by $apply(R, P)$. Alternatively, R is said to match P and vice versa.*

The match definition based on ranges is not compulsive. Common alternatives are prefix or mask matches. A general definition would be based on arbitrary subsets of $\mathcal{U}$. Table 2.1 reflects the worst case when converting those match representations into each other. The conversion of a single rule with $d \geq 1$ matches of type X requires at most $\mathcal{C}(X, Y)^d$ rules with matches of type Y where $\mathcal{C}(X, Y)$ is the maximum number of matches of type Y which are required to express a single match of type X. The table entry in the row with label X and the column with label Y states $\mathcal{C}(X, Y)$.

Apart from the worst case number of mask matches required to express a set match, the other table entries don't require a noteworthy reasoning. Below, the worst case set $M_k \subseteq \mathcal{U}_k$ requiring 2^{k-1} mask matches is inductively constructed.

- $M_1 = \{\langle 0 \rangle\}$ (alternative: $M_1 = \{\langle 1 \rangle\}$)
- $M_{2 \leq i \leq k} = \{\langle 0 \langle S \rangle \rangle, \langle 1 \langle S' \rangle \rangle \mid S \in M_{i-1},\ S' \in \mathcal{U}_{i-1} \backslash M_{i-1}\}$
  (alternative: $M_{2 \leq i \leq k} = \{\langle 0 \langle S' \rangle \rangle, \langle 1 \langle S \rangle \rangle \mid S \in M_{i-1},\ S' \in \mathcal{U}_{i-1} \backslash M_{i-1}\}$)

Since $|M_k| = 2^{k-1}$, it must be shown that 2^{k-1} mask matches of the form $n / max(\mathcal{U}_k)$, $n \in M_k$ are required to express M_k. This is done by proving the following condition. Note that the existence of n'' is not relevant for the claim but simplifies the proof.

Claim 2.1 $\forall n \in \mathcal{U}_k\ \forall m \in \mathcal{U}_k \backslash \{max(\mathcal{U}_k)\}\ \exists n', n'' \in n / m :\ n' \notin M_k \wedge n'' \in M_k$

Proof 2.1 *The mask match n / m represents the set $\{n' \in \mathcal{U}_k \mid \forall 0 \leq i < k : \langle m \rangle_i = 1 \Rightarrow \langle n' \rangle_i = \langle n \rangle_i\}$. The claim is shown by induction over k using the main definition of M_k. The proof for the alternative definitions is analog.*

	prefix	range	mask	set
prefix	1	1	1	1
range	$2(k-1)$	1	$2(k-1)$	1
mask	2^{k-1}	2^{k-1}	1	1
set	2^{k-1}	2^{k-1}	2^{k-1}	1

Table 2.1: Number of matches required in the worst case when converting from one match representation into another. $\mathcal{U}_k$ is assumed to be the universe. The rows indicate the source of the conversion and the columns specify the destination, e.g. to express a range match by a number of mask matches at most $2(k-1)$ mask matches are required.

$k = 1: \quad m = 0 \Rightarrow n/m = \{0,1\} \Rightarrow n' = 1,\ n'' = 0$

$k-1 \rightarrow k:$

Induction hypothesis: $\forall \hat{n} \in \mathcal{U}_{k-1}\ \forall \hat{m} \in \mathcal{U}_{k-1} \backslash \{max(\mathcal{U}_{k-1})\}\ \exists \hat{n}', \hat{n}'' \in \hat{n}/\hat{m}:$
$\hat{n}' \notin M_{k-1} \wedge \hat{n}'' \in M_{k-1}$

Let $n \in \mathcal{U}_k,\ m \in \mathcal{U}_k \backslash \{max(\mathcal{U}_k)\}$. *Three cases have to be considered:*

a) $\langle m \rangle_{k-1} = 1,\ \langle n \rangle_{k-1} = 0:$ *Let* $n = \langle 0\langle \hat{n} \rangle\rangle,\ m = \langle 1\langle \hat{m} \rangle\rangle$.
Choose $n' = \langle 0\langle \hat{n}' \rangle\rangle,\ n'' = \langle 0\langle \hat{n}'' \rangle\rangle$.
Then $n' \notin M_k$ *since* $\hat{n}' \notin M_{k-1}$ *and* $n'' \in M_k$ *since* $\hat{n}'' \in M_{k-1}$.

b) $\langle m \rangle_{k-1} = 1,\ \langle n \rangle_{k-1} = 1:$ *Let* $n = \langle 1\langle \hat{n} \rangle\rangle,\ m = \langle 1\langle \hat{m} \rangle\rangle$.
Choose $n' = \langle 1\langle \hat{n}'' \rangle\rangle,\ n'' = \langle 1\langle \hat{n}' \rangle\rangle$.
Then $n' \notin M_k$ *since* $\hat{n}'' \in M_{k-1}$ *and* $n'' \in M_k$ *since* $\hat{n}' \notin M_{k-1}$, *i.e.* $\hat{n}' \in \mathcal{U}_{k-1} \backslash M_{k-1}$.

c) $\langle m \rangle_{k-1} = 0,\ \langle n \rangle_{k-1} \in \{0,1\}:$ *Let* $n^{[0:k-2]} = \hat{n},\ \forall 0 \leq i < k-1:\ \langle m \rangle_i = 1.$ *(*)*
Note that the case $n = \langle 0\langle \hat{n} \rangle\rangle,\ m = \langle 0\langle \hat{m} \rangle\rangle$ *is analog to a) while the case* $n = \langle 1\langle \hat{n} \rangle\rangle,\ m = \langle 0\langle \hat{m} \rangle\rangle$ *is analog to b). As for (*), there are two cases:*
1) $\hat{n} \in M_{k-1}:$ *Choose* $n' = \langle 1\langle \hat{n} \rangle\rangle,\ n'' = \langle 0\langle \hat{n} \rangle\rangle$.
2) $\hat{n} \notin M_{k-1}:$ *Choose* $n' = \langle 0\langle \hat{n} \rangle\rangle,\ n'' = \langle 1\langle \hat{n} \rangle\rangle$.

While range matches are clearly more expressive than prefix matches, the same does not hold for non-contiguous mask matches. At first glance, mask matches seem to be more powerful since it takes only at most $2(k-1)$ mask matches to express a single range match whereas it takes at most 2^{k-1} range matches to express a single mask match. Nevertheless, apart from matching flags, non-contiguous masks are hardly used in practice. One way to reduce the number of range matches required to express a mask match is to split the corresponding packet field into separate chunks and treat each chunk as separate dimension. If a bit field of width k is split into i equally sized chunks, the number of rules required to express a single mask match is reduced to at most 2^{k-i} because each part of the mask match, corresponding to a chunk of width $\frac{k}{i}$, requires at most $2^{\frac{k}{i}-1}$ range matches and since there are i parts, the number of rules is at most $(2^{\frac{k}{i}-1})^i = 2^{k-i}$. On the other hand, matches based on arbitrary subsets of $\mathcal{U}$ are no reasonable basis for an implementation since they can neither be implemented efficiently nor their expressiveness is required in practice.

The definition of a rule set, which is given below, completes the constituents of PCP. A rule set must have the property that for each packet exactly one rule applies. This can be achieved by choosing the priorities of the rules pairwise disjoint and adding a default rule which applies in case no other rule does. This leads to the following definitions.

Definition 2.4 *A **default rule** R_a^{def}, $a \in \mathbb{N}$ is the rule (∞, M, a) where $M(i) = \mathcal{U}\ \forall 1 \leq i \leq d$. It is also called **policy rule**.*

Definition 2.5 *A d-dimensional **rule set** $\mathcal{R}_a$, $a \in \mathbb{N}$ is a set of rules with $d \in \mathbb{N}^+$ matches each and pairwise disjoint priorities:*

$$\mathcal{R}_a = \{R_i,\ 1 \leq i \leq n \mid prio(R_i) \neq prio(R_j)\ \forall 1 \leq i \neq j \leq n\} \cup \{R_a^{def}\}$$

*where a is the default action of the rule set, also called **policy**. The subscript a is omitted if the policy of a rule set does not need to be specified.*

The packet classification problem can be defined as follows.

Definition 2.6 *Given a d-dimensional rule set $\mathcal{R}$ and a packet P with d packet fields, the d-dimensional **packet classification problem (PCP)** returns the smallest priority rule R in $\mathcal{R}$ which applies to P. That is:*

$$PCP(\mathcal{R}, P) = R,\ prio(R) = min\{prio(R'),\ R' \in \mathcal{R} \mid apply(R', P)\}$$

For $d > 1$ the problem is also called multi-dimensional PCP.

2.3 Range location problem (RLP)

The range location problem (RLP) is a very simple problem which turns out to be the basic building block for solving the multi-dimensional PCP by reduction. Given a set of ranges which form a partition of $\mathcal{U}$ and a point in $\mathcal{U}$, RLP returns the range which contains the point. The set of ranges can be represented by a subset of $\mathcal{U}$ which contains the right endpoints of all ranges. This representation is unique since the maximum of each range is unique and it's also well-defined since the minimum of each range is the maximum of the lower adjacent range plus 1 or 0 if no such range exists.

Definition 2.7 *A **range partition** G of $\mathcal{U}$ is a subset of $\mathcal{U}$ with $max(\mathcal{U}) \in G$. It represents the following partition of $\mathcal{U}$: $\{[l, r] \mid r \in G,\ l = max(\{0\} \cup \{r' + 1,\ r' \in G \mid r' < r\})\}$.*

The range location problem can be stated as follows.

Definition 2.8 *Given a range partition G of $\mathcal{U}$ and a packet P with a single field, the **range location problem (RLP)** returns the element of G representing the range in which P lies. That is: $RLP(G, P) = min\{g \in G \mid g \geq P\}$.*

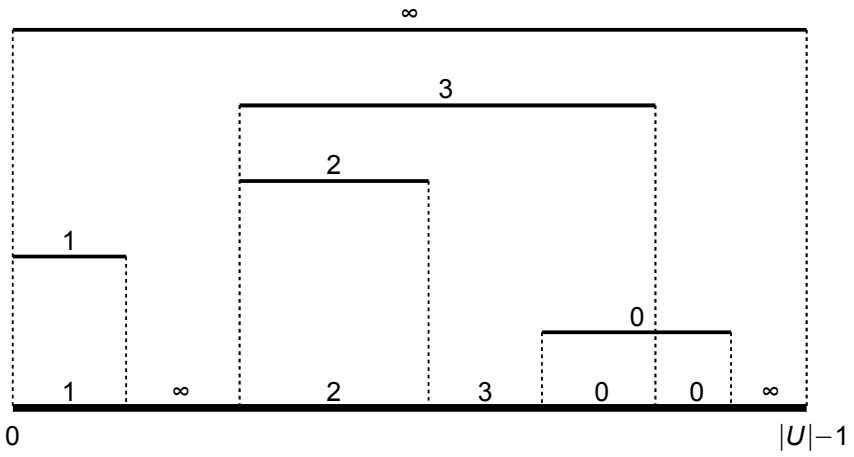

Figure 2.1: Example reduction of an instance of one-dimensional PCP to RLP. Each bar represents the single range match of a rule. The number above each bar is the priority of the rule. The corresponding instance of RLP is created by projecting the endpoints of the ranges onto the universe which results in a range partition. Above each range of the partition, the priority of the smallest priority rule overlapping the range is stated.

2.4 Reduction of one-dimensional PCP to RLP

The reduction of the one-dimensional PCP to RLP is straightforward. Considering the range matches of the rules geometrically, one can create a range partition by projecting the endpoints of all ranges onto the universe. There is exactly one smallest priority rule overlapping each range of the partition. Figure 2.1 illustrates an example which reflects the basic idea behind the reduction. In the following, the reduction is formalized.

Definition 2.9 *Let $M \subseteq \{[l,r] \mid 0 \leq l \leq r \leq max(\mathcal{U})\}$, $\mathcal{U} \in M$ be a set of ranges. The range partition created by projecting the endpoints of the ranges onto $\mathcal{U}$ is given by:*

$$\Phi(M) = \left\{ g \in \mathcal{U} \mid \exists [l,r] \in M : \; g = r \;\vee\; (l > 0 \;\wedge\; g = l-1) \right\}$$

Note that $\Phi(M)$ is indeed a range partition, i.e. $max(\mathcal{U}) \in \Phi(M)$ since $\mathcal{U} \in M$. Therewith, the reduction of the one-dimensional PCP to RLP can be stated as follows.

Claim 2.2 *Let $PCP(\mathcal{R}, P)$ be an arbitrary instance of the one-dimensional packet classification problem. Let $G = \Phi(\{mat(R),\ R \in \mathcal{R}\})$ be the range partition which is created by projecting the endpoints of the matches of all rules in $\mathcal{R}$ onto the universe. Let $\mathcal{A} \in G \to \mathcal{R}$ be a total function such that: $\mathcal{A}(g) = R,\ prio(R) = min\{prio(R'),\ R' \in \mathcal{R} \mid apply(R', g)\}$.*
Then the following condition holds: $PCP(\mathcal{R}, P) = \mathcal{A}(RLP(G, P))$.

Proof 2.2 *Note that G is indeed a range partition since $mat(R_a^{def}) = \mathcal{U},\ R_a^{def} \in \mathcal{R}$. From the construction of $\mathcal{A}$ follows $\forall P \in G : \; PCP(\mathcal{R}, P) = \mathcal{A}(P) = \mathcal{A}(RLP(G, P))$ immediately. It is still left to be shown that the claim also holds for all elements of $\mathcal{U} \backslash G$. Let $P \in \mathcal{U} \backslash G$ and $\hat{P} \in G$ such that $RLP(G, P) = \hat{P}$. Thus $P \leq \hat{P}$ holds. Below, it is shown that the construction*

yields $L_P := \{R \in \mathcal{R} \mid apply(R,P)\} = \{R \in \mathcal{R} \mid apply(R,\hat{P})\} =: L_{\hat{P}}$. *As convenient shortcut,* $max(R)$ *is used to denote* $max(mat(R))$ *while* $min(R)$ *denotes* $min(mat(R))$. *Assume* $L_P \neq L_{\hat{P}}$ *which yields two cases:*

a) $\exists R' \in L_P : R' \notin L_{\hat{P}}$: *Thus* $min(R') \leq P \leq max(R')$. $R' \notin L_{\hat{P}}$ *yields two cases:*
   1) $max(R') < \hat{P}$
      $\Longrightarrow RLP(G,P) \leq max(R')$ *since* $max(R') \in G$ *and* $P \leq max(R')$
      $\Longrightarrow RLP(G,P) < \hat{P}$ *which contradicts* $RLP(G,P) = \hat{P}$
   2) $min(R') > \hat{P}$
      $\Longrightarrow P \geq min(R') > \hat{P}$ *which contradicts* $P \leq \hat{P}$

b) $\exists R' \in L_{\hat{P}} : R' \notin L_P$: *Thus* $min(R') \leq \hat{P} \leq max(R')$. $R' \notin L_P$ *yields two cases:*
   1) $max(R') < P$
      $\Longrightarrow \hat{P} \leq max(R') < P$ *which contradicts* $P \leq \hat{P}$
   2) $min(R') > P$
      $\Longrightarrow RLP(G,P) \leq min(R') - 1$ *since* $\underbrace{min(R')}_{>P \geq 0} - 1 \in G$ *and* $P < min(R')$
      $\Longrightarrow RLP(G,P) < min(R') \leq \hat{P}$ *which contradicts* $RLP(G,P) = \hat{P}$

Thus $L_P = L_{\hat{P}}$ *holds which implies* $\mathcal{A}(P) = \mathcal{A}(\hat{P})$. *Since* $PCP(\mathcal{R},P) = \mathcal{A}(P)$ *and* $RLP(G,P) = \hat{P}$, *the claim* $PCP(\mathcal{R},P) = \mathcal{A}(RLP(G,P))$ *follows.*

2.5 Reduction of multi-dimensional PCP to multi RLP (MRLP)

The reduction of one-dimensional PCP to RLP can be generalized in a way that allows any instance of the multi-dimensional packet classification problem to be expressed as a set of instances of the range location problem. The RLPs are hierarchically arranged to form a tree. The construction of the tree is inductively defined and starts with the root node which is the RLP created by the first dimension matches of all rules contained in the rule set. Each range of the partition is overlapped by a number of matches and thus by the corresponding rules. At least one rule overlaps each range due to the default rule being contained in every rule set. The construction of the tree evolves by creating one child node for each range of the partition. The rules which are considered in a certain child are those whose first dimension match overlaps the corresponding range of the partition. Hence, one can think of a subset of rules propagating into the child RLP. The construction of the child RLP works the same way as for the root RLP but this time, the second dimension matches of all propagated rules are considered. The construction continues until a tree of depth[1] $d-1$ is constructed. Regarding the whole tree, one observes that all nodes in depth $i-1$ consist of RLPs which are created by the i-th dimension matches of the propagated rules. In the leaf level of the tree, each range of the partition of a RLP can be associated with the smallest priority rule overlapping the range. The lookup in the tree is performed by solving one range location problem

[1] A tree solely consisting of the root node has depth 0.

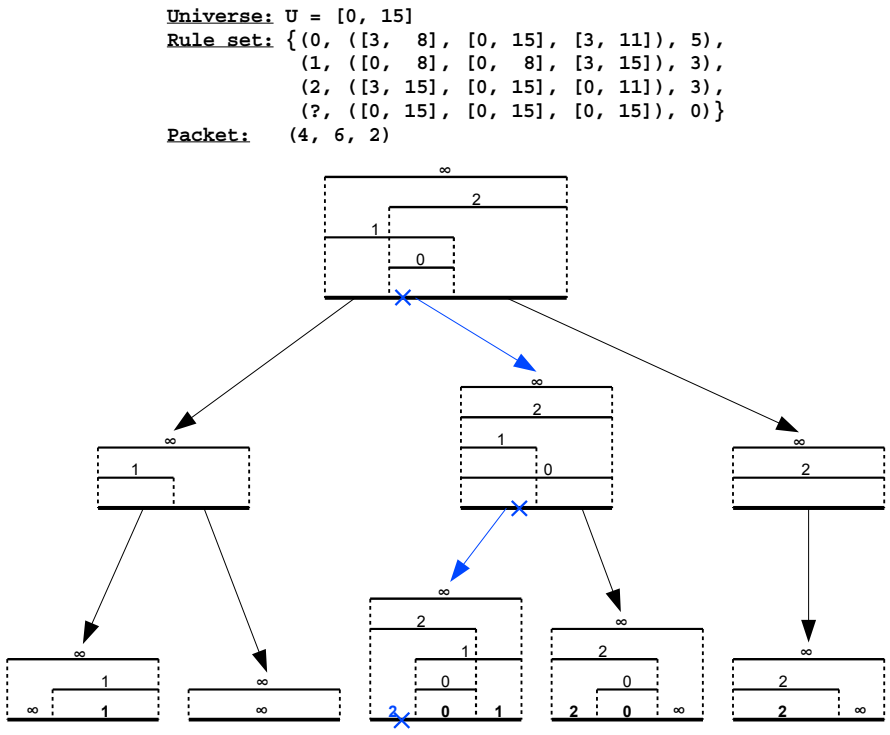

Figure 2.2: Example reduction of an instance of three-dimensional PCP to MRLP. The RLPs are illustrated as nodes in the tree. Each bar in a node is uniquely identified by the number above the bar and the depth of the node in the tree (starting from 0). The bar with number p in depth $i-1$ represents the i-th dimension match of the rule with priority p. Each leaf RLP contains an additional number above each range of the range partition. This number is the priority of smallest priority rule overlapping the range. The blue arrows demonstrate the lookup path for the given packet. Each field of the packet is represented by a blue cross in the range partition of the corresponding RLP.

for each packet field starting with the root RLP and the first packet field. The solution yields a range in the partition and the lookup continues with the child RLP associated with that range. In figure 2.2 both the reduction and the lookup is shown for a three-dimensional rule set and a sample packet.

One way to formalize the tree of RLPs is to construct d functions, one for each level of the tree. For $0 \leq i < d$, the i-th function maps a path of length i to the range partition associated with the node in level i of the tree to which the path leads. The path is a sequence of ranges specified by their maximum values. This leads to the following definition of the multi range location problem (MRLP) along with the reduction of multi-dimensional PCP to MRLP which serves as formal foundation of the novel PCP algorithm.

Definition 2.10 *Let $\mathcal{G} = \{\mathcal{U}' \subseteq \mathcal{U} \mid max(\mathcal{U}) \in \mathcal{U}'\}$ be the set of all range partitions. Let P be a packet with d fields and $\mathcal{M} = (M_0, \ldots, M_{d-1})$ be a d-tuple of total functions mapping a path of length i to the range partition of the corresponding RLP. The function domains are inductively defined as follows:*

$$M_0 \in \{()\} \rightarrow \mathcal{G}$$
$$M_{1 \leq i < d} \in \{K \in \mathcal{U}^i \mid K^{[1:i-1]} \in dom(M_{i-1}) \wedge K(i) \in M_{i-1}(K^{[1:i-1]})\} \rightarrow \mathcal{G}$$

*The **multi range location problem (MRLP)** solves d range location problems, one for each packet field. The range partition subject to the RLP determining the range which contains the i-th packet field depends on the $i-1$ previously determined ranges. More formally:*

$$MRLP(\mathcal{M}, P) = K \in U^d,\ \forall 1 \leq i \leq d:\ K(i) = RLP(M_{i-1}(K^{[1:i-1]}), P(i))$$

The formal reduction of multi-dimensional PCP to MRLP is stated in the following claim.

Claim 2.3 *Let $PCP(\mathcal{R}, P)$ be an arbitrary instance of the d-dimensional packet classification problem. Let $\mathcal{O} = (O_0, \ldots, O_d)$ be a $(d+1)$-tuple of total functions $O_{0 \leq i \leq d} : \mathcal{U}^i \rightarrow \{\mathcal{R}' \subseteq \mathcal{R}\}$ mapping the first i packet fields to the set of rules matching these fields. The functions are inductively defined as follows:*

$$O_0() = \mathcal{R}$$
$$O_{1 \leq i \leq d}(P) = \left\{R \in O_{i-1}(P^{[1:i-1]}) \mid P(i) \in (mat(R))(i)\right\}$$

Let $\mathcal{M} = (M_0, \ldots, M_{d-1})$ be a d-tuple of total functions subject to definition 2.10 which are defined as follows:

$$M_{0 \leq i < d}(K) = \Phi(\{(mat(R))(i+1),\ R \in O_i(K)\})$$

Let $\mathcal{A}$ be a total function which maps a path of length d to the smallest priority rule matching the path.

$$\mathcal{A} : \{K \in \mathcal{U}^d \mid K^{[1:d-1]} \in dom(M_{d-1}) \wedge K(d) \in M_{d-1}(K^{[1:d-1]})\} \rightarrow \mathcal{R},$$
$$\mathcal{A}(K) = R,\ prio(R) = min\{prio(R'),\ R' \in O_d(K)\}$$

The construction yields: $PCP(\mathcal{R}, P) = \mathcal{A}(MRLP(\mathcal{M}, P))$

Proof 2.3 *$\mathcal{O}$ is inductively defined to be suitable for an induction based correctness proof. One can easily ascertain that $\mathcal{O}$ can also be written as a tuple of non-inductive functions $O_{0 \leq i \leq d}(P) = \{R \in \mathcal{R} \mid \forall 1 \leq j \leq i:\ P(j) \in (mat(R))(j)\}$. Thus $O_d(P) = \{R \in \mathcal{R} \mid apply(R, P)\}$ holds which means that $PCP(\mathcal{R}, P) = \mathcal{A}(P)$ for $P \in dom(\mathcal{A})$.*

For P being an arbitrary packet with d fields, let $K \in U^d$, $K = MRLP(\mathcal{M}, P)$ be the solution of the corresponding MRLP, i.e. $\forall 1 \leq i \leq d:\ K(i) = RLP(M_{i-1}(K^{[1:i-1]}), P(i))$. As convenient shortcut, $R(i)$ is used to denote $(mat(R))(i)$. Below, it is shown by induction over d that $O_d(P) = O_d(K)$ holds.

$d = 1:\ O_1(P(1)) = O_1(K(1))$ *must be shown.*

$$O_1(P(1)) = \{R \in \mathcal{R} \mid P(1) \in R(1)\} \quad \textit{and}$$
$$O_1(K(1)) = \{R \in \mathcal{R} \mid K(1) \in R(1)\}$$
$$= \{R \in \mathcal{R} \mid RLP(M_0(), P(1)) \in R(1)\}$$
$$= \{R \in \mathcal{R} \mid RLP(\Phi(\{(R'(1),\ R' \in \mathcal{R}\}), P(1)) \in R(1)\}$$

Thus, $O_1(P(1)) = O_1(K(1))$ is already shown in proof 2.2 by $L_P = L_{\hat{P}}$.

$d-1 \to d$: *Induction hypothesis (IH):* $O_{d-1}(P^{[1:d-1]}) = O_{d-1}(K^{[1:d-1]})$
Assume $O_d(P) \neq O_d(K)$ *which yields two cases:*

a) $\exists R' \in O_d(P) : R' \notin O_d(K)$
$\Longrightarrow \forall 0 \leq i < d : \ R' \in O_i(P^{[1:i]}) \ \wedge \ R' \in O_i(K^{[1:i]})$ *(IH)*
$\Longrightarrow max(R'(d)) \in M_{d-1}(K^{[1:d-1]})$ *since* $R' \in O_{d-1}(K^{[1:d-1]})$ *and*
$min(R'(d)) \leq P(d) \leq max(R'(d))$ *since* $R' \in O_d(P)$
$R' \notin O_d(K)$ *implies* $K(d) \notin R'(d)$ *since* $R' \in O_{d-1}(K^{[1:d-1]})$. *This yields two cases:*
1) $max(R'(d)) < K(d)$
$\Longrightarrow RLP(M_{d-1}(K^{[1:d-1]}), P(d)) \leq max(R'(d))$ *since* $max(R'(d)) \in M_{d-1}(K^{[1:d-1]})$
and $P(d) \leq max(R'(d))$
$\Longrightarrow RLP(M_{d-1}(K^{[1:d-1]}), P(d)) < K(d)$ *which contradicts*
$RLP(M_{d-1}(K^{[1:d-1]}), P(d)) = K(d)$
2) $min(R'(d)) > K(d)$
$\Longrightarrow P(d) \geq min(R'(d)) > K(d)$ *which contradicts*
$P(d) \leq RLP(M_{d-1}(K^{[1:d-1]}), P(d)) = K(d)$

b) $\exists R' \in O_d(K) : R' \notin O_d(P)$
$\Longrightarrow \forall 0 \leq i < d : \ R' \in O_i(K^{[1:i]}) \ \wedge \ R' \in O_i(P^{[1:i]})$ *(IH)*
$\Longrightarrow min(R'(d)) - 1 \in M_{d-1}(K^{[1:d-1]})$ *(if* $min(R'(d)) > 0$*) since*
$R' \in O_{d-1}(K^{[1:d-1]})$ *(IH) and*
$min(R'(d)) \leq K(d) \leq max(R'(d))$ *since* $R' \in O_d(K)$
$R' \notin O_d(P)$ *implies* $P(d) \notin R'(d)$ *since* $R' \in O_{d-1}(P^{[1:d-1]})$. *This yields two cases:*
1) $max(R'(d)) < P(d)$
$\Longrightarrow K(d) \leq max(R'(d)) < P(d)$ *which contradicts*
$P(d) \leq RLP(M_{d-1}(K^{[1:d-1]}), P(d)) = K(d)$
2) $min(R'(d)) > P(d)$
$\Longrightarrow RLP(M_{d-1}(K^{[1:d-1]}), P(d)) \leq min(R'(d)) - 1$ *since*
$\underbrace{min(R'(d))}_{>P(d)\geq 0} - 1 \in M_{d-1}(K^{[1:d-1]})$ *and* $P(d) < min(R'(d))$
$\Longrightarrow RLP(M_{d-1}(K^{[1:d-1]}), P(d)) < min(R'(d)) \leq K(d)$ *which contradicts*
$RLP(M_{d-1}(K^{[1:d-1]}), P(d)) = K(d)$

Thus $O_d(P) = O_d(K)$ *holds which implies* $\mathcal{A}(P) = \mathcal{A}(K)$. *Since* $PCP(\mathcal{R}, P) = \mathcal{A}(P)$ *and* $MRLP(\mathcal{M}, P) = K$, *the claim* $PCP(\mathcal{R}, P) = \mathcal{A}(MRLP(\mathcal{M}, P))$ *follows.*

2.6 PCP algorithm

The PCP algorithm requires a RLP solving data structure. Since there are several implementations possible, the data structure is referred to by the following abstract interface.

Definition 2.11 *Let* ***rlp*** *be a RLP solving data structure which represents the range partition* G *where each element of* G *is associated with a counter and an object represented as an element of* $\mathbb{N}$. *Thus,* ***rlp*** *represents the set* $L = \{(g, o, c), (g', o', c') \in G \times \mathbb{N} \times \mathbb{N} \mid (g, o, c) \neq$

$(g', o', c') \Rightarrow g \neq g'\}$. *Let* $n = |L| = |G|$. *The space complexity of* ***rlp*** *is* $f_{rlp\_space}(n)$ *where* $f_{rlp\_space} : \mathbb{N} \rightarrow \mathbb{R}^+$ *is monotonically increasing. The following operations must be supported:*

operation	***semantics***	***precondition***	***time complexity***
RLP.new(o)	$L := \{(max(\mathcal{U}), o, 1)\}$ *return* L	–	$O(1)$
rlp.free()	*destroy* L	–	$O(1)$
rlp.clone()	$L' :=$ *copy*(L) *return* L'	–	$O(f_{rlp\_space}(n))$
rlp.insert(g, o)	$L := L \cup \{(g, o, 1)\}$	$g \in \mathcal{U} \wedge \nexists (g, o', c') \in L$	$f_{rlp\_insert}(n)$
rlp.delete(g)	$L := L \backslash \{(g, o, c)\}$	$\exists (g, o, c) \in L$	$f_{rlp\_delete}(n)$
rlp.locate(k)	$g := RLP(G, k)$ *let* $(g, o, c) \in L$ $loc.key := g$ $loc.next := o$ $loc.count := c$ *return* loc	$k \in \mathcal{U}$	$f_{rlp\_locate}(n)$
rlp.get_next(g)	*if* $g = max(\mathcal{U})$ *return nil* $g' := RLP(G, g + 1)$ *let* $(g', o, c) \in L$ $loc.key := g'$ $loc.next := o$ $loc.count := c$ *return* loc	$g \in G$	$f_{rlp\_get\_next}(n)$

Note that the counter c must be exposed to the outside via loc, i.e. if loc.count is modified then c is also modified. The same holds for the object o. This behavior must only be guaranteed until the next rlp operation is executed.

The rlps, representing the nodes in the MRLP tree, must be implemented as compact as possible since the tree may grow fairly large. Hence, $f_{rlp\_space}(n) \in O(n)$ is required for the rlp implementation to be practically useful. [Bel04] presents a detailed overview of common RLP solutions, e.g. binary search, and state of the art RLP algorithms. The practicability of the data structures is evaluated by means of their performance on modern cpu architectures.

In the following, the novel PCP algorithm is defined by a set of functions managing a dynamic, d-dimensional MRLP data structure. The data structure is a tree of depth d whose internal nodes are RLP solving data structures, henceforth called rlps. Each rlp contains a child node associated with each range of the partition. A leaf node is the smallest priority rule matching the path leading to the leaf. The presented functions describe only the operations on the tree itself. Additionally, a list of rules must be maintained which is sorted after the rule priorities in ascending order. The list is assumed to contain a policy rule which is referred to by *default_rule*. Most functions are stated recursively involving the function parameters *max_dim* (depth of the tree) and

cur_dim (depth of the current node in the tree). The initial call requires *max_dim* to be set to $d-1$ and *cur_dim* to 0. The following operations are supported:

MRLP.new(*max_dim*): Create a tree of depth max_dim+1 representing the rule set solely consisting of *default_rule*.

MRLP.locate(*rlp*, *max_dim*, *packet*): Solve PCP on the packet *packet* (array of length max_dim+1) and return the corresponding rule. The tree is rooted at *rlp*.

MRLP.insert(*rlp*, *cur_dim*, *max_dim*, *new_rule*): Insert rule *new_rule* into the tree rooted at *rlp*.

MRLP.delete(*rlp*, *cur_dim*, *max_dim*, *del_rule*, *path*): Delete rule *del_rule* from the tree rooted at *rlp*. *path* is an uninitialized array of length max_dim+1.

The functions are intentionally kept straight and simple. [Bel04] presents several optimizations to significantly speedup the performance, especially for real world rule sets.

Function `MRLP.new(`*max_dim*`)`

Precondition: $max\_dim \geq 0$

i ← 0
rlp ← RLP.new(*default_rule*)
while *i* < *max_dim* **do**
    rlp' ← RLP.new(*rlp*)
    rlp ← *rlp'*
    i ← *i* + 1
end
return *rlp*

According to definition 2.11, *rlp*.locate(*k*) returns a data structure *loc* encompassing three components: *loc.key*, *loc.next* and *loc.count*. *loc.key* is the element of *rlp* representing the range which contains *k*. *loc.next* is a reference to the child node associated with *loc.key* which is either another rlp or a rule. *loc.count* counts the number of matches (rules) creating *loc.key*. This information is used by `MRLP.delete` to decide whether the range must be deleted from the partition. In function `MRLP.locate`, a convenient notation is introduced. If *loc.next* references a rlp then *loc.rlp* may be written instead of *loc.next* and if it references a rule then *loc.rule* may be used instead.

Before `MRLP.insert` can be defined, a simple function is stated which is used to clone subtrees. `MRLP.clone`(*p*, *cur_dim*, *max_dim*) recursively copies all rlps contained in the subtree rooted at *p* where *cur_dim* is the depth of *p* with respect to the whole tree. Note that leaves are not copied because they are rules contained in the additional rule list, i.e. the rlps in depth *max_dim* directly reference elements of the rule list.

Function `MRLP.locate(`*rlp, max_dim, packet*`)`

Precondition: $max\_dim \geq 0 \wedge$
packet is an array of length $max\_dim + 1$

```
i ← 0
while i ≤ max_dim do
    l ← rlp.locate(packet[i])
    rlp ← l.rlp
    i ← i + 1
end
return l.rule
```

Function `MRLP.clone(`*p, cur_dim, max_dim*`)`

Precondition: $max\_dim \geq 0 \wedge$
$0 \leq cur\_dim \leq max\_dim + 1 \wedge$
p is a rlp if $cur\_dim \leq max\_dim$ and a rule if $cur\_dim > max\_dim$

```
if cur_dim > max_dim then
    return p
end
rlp ← p
new_rlp ← rlp.clone()
l ← new_rlp.locate(0)
do
    l.next ← MRLP.clone(l.next, cur_dim + 1, max_dim)
    l ← new_rlp.get_next(l)
while l ≠ nil
return new_rlp
```

During each recursive step, `MRLP.insert`($rlp, cur\_dim, max\_dim, new\_rule$) inserts the *cur_dim*-th match $[l,r]$ of *new_rule* into *rlp*. The algorithm consists of two parts. In the first part, at most two keys are added to *rlp* if they are not already contained, namely $l-1$ if $l > 0$ and r (cf. definition 2.9). If a key is already contained, the associated counter *loc.count* is incremented to indicate that the corresponding range is also created by *new_rule*. Inserting a new key semantically translates to splitting an already existing range in two adjacent ranges which requires the subtree associated with the original range to be recursively cloned. Note that the range $[l',r']$ is split twice if $l' < l \leq r < r'$. In the second part, *new_rule* is inserted into the child nodes of *rlp* associated with the ranges overlapped by $[l,r]$. If the child nodes are rules, i.e. $cur\_dim = max\_dim$, it is checked whether *new_rule* has a smaller priority than the current rule. In this case, *new_rule* becomes the smallest priority rule matching the current path. For rlp child nodes, a recursive call is issued to insert the remaining matches into the subtree rooted by the child rlp. Figure 2.3 illustrates a step-by-step run of `MRLP.insert` on a three-dimensional example rule set containing 3 rules.

Before `MRLP.delete` can be defined, two auxiliary functions are stated. The first one, `MRLP.deltree`(p, *cur_dim*, *max_dim*), is the counterpart of `MRLP.clone` since it recursively deletes the subtree rooted at p given that *cur_dim* is the depth of p with respect to the whole tree. The function may also be used to delete the entire tree. The other function, `MRLP.get_rule`(*path*, *len*, *prio*), returns the smallest priority rule R, $prio(R) > prio$ matching *path* which is a *len*-dimensional packet. This function is used to find the "second-best" rule for a range referencing the rule being deleted.

During each recursive step, `MRLP.delete`($rlp, cur\_dim, max\_dim, del\_rule, path$) deletes the *cur_dim*-th match $[l,r]$ of *del_rule* from *rlp*. Like `MRLP.insert`, the algorithm consists of two parts. In the first part, the keys which have been created during the insertion of *del_rule*, namely $l-1$ (if $l > 0$) and r, are deleted from *rlp* if necessary. The range corresponding to each key may only be deleted if there is no other rule besides *del_rule* which creates it. This condition holds if the associated counter *loc.count* is 0 after being decremented. Note that the counter of $l-1$ and r may be different which means that either key may be deleted while the other one stays in place. Also note that if $r = max(\mathcal{U})$ then r is not deleted since it is not allowed to remove *default_rule*. Deleting a key k from *rlp* means that the corresponding range is merged together with the right adjacent range represented by the key k'. This is only feasible if the set of rules propagated into the subtree associated with k, called k-subtree, excluding *del_rule* equals the set of rules propagated into the subtree associated with k', called k'-subtree. Assuming *loc.count* of k is 0, one can easily ascertain that this condition holds. If it wouldn't be true, there are two cases possible. Either a rule R is propagated into k-subtree which is not propagated into k'-subtree or vice versa. In the first case, $min(mat(R)) \leq k < max(mat(R)) < k'$ holds which contradicts the fact that k' is the right adjacent range of k. In the second case, $k < min(mat(R)) - 1 < k' \leq max(mat(R))$ holds which leads to the same contradiction. For $k = l-1$, it may happen that $k' > r$ if r has been removed from *rlp*. In this case, k-subtree must be deleted if *loc.count* of k

Function `MRLP.insert` (*rlp, cur_dim, max_dim, new_rule*)

Precondition: $0 \leq cur\_dim \leq max\_dim \wedge$
new_rule.match[*cur_dim*] defined

```
/* create new ranges in rlp if necessary */
m ← new_rule.match[cur_dim]
if m.left > 0 then
    l ← rlp.locate(m.left - 1)
    if l.key ≥ m.left then
        rlp.insert(m.left - 1, MRLP.clone(l.next, cur_dim + 1, max_dim))
    else
        l.count ← l.count + 1
    end
end
l ← rlp.locate(m.right)
if l.key > m.right then
    rlp.insert(m.right, MRLP.clone(l.next, cur_dim + 1, max_dim))
else
    l.count ← l.count + 1
end

/* iterate over overlapped ranges */
l ← rlp.locate(m.left)
if cur_dim = max_dim then
    /* terminal case */
    do
        if l.rule.prio > new_rule.prio then
            l.rule ← new_rule
        end
        l ← rlp.get_next(l)
    while l ≠ nil ∧ l.key ≤ m.right
else
    /* nonterminal case */
    do
        MRLP.insert(l.rlp, cur_dim + 1, max_dim, new_rule)
        l ← rlp.get_next(l)
    while l ≠ nil ∧ l.key ≤ m.right
end
```

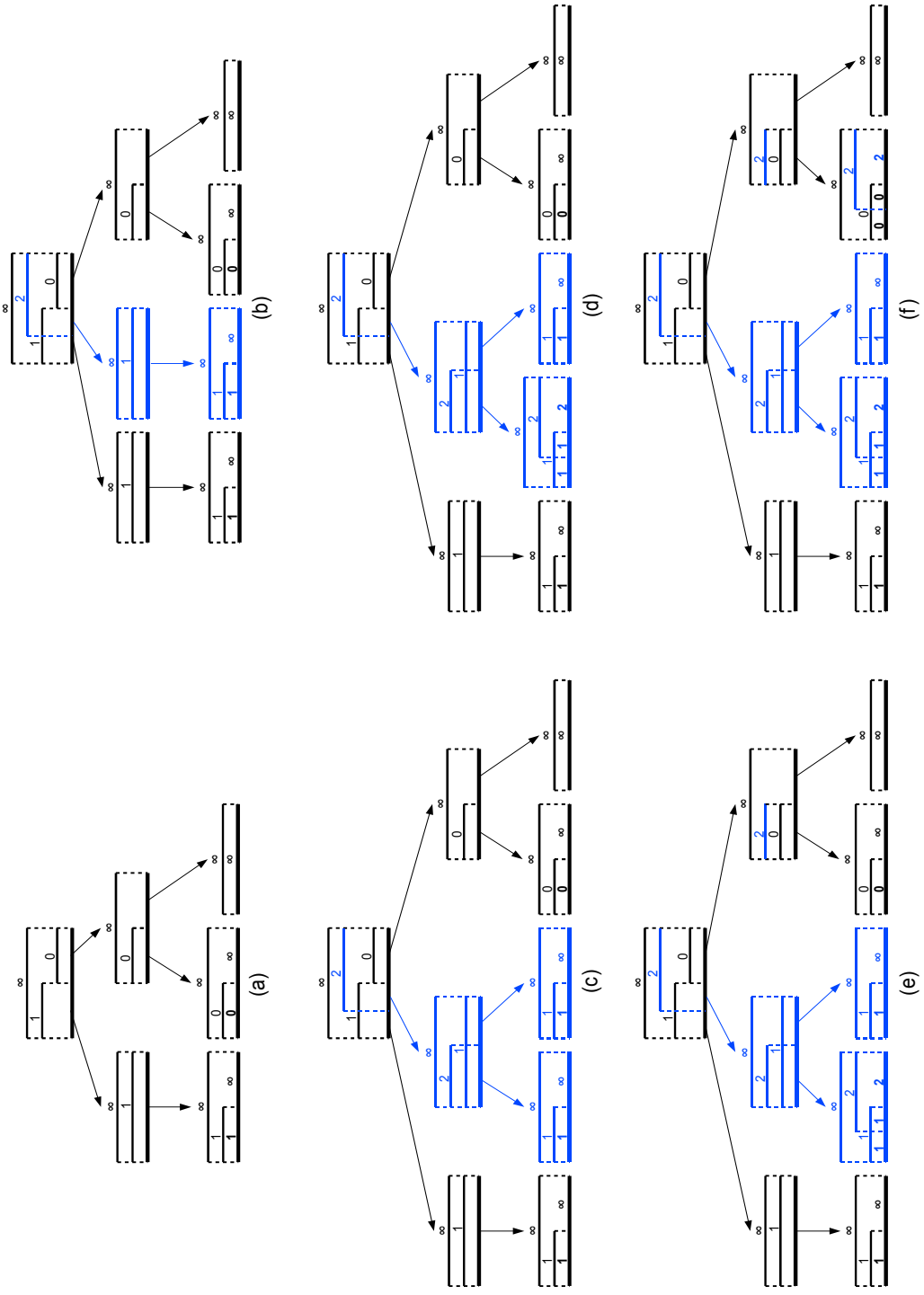

Figure 2.3: Step-by-step run of `MRLP.insert` on a three-dimensional example rule set containing 3 rules. The number above each bar of a rlp is the priority of the rule to which the match belongs. The numbers above the ranges of the rlp range partitions in depth 2 state the priority of the smallest priority rule matching the corresponding path.

Function `MRLP.deltree(`*p, cur_dim, max_dim*`)`

Precondition: $max\_dim \geq 0 \wedge$
$0 \leq cur\_dim \leq max\_dim + 1 \wedge$
p is a rlp if $cur\_dim \leq max\_dim$ and a rule if $cur\_dim > max\_dim$

```
if cur_dim > max_dim then
    return
end
rlp ← p
l ← rlp.locate(0)
do
    MRLP.deltree(l.next, cur_dim + 1, max_dim)
    l ← rlp.get_next(l)
while l ≠ nil
rlp.free()
```

Function `MRLP.get_rule(`*path, len, prio*`)`

Precondition: $len \geq 0 \wedge$
path is an array of length *len* $\wedge$
rule set is sorted by rule priorities in ascending order

```
forall rules rule in rule set do
    if rule.prio > prio then
        i ← 0
        m ← rule.match
        while i < len ∧ m[i].left ≤ path[i] ≤ m[i].right do
            i ← i + 1
        end
        if i = len then
            return rule
        end
    end
end
/* Unreachable */
```

is 0. In the second part of the algorithm, the remaining ranges overlapped by $[l,r]$ are iterated. If the child nodes are rlps, the remaining matches of *del_rule* are deleted recursively from the child associated with each range. If the child nodes are rules, each child equal to *del_rule* must be replaced by the "second-best" rule, i.e. the second smallest priority rule matching the path leading to the child. Figure 2.4 illustrates a step-by-step run of `MRLP.delete` on a three-dimensional example rule set containing 4 rules.

2.7 Time and space complexity of the PCP algorithm

The worst case space complexity of the MRLP tree depends on the number of rules n and the number of matches per rule d. Consider the following d-dimensional rule set containing n rules including the policy rule.

$$\mathcal{R}_w = R^{def} \cup \{(i, M_i, 0),\ 1 \leq i < n \mid \forall 1 \leq j \leq d :\ M_i(j) = [i,\ max(\mathcal{U}) + i - n]\}$$

Figure 2.5 illustrates the root node of the tree representing $\mathcal{R}_w$. Note that each match creates two ranges in the range partition apart from the match which belongs to the policy rule. This one creates only a single range. Thus, the maximum number of ranges is created, namely $2n-1$. The matches are arranged in a way that they are maximally overlapping. Implicitly, we assume that $2(n-1) < max(\mathcal{U})$ holds which ensures that the left endpoints are smaller than the right endpoints. The worst case space complexity is described by the following recurrence:

$$\begin{aligned} T_{space}(n,1) &= f_{rlp\_space}(2n-1) \\ T_{space}(n,d) &= f_{rlp\_space}(2n-1) + T_{space}(n,d-1) + 2 \cdot \sum_{i=1}^{n-1} T_{space}(i,d-1) \end{aligned}$$

Note that d is a constant in the recurrence whereas n is not. This coincides perfectly with the nature of the data structure which offers functions to dynamically increase (or decrease) n while d is fixed after instantiating a new data structure. The recurrence yields:

$$\begin{aligned} T_{space}(n,d) &\leq f_{rlp\_space}(2n-1) + (2n-1) \cdot T_{space}(n,d-1) \\ &= f_{rlp\_space}(2n-1) \cdot \sum_{i=0}^{d-1} (2n-1)^i \end{aligned}$$

Hence, $T_{space}(n,d) \in O(n^{d-1} \cdot f_{rlp\_space}(2n-1))$ and assuming $f_{rlp\_space}(n) \in O(n)$ yields $T_{space}(n,d) \in O(n^d)$.

The running time of `MRLP.locate` $T_{locate}(n,d)$ is $O(d \cdot f_{rlp\_locate}(2n-1))$. Using btrees as RLP solving data structure yields $T(n,d) \in O(d \cdot log_m(n))$. The best known data structure solving the unbounded range location problem by Beame and Fich (cf. [Bel04]) achieves even $T(n,d) \in O\left(d \cdot \sqrt{\frac{log(n)}{log(log(n))}}\right)$. Stratified trees aka van Emde Boas trees are specialized for the bounded range location problem which corresponds to the

Function `MRLP.delete(`*rlp, cur_dim, max_dim, del_rule, path*`)`

```
Precondition: 0 ≤ cur_dim ≤ max_dim ∧
              del_rule ≠ default_rule ∧
              del_rule.match[cur_dim] defined ∧
              path is an array of length max_dim + 1
/* delete ranges in rlp if necessary */
m ← del_rule.match[cur_dim]
leftmost ← m.left
l ← rlp.locate(m.right)
l.count ← l.count - 1
if l.count = 0 then
    MRLP.deltree(l.next, cur_dim + 1, max_dim)
    rlp.delete(m.right)
end
if m.left > 0 then
    l ← rlp.locate(m.left - 1)
    l.count ← l.count - 1
    if l.count = 0 then
        l' ← rlp.get_next(l)
        if l'.key ≤ m.right then
            MRLP.deltree(l'.next, cur_dim + 1, max_dim)
            l'.next ← l.next
            leftmost ← l'.key + 1
        else
            MRLP.deltree(l.next, cur_dim + 1, max_dim)
            leftmost ← m.right + 1
        end
        rlp.delete(m.left - 1)
        if leftmost > m.right then return
    end
end
/* iterate over overlapped ranges */
l ← rlp.locate(leftmost)
if cur_dim = max_dim then
    /* terminal case */
    while l ≠ nil ∧ l.key ≤ m.right do
        if l.rule = del_rule then
            path[cur_dim] ← l.key
            l.rule ← MRLP.get_rule(path, max_dim + 1, del_rule.prio)
        end
        l ← rlp.get_next(l)
    end
else
    /* nonterminal case */
    while l ≠ nil ∧ l.key ≤ m.right do
        path[cur_dim] ← l.key
        MRLP.delete(l.rlp, cur_dim + 1, max_dim, del_rule, path)
        l ← rlp.get_next(l)
    end
end
```

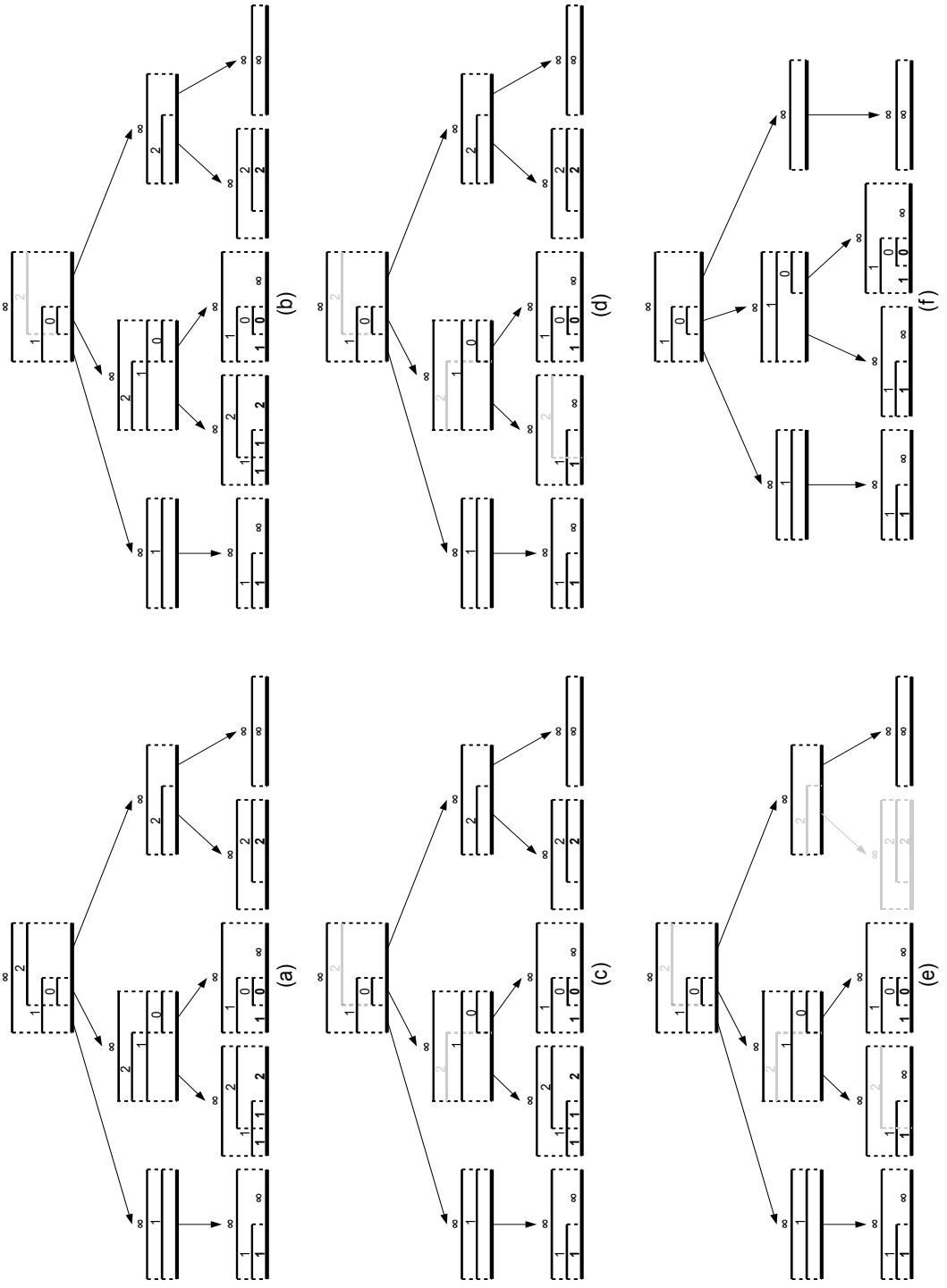

Figure 2.4: Step-by-step run of `MRLP.delete` on a three-dimensional example rule set containing 4 rules. The numbers in the rlps have the same meaning as in figure 2.3. The rlps and matches being deleted are drawn grey.

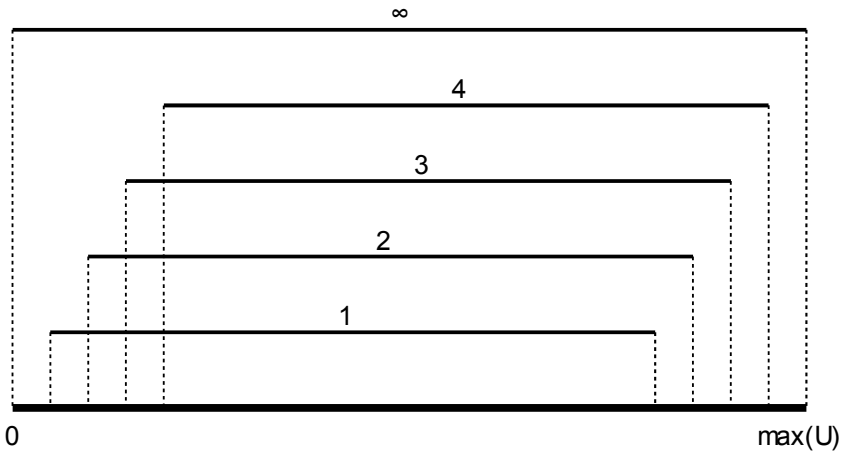

Figure 2.5: Root node in the tree representing $\mathcal{R}_w$ where $n = 5$. The bars, representing the first dimension match of the rules, are signed over with the priority of the rule containing the match.

RLP definition used in this thesis. With them, $T(n,d) \in O(d \cdot log(log(\mathcal{U})))$ is achievable which is independent from the number of rules.

The running time of `MRLP.insert` is described by the following recurrence assuming that *new_rule* is inserted into a d-dimensional worst case rule set of n rules:

$$
\begin{aligned}
T_{insert}(n,1) &= f_{rlp\_locate}(2n-1) + f_{rlp\_insert}(2n-1) + f_{rlp\_locate}(2n) + \\
&\quad f_{rlp\_insert}(2n) + f_{rlp\_locate}(2n+1) + n \cdot f_{rlp\_get\_next}(2n+1) \\
&\in O(f_{rlp\_locate}(2n+1) + f_{rlp\_insert}(2n) + n \cdot f_{rlp\_get\_next}(2n+1))
\end{aligned}
$$

$$
\begin{aligned}
T_{insert}(n,d) &= f_{rlp\_locate}(2n-1) + T_{space}(n,d-1) + f_{rlp\_insert}(2n-1) + \\
&\quad f_{rlp\_locate}(2n) + T_{space}(1,d-1) + f_{rlp\_insert}(2n) + \\
&\quad f_{rlp\_locate}(2n+1) + n \cdot f_{rlp\_get\_next}(2n+1) + \\
&\quad \textstyle\sum_{i=1}^{n} T_{insert}(i,d-1) \\
&\in O(f_{rlp\_locate}(2n+1) + f_{rlp\_insert}(2n) + T_{space}(n,d-1) + \\
&\quad n \cdot f_{rlp\_get\_next}(2n+1) + n \cdot T_{insert}(n,d-1)) \\
&\in O(f_{rlp\_locate}(2n+1) + f_{rlp\_insert}(2n) + n^{d-2} \cdot f_{rlp\_space}(2n-1) + \\
&\quad n \cdot f_{rlp\_get\_next}(2n+1) + n \cdot T_{insert}(n,d-1)) \\
&\in O(n^{d-1} \cdot (f_{rlp\_locate}(2n+1) + f_{rlp\_insert}(2n)) + \\
&\quad n^d \cdot f_{rlp\_get\_next}(2n+1) + \\
&\quad \textstyle\sum_{i=0}^{d-2} (n^i \cdot (f_{rlp\_locate}(2n+1) + f_{rlp\_insert}(2n) + \\
&\qquad n \cdot f_{rlp\_get\_next}(2n+1) + n^{d-i-2} \cdot f_{rlp\_space}(2n-1)))) \\
&\in O(n^{d-1} \cdot (f_{rlp\_locate}(2n+1) + f_{rlp\_insert}(2n)) + \\
&\quad n^d \cdot f_{rlp\_get\_next}(2n+1) + n^{d-2} \cdot f_{rlp\_space}(2n-1))
\end{aligned}
$$

Thus, $f_{rlp\_locate}(n), f_{rlp\_insert}(n), f_{rlp\_space}(n) \in O(n)$ and $f_{rlp\_get\_next}(n) \in O(1)$ yield $T_{insert}(n,d) = O(n^d)$.

The running time of `MRLP.delete` is described by the following recurrence assuming that *del_rule* is the smallest priority rule of a d-dimensional worst case rule set consisting of n rules:

$$\begin{aligned}
T_{delete}(n,1) &= f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1) + f_{rlp_locate}(2n-2) + \\
&\quad f_{rlp_get_next}(2n-2) + f_{rlp_delete}(2n-2) + \\
&\quad f_{rlp_locate}(2n-3) + (n-3)\cdot(dn + f_{rlp_get_next}(2n-3)) \\
&\in O(f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1) + n\cdot f_{rlp_get_next}(2n-2) + n^2)
\end{aligned}$$

$$\begin{aligned}
T_{delete}(n,d) &= f_{rlp_locate}(2n-1) + T_{space}(n,d-1) + f_{rlp_delete}(2n-1) + \\
&\quad f_{rlp_locate}(2n-2) + f_{rlp_get_next}(2n-2) + T_{space}(2,d-1) + \\
&\quad f_{rlp_delete}(2n-2) + f_{rlp_locate}(2n-3) + \\
&\quad (n-3)\cdot f_{rlp_get_next}(2n-3) + \textstyle\sum_{i=3}^{n-1} T_{delete}(i,d-1) \\
&\in O(f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1) + T_{space}(n,d-1) + \\
&\qquad n\cdot f_{rlp_get_next}(2n-2) + n\cdot T_{delete}(n,d-1)) \\
&\in O(f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1) + \\
&\qquad n^{d-2}\cdot f_{rlp_space}(2n-1) + n\cdot f_{rlp_get_next}(2n-2) + n\cdot T_{delete}(n,d-1)) \\
&\in O(n^{d-1}\cdot(f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1)) + \\
&\qquad n^d\cdot f_{rlp_get_next}(2n-2) + n^{d+1} + \\
&\qquad \textstyle\sum_{i=0}^{d-2}(n^i\cdot(f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1) + \\
&\qquad\qquad n\cdot f_{rlp_get_next}(2n-2) + n^{d-i-2}\cdot f_{rlp_space}(2n-1)))) \\
&\in O(n^{d-1}\cdot(f_{rlp_locate}(2n-1) + f_{rlp_delete}(2n-1)) + \\
&\qquad n^d\cdot f_{rlp_get_next}(2n-2) + n^{d+1} + n^{d-2}\cdot f_{rlp_space}(2n-1))
\end{aligned}$$

Thus, $f_{rlp\_locate}(n), f_{rlp\_insert}(n), f_{rlp\_space}(n) \in O(n)$ and $f_{rlp\_get\_next}(n) \in O(1)$ yield $T_{delete}(n,d) = O(n^{d+1})$. The additional factor of n compared to the running time of `MRLP.insert` is the result of the naive implementation of `MRLP.get_rule` which is responsible for the n^2 in $T_{delete}(n,1)$. The running time of this function may be significantly improved in practice by precomputing the subset of rules which are possible candidates for becoming "second-best" rule, i.e. the set of rules $\mathcal{R}_{cand} \subset \mathcal{R}$ overlapping the rule R_{del} which is deleted:

$$\mathcal{R}_{cand} = \{R \in \mathcal{R}\backslash\{R_{del}\} \mid \forall 1 \le i \le d : \ (mat(R))(i) \cap (mat(R_{del}))(i) \neq \emptyset\}$$

Thus, `MRLP.get_rule` only needs to consider the rules in $\mathcal{R}_{cand}$ instead of the whole rule set.

Despite the high time complexity worst case, `MRLP.insert` and `MRLP.delete` perform well for real world rule sets because the size of the rlps usually decreases dramatically with increasing depth if the dimensions are reasonably arranged.

3 Design and implementation of HiPAC

This chapter presents the design and implementation of HiPAC along with its integration as linux kernel module. The structure embodies a top down approach, starting with an overview of packet classification in linux version 2.4. The information presented in this part is the result of an intensive code study using lxr [GG01], an excellent html based cross-referencing tool which facilitates browsing the kernel sources [Org97]. Other helpful resources are [BBD+01], [WPR+02] and [kt00]. Although the description is abstracted from the source code level, it is sometimes necessary to refer to kernel data structures, functions and macros whose definition may be looked up in the source code with the aid of appendix A which provides a list of the names along with the files containing the definitions.

Subsequent to the linux kernel internals, a high level view of the HiPAC design is presented which focusses on the kernel module and its user space counterpart nf-hipac which is used to modify and list the rule set. The kernel module is divided into two main parts: HiPAC front-end and HiPAC core. The former, which is described in this part, is OS dependent and implements the user interface which manages the interaction with the API offered by the HiPAC core.

The third part is concerned with the HiPAC core which implements the novel classification algorithm. The implementation is almost OS independent and runs particularly in user space. To attain the expressiveness of iptables, PCP is generalized to the so-called non-terminal packet classification problem (NPCP). NPCP leads to a natural extension of the classification algorithm from chapter 2 which is incorporated in the HiPAC core.

3.1 Packet classification in linux 2.4

This section provides an overview of the network implementation in the linux 2.4 kernel which is relevant for the integration of HiPAC. The linux forwarding path is illustrated with focus on the netfilter framework which offers a set of so-called hooks inside the forwarding path. These hooks allow the interception of packets in order to influence their further processing, e.g. drop them. The netfilter framework is the basis of the linux 2.4 packet filter called iptables which provides several extensions over PCP as described in chapter 2. Both design and implementation of iptables are examined revealing several deficiencies which have significant impact on performance.

3.1.1 Linux 2.4 forwarding path

Figure 3.1 illustrates the forwarding path of the linux 2.4 kernel for IPv4 packets. The blue ovals represent the netfilter hooks inside the IPv4 network stack. The set of hooks netfilter provides is grouped according to the network entity they reside in. Besides the IPv4 hooks, there are hooks for IPv6, DECnet, ARP and the bridge code. Once a packet passes a certain hook, netfilter hands it on to a set of functions previously registered at the framework. The functions are called in the order of their priority defined at registration time. Each function returns a so-called verdict which influences the further packet processing. The following verdicts are offered:

`NF_ACCEPT:` The subsequent function is called if any. Otherwise, the packet is granted to continue on the forwarding path.

`NF_DROP:` The packet is dropped instantly. Subsequent functions are not processed.

`NF_STOLEN:` Further packet processing is deferred to the current function. Subsequent functions are not processed.

`NF_REPEAT:` The current function is called again.

`NF_QUEUE:` The packet is passed on to user space via netlink[1]. Subsequent functions are not processed. If there is no user space process attached to the netlink socket, the packet is dropped.

Regarding the forwarding path, there are two entry points depending on the origin of the packet. A packet is either received on a network interface and is thus externally originated or it is generated by the local host.

Externally originated packets

An incoming packet issues a hardware interrupt which is handled by the network driver of the corresponding network interface. The handler passes the packet on to a generic interface layer which queues the packet and schedules a software interrupt (softirq). The packet processing is thus deferred to the softirq handler which is called asynchronously. If the packet arrived on a bridge port, it is further treated by the bridge code which essentially decides whether the packet is to be delivered locally or sent out onto another bridge port. In the first case, the packet is reinjected into the generic interface layer after the label of the incoming interface (bridge port), which is stored in the linux packet representation (`struct sk_buff`), has been replaced by the label of the bridge to which the port belongs. In the latter case, the packet is passed on to the traffic shaping facility which is addressed later on.

[1] Netlink is a datagram oriented protocol for communicating information between linux kernel and user space. It is accessible from user space via the common socket API.

If the packet does not arrive on a bridge port, the appropriate network protocol handler is called. In case of an IPv4 packet, the corresponding handler enters the netfilter hook **NF_IP_PRE_ROUTING**. If further processing is granted, the route lookup determines the outgoing interface for the packet and stores whether the packet is locally destined or not. Note that the outgoing interface does not have to be a physical network interface. Instead, it may be a tunnel, the local interface *lo* or a virtual interface like a PPPoE (point-to-point protocol over ethernet) device.

In case the packet is locally destined, it is passed on to the fragmentation reassembly if it is a fragment. The IPv4 fragmentation unit queues fragments and returns the reassembled packet when the last fragment of a series arrives. Fragment series which stay incomplete for a certain time are removed from the queue. Hence, if the packet is a fragment but not the last one of a series, the forwarding path ends here. If the packet is the last fragment of a series, the reassembled packet enters the netfilter hook **NF_IP_LOCAL_IN**. This also happens for packets which are not fragments. If further processing is granted, an upper layer protocol handler is called which is either a transport protocol handler for TCP, UDP, IGMP or ICMP packets or alternatively a tunnel handler for IPIP, GRE or PIM-SM[2] (version 2) tunnels. The tunnel handler decapsulates the embedded packet and reinjects it into the generic interface layer where it traverses the forwarding path a second time. Note that this scheme allows nested tunnels. If the packet is handled by a transport protocol handler instead of a tunnel handler, there are two cases. Further processing of ICMP and IGMP packets is completely conducted in kernel space whereas TCP and UDP packets are queued in a corresponding socket attached to a user space process which is able to read the packet content asynchronously. The packet is also queued in raw sockets which are not necessarily bound to a specific transport protocol[3]. Therewith, the forwarding path for externally originated, locally destined packets is complete.

If the packet is found to be not locally destined, it enters the netfilter hook **NF_IP_FORWARD** after its ttl value has been decremented. If further processing is granted, the packet is fragmented if necessary, and the fragments resp. the unfragmented packet enter the netfilter hook **NF_IP_POST_ROUTING**. If further processing is granted, the link layer address of the destination host is requested. For ethernet links, this may involve sending an ARP request. Note that linux abstracts the link address resolution by the so-called neighbor interface which currently handles ARP and ND (neighbor discovery for IPv6). The packet is queued until the request has been answered or a timeout occurs. If the request is successful, the packet is enqueued into the aforementioned linux traffic shaping facility. Linux traffic shaping provides a very flexible framework for classifying and shaping outgoing packets (egress shaping), e.g. packets may be reordered or limited by bandwidth. After dequeuing a packet from the traffic shaper, it is either sent to the tunnel handler if the outgoing device is a tunnel or passed on to the network device driver which emits the packet. The tunnel handler encapsulates

[2] Protocol independent multicast – sparse mode: multicast routing protocol for multicast groups which are sparsely distributed across a wide area; more bandwidth efficient than DVMRP or MOSPF in this scenario.

[3] Apart from raw sockets, there is another, more low level user space interface called packet sockets.

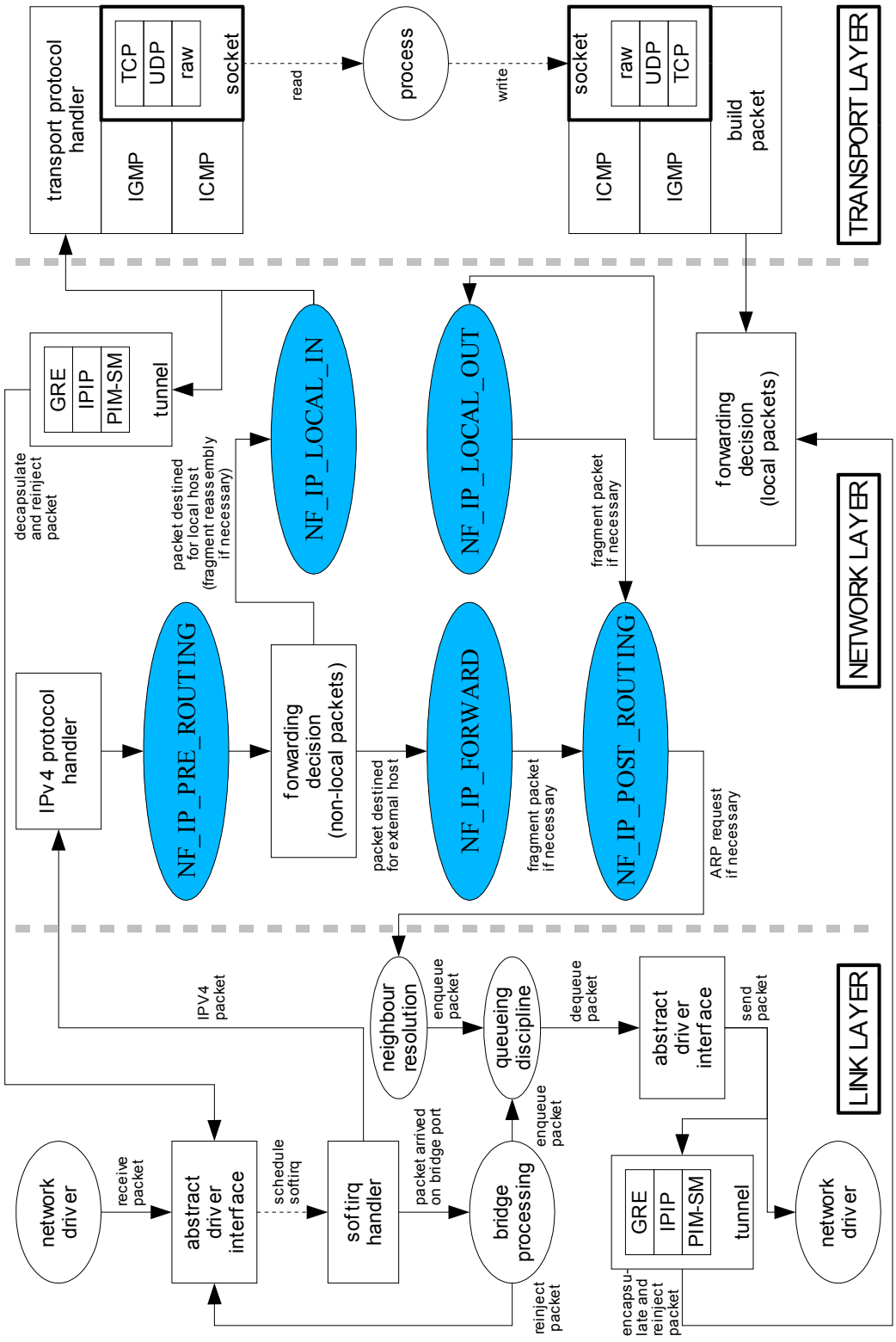

Figure 3.1: Overview of the linux 2.4 kernel forwarding path for IPv4 packets.

the packet and reinjects it into the network layer where a route lookup is performed to determine the outgoing interface for the modified packet. The following steps coincide with the forwarding path of a locally originated packet.

Locally originated packets

A locally originated packet is either created on a kernel event or by a user space process which is attached to a socket. In case of an IPv4 packet, a route lookup is performed after the packet has entered the network layer. Note that a route lookup is not necessary if the packet is associated with a connected socket since the socket caches the forwarding decision. After the outgoing interface has been determined, the packet enters the netfilter hook **NF_IP_LOCAL_OUT**. If further processing is granted, the packet is fragmented if necessary and the fragments resp. the unfragmented packet enter the netfilter hook **NF_IP_POST_ROUTING**. Afterwards, the processing is the same as already described for externally originated packets which are not locally destined.

Note that the forwarding path allows local communication, i.e. forwarding of locally generated packets which are locally destined, using the local interface *lo*. The corresponding device driver emits a packet by passing it on to the receive function of the generic interface layer and is thus sending and receiving device at the same time.

In contrast to externally originated packets, local packets don't pass the netfilter hook **NF_IP_PRE_ROUTING**. This design decision is motivated by the fact that the forwarding decision is already determined by the socket for most locally generated packets which means that there is usually no "pre routing state". The consequence is an additional requirement for the functions attached to the **NF_IP_LOCAL_OUT** hook. If one of these functions modifies the source IP address, destination IP address or tos value of the packet, the forwarding decision must be renewed before the packet is allowed to enter the forwarding path again since the result of the decision depends on these values[4].

3.1.2 Concepts and design of iptables

Iptables uses the netfilter framework to implement a packet filter for IPv4 packets. There is also a packet filter implementation for IPv6 packets called ip6tables which is very similar to iptables. Besides stateless packet filtering based on rule sets, iptables provides with the so-called *connection tracking* a means to associate packets with connections which enables *stateful packet filtering*. While packet filtering never alters the packet, iptables offers several packet modification capabilities, especially network address translation (NAT).

Figure 3.2 illustrates the functions registered to the IPv4 netfilter hooks. There are five entities including three *tables*, connection tracking and one qdisc (queueing discipline). The qdisc is not related to iptables but instead belongs to the traffic shaping facility. It is called "ingress qdisc" because it controls the incoming traffic (policing) unlike the other qdiscs which control outbound packets. The ingress qdisc does not maintain a queue of

[4]The linux representation of a packet is associated with a special value called netfilter mark. If this value is changed, the forwarding decision must also be renewed since it may be influenced by the mark.

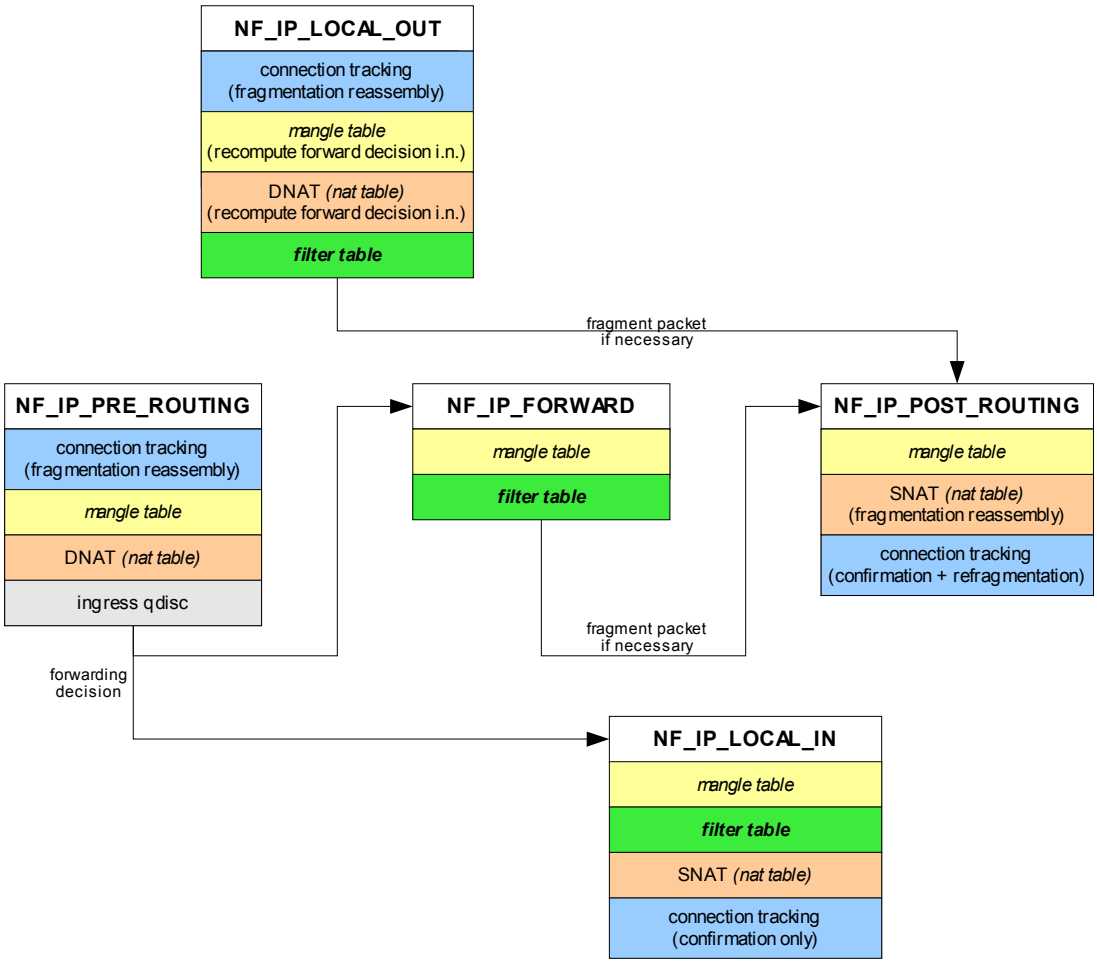

Figure 3.2: Iptables functions registered to the IPv4 netfilter hooks. The functions are displayed in descending priority from top to bottom of each box, i.e. the topmost function is executed firstly. Functions belonging to the same entity are drawn in the same color. ["i.n." means "if necessary"]

packets so that it is only possible to implement simple rate limiting which decides for each packet whether it should be dropped or not [5]. Note that each entity is independent from the others apart from the NAT facility which requires connection tracking.

A *table* is the iptables denotation for rule set. It differs from a PCP rule set as defined in chapter 2 in several ways which are discussed later. The tables along with their purposes are stated in the following summary.

filter table: Rule set for stateless or stateful packet filtering depending on whether connection tracking is enabled or not. The packet remains unchanged.

nat table: Rule set for network address translation (NAT). Iptables distinguishes two types of NAT: source NAT (SNAT) and destination NAT (DNAT). SNAT modifies the source address of the IPv4 header while DNAT changes its destination address. For TCP and UDP packets, it is also possible to modify the source port (SNAT) and destination port (DNAT) of the TCP / UDP header.

mangle table: Rule set used to manipulate the packet resp. its representation, e.g. modify the tos field of the IPv4 header or mark the packet.

Unlike the filter and mangle table which are processed for every packet, the nat table is only processed for packets initiating a new connection. The rule set is used to create NAT mappings for selected connections. With the aid of connection tracking, it is possible to apply NAT mappings to packets belonging to already established connections without processing the nat table. Note that DNAT happens before the forwarding decision is computed while SNAT happens afterwards. Thus, destination IP modifications affect the routing decision while source IP modifications don't[6]. In case of the **NF_IP_LOCAL_OUT** hook, it is necessary to recompute the forwarding decision if the packet is modified by the nat or mangle table (or due to an existing NAT mapping) since the forwarding decision has already been determined before the packet enters the hook. Both NAT and connection tracking require the packet to be unfragmented. Fragmentation reassembly is done by connection tracking as one of its first tasks at the **NF_IP_PRE_ROUTING** and **NF_IP_LOCAL_OUT** hook. The packet is fragmented again after the **NF_IP_LOCAL_OUT** resp. **NF_IP_FORWARD** hook which means that fragmentation reassembly has to be performed a second time at the **NF_IP_POST_ROUTING** hook. Before the packet finally exits this hook, it is refragmented again (if necessary) by the connection tracking facility.

Connection tracking

There are two entry (**NF_IP_PRE_ROUTING**, **NF_IP_LOCAL_OUT**) and two exit points (**NF_IP_POST_ROUTING**, **NF_IP_LOCAL_IN**) for a packet traversing the IPv4 netfilter

[5] The IMQ patch (`http://trash.net/~kaber/imq/`) implements a virtual queueing interface which makes ingress policing as flexible as egress shaping.

[6] Traditionally, the forwarding decision is solely based on the destination IP address but linux 2.4 offers policy routing which incorporates the source IP address into the forwarding decision.

hooks. If connection tracking is enabled, it is called both as the very first function on either entry hook and as the very last function on either exit hook. Connection tracking starts with fragmentation reassembly, i.e. packet fragments are queued until all fragments necessary to reassemble the original packet have been received. Further processing is done by transport layer protocol specific handlers which are available for TCP, UDP and ICMP or by a generic handler for all other protocols. Since UDP and ICMP are connectionless protocols, it is only possible to track network flows, i.e. source and destination address and port for UDP and source and destination address plus ICMP type and code for ICMP. In addition to the network flow, TCP connection tracking stores certain states (not all) of the connection according to the TCP state machine. The network flow information is stored in two separate data structures (*conntrack tuples*), one for the original direction defined by the packet initiating a new connection / flow (*initial packet*) and the other for the reply direction. In case of ICMP packets, original and reply direction are defined by corresponding request and reply messages. Each connection / flow is represented by an instance of the data structure `struct ip_conntrack` which is stored in a hash table. In fact, each such data structure is contained twice in the hash, once for each conntrack tuple which is the basis for the hash key. Upon receiving an initial packet, the data structure and **both** conntrack tuples are created and attached to the packet instead of being inserted into the hash table. The insertion is accomplished within the scope of the *confirmation* (cf. figure 3.2) which happens at the end of each exit hook. This ensures that connection data structures created by initial packets and dropped during their traversal through the netfilter hooks are never inserted into the hash table.

The principal task of connection tracking is to associate a connection tracking state with each packet. There are three such states which are independent of the transport layer protocol:

NEW: All initial packets which are not expected (see **RELATED**) and processed before the first packet in reply direction obtain this state.

ESTABLISHED: Packets belonging to a connection for which a packet in reply direction has been received including the first reply packet obtain this state.

RELATED: Connection tracking implements an interface for functions analyzing the application protocol data of packets. The so-called *conntrack helpers* are mainly used to define *expected connections*. An expected connection is a connection which is initiated in the course of an already established connection, e.g. ftp communicates files over a separate data connection which is initiated by the control connection. Hence, all initial packets which are expected and processed before the first packet in reply direction obtain the state `RELATED`. Note that all expected connections are stored in a list which has to be traversed for each initial packet.

INVALID: Packets which cannot be associated with one of the states above obtain this state. This happens on a processing error (e.g memory shortage), if the packet is too small, if the transport layer protocol handler rejects the packet (e.g. if the

initial packet is a TCP packet without the syn flag set) or if the packet does not reveal a reply direction which is the case for all ICMP packets which are neither requests nor replies.

Apart from the states above, connection tracking provides an additional state called ASSURED which is transport layer protocol dependent. For UDP connections, it is equivalent to ESTABLISHED. For TCP connections, it is set for all packets processed after the three-way handshake including the ack packet after the syn-ack. Packets belonging to other protocols including ICMP are never set ASSURED. The state is used to decide which entries may be removed from the connection tracking hash in case the number of entries reaches a pre-defined maximum (ip_conntrack_max). ASSURED connections are never removed from the hash whereas other connections are deleted if necessary using a simple LRU (least recently used) scheme.

Compared to the other iptables entities registered to the netfilter hooks, connection tracking is the only one which is rule set independent. In the common scenario where the rules of the filter table are defined such that packets belonging to established connections are generally accepted while initial packets are filtered, a rule set update may cause established connections to become invalid in the sense that if an initial packet creating one of these connections arrived after the change, it would be dropped by the new rule set. However, since the connection tracking hash is not updated after a rule set change, invalidated established connections are not interrupted.

Iptables rule set

An iptables rule set is called *table*. There are three standard tables: **filter**, **nat** and **mangle**. Each table consists of a set of chains, each being an ordered list of rules. The set of chains can be divided in two disjoint subsets: *built-in chains* and *user-defined chains*. The number of built-in chains corresponds to the number of netfilter hooks the table is registered to. Each built-in chain is uniquely associated with a netfilter hook. In contrast to user-defined chains, built-in chains contain a policy rule whose target is one of the netfilter verdicts NF_ACCEPT and NF_DROP. Each rule in a chain consists of a set of matches and one target. In contrast to the formal rule definition 2.2 used to state PCP, an iptables match is not restricted to represent a numeric range. Instead, it is implemented as C function which makes stateful and even nondeterministic matches possible, i.e. a match is not guaranteed to behave the same for two equal packets. Nevertheless, common matches like source / destination IP, protocol and source / destination port matches are of course stateless and deterministic. An iptables target is either a user-defined chain or a function returning a certain verdict. The verdict is either a netfilter verdict, i.e. NF_ACCEPT, NF_DROP, NF_STOLEN, NF_REPEAT or NF_QUEUE, or an iptables verdict, i.e. IPT_CONTINUE or IPT_RETURN. Rules with a netfilter verdict as target are called *terminal* whereas rules with the target IPT_CONTINUE are called *non-terminal*.

Rules with a user-defined chain as target are called *jump rules*. They are used to interconnect the chains which form a directed acyclic graph (DAG) where each jump rule in a chain c with target c' creates the edge (c, c'). Built-in chains are per definition

source nodes in the graph. If a user-defined chain is a source node, it is said to be unconnected which means that the rules in the chain cannot match any packet. Note that each rule including the policy rules of the built-in chains are provided with a 64 bit packet and byte counter. Every time a packet matches a rule, its packet counter is incremented by one and its byte counter is incremented by the size of the IPv4 packet indicated by the value of the total length field in the IPv4 header.

The iptables match algorithm is described by the function `iptables_match` (recursive and simplified version of the function `ipt_do_table()`). In case of the filter and mangle table, it is called for every packet which enters a netfilter hook the table is registered to, provided that all higher prioritized functions registered to the same hook return `NF_ACCEPT`. In case of the nat table, the match function is only called for packets initiating a new connection. Hence, given a packet *p* passing a netfilter hook *h* the match function call for a table *t* is: `iptables_match`(*t*, *c*, *p*) where *c* is the built-in chain associated with *h*. The rules are evaluated linearly in the order of their appearance in the chain. For each rule, *rule*.match(*packet*) calls the match function of each match in *rule*. The match function returns TRUE if the packet applies, FALSE if it does not apply or HOTDROP if the packet should be dropped. Consequently, if a match function does not return TRUE, *rule*.match(*packet*) does not process the remaining match functions and immediately returns the result of the function. Note that the packet and byte counter of a rule returning HOTDROP is not updated which means that the rule is not considered to match the packet. If a rule matches a packet, its target is evaluated. There are four cases:

1. If the rule is terminal, `iptables_match` is finished and the corresponding netfilter verdict is returned.
2. If the rule is non-terminal, the next rule in the chain is evaluated.
3. If the rule is a jump rule, the rules in the target chain are evaluated.
4. If the rule target indicates `IPT_RETURN`, there are two cases:
   a) The rule is contained in a built-in chain: `iptables_match` is finished, the policy rule applies and its verdict is returned.
   b) The rule is contained in a user-defined chain: the remaining rules after the previously processed jump rule leading to the current chain are evaluated.

After all rules of a user-defined chain have been evaluated, the remaining rules after the previously processed jump rule leading to the chain are evaluated. If no terminal rule is found to match the packet, the policy rule applies.

3.1.3 Implementation of iptables

The iptables framework consists of a kernel part and a user space part. The user space part encompasses the “iptables” command line tool which is used to output and modify the tables. The supported operations can be divided in two categories: per rule and per

Function `iptables_match`(*table*, *chain*, *packet*)

```
psize ← packet.size
rule ← chain.get_first_rule()
while rule ≠ nil do
    match ← rule.match(packet)
    if match = HOTDROP then
        return NF_DROP
    end
    if match = TRUE then
        rule.packet_count ← rule.packet_count + 1
        rule.byte_count ← rule.byte_count + psize
        if rule.target is chain then
            assert(rule.target is user-defined chain in table)
            verdict ← iptables_match(table, rule.target, packet)
        else
            assert(rule.target is a function)
            verdict ← rule.target(packet)
        end
        if verdict = IPT_RETURN then
            if chain is user-defined chain in table then
                return IPT_CONTINUE
            else
                chain.policy_rule.packet_count ← chain.policy_rule.packet_count + 1
                chain.policy_rule.byte_count ← chain.policy_rule.byte_count + psize
                return chain.policy_rule.verdict
            end
        end
        if verdict ≠ IPT_CONTINUE then
            assert(verdict is netfilter verdict)
            return verdict
        end
    end
    rule ← chain.get_next_rule(rule)
end
if chain is built-in chain in table then
    chain.policy_rule.packet_count ← chain.policy_rule.packet_count + 1
    chain.policy_rule.byte_count ← chain.policy_rule.byte_count + psize
    return chain.policy_rule.verdict
else
    return IPT_CONTINUE
end
```

chain operations. The former are: append, insert, delete and replace a single rule and modify the target of the policy rule of a built-in chain. The latter are: add, delete, rename a chain, list, delete all rules of a chain and zero the counters of all rules in a chain. The rule set is stored in kernel space in one virtually contiguous memory block, i.e. the memory pages are allocated via `vmalloc()`. On SMP (symmetric multiprocessing) systems, the table is duplicated for each cpu, i.e. a total of $3 \cdot number\ of\ cpus$ tables reside in kernel space. The reason behind is twofold. Firstly, counter updates don't require a lock which is otherwise necessary since two cpus may update the same counter at the same time leading to incorrect results due to the update not being atomic at the machine level of most architectures. Secondly, even if 64 bit atomic operations are used, updating a counter by one cpu invalidates the cached counter value of the remaining cpus and thus leads to additional memory accesses. This memory performance degrading effect is known as cache thrashing.

A rule set update always involves kernel and user space activity. The kernel interface to control the update is implemented as a set of socket options. They can be used in user space via `getsockopt()` and `setsockopt()`. The following socket options are available:

`IPT_SO_GET_INFO`: Returns some information about the table, e.g. the netfilter hooks associated with the built-in chains, the number of rules and the total size of the table.

`IPT_SO_GET_ENTRIES`: Copies the requested table associated with cpu 0 to a user space buffer using the following algorithm:

- allocate and zero out temporary counter array whose length is equal to the number of rules in the table
- *write lock table*
- for each per cpu table: for each rule in table: add rule counter to the corresponding field of the counter array
- *write unlock table*
- copy table from cpu 0 to user space buffer
- for each rule in user space table: set counter to the value of the corresponding field of the counter array and store iptables matches and targets by their names

`IPT_SO_SET_REPLACE`: Replaces all copies of the requested table by a copy of a supplied user space table and returns atomic counter snapshot of original table using the following algorithm:

- allocate memory for s tables, each of the size of the user space table (s = number of cpus)
- copy user space table to new table t associated with cpu 0

- allocate and zero out temporary counter array whose length is equal to the number of rules in *t*
- for each rule in *t*: check alignment and offsets of the rule within *t* and initialize the rule counter to 0
- **loop detection:**
  for each chain in *t*: unmark chain
  for each built-in chain *c*: if `iptables_loop_check`(*c*) = LOOP then cancel operation
- for each rule in *t*: load kernel modules for matches and target if necessary and call `checkentry()` function (consistency check) for each match and target
- clone *t* for the remaining cpus
- *write lock table*
- replace current tables by new tables (set pointers)
- *write unlock table*
- for each old per cpu table: for each rule in table: add rule counter to the corresponding field of the counter array
- for each rule in old table from cpu 0: call `destroy()` function (release resources) for each match and target
- copy temporary counter array to user space counter array

Note that the counters of the new table are set to 0 while the counters of the original table are written to user space.

`IPT_SO_SET_ADD_COUNTERS`: Add counter snapshot supplied by user space to the counters of the corresponding rules in the requested table using the following algorithm:

- allocate temporary counter array whose length is equal to the user space counter array
- copy user space counter array to temporary counter array
- *write lock table*
- for each rule in table from cpu 0: add counter from the corresponding field of the counter array to the rule counter
- *write unlock table*

The socket options provide a static interface which can only be used to install a complete rule set and read an atomic counter snapshot. The implementation as described above reveals several severe performance bottlenecks. First of all, the loop detection algorithm (recursive and simplified version of the function `mark_source_chains()`) is naive. Loop detection is usually implemented using depth first search where all nodes

Function `iptables_loop_check`(*chain*)

```
if chain is marked then
    return LOOP
else
    mark(chain)
end
forall rules r in chain do
    if r.target is chain then
        if iptables_loop_check(r.target) = LOOP then
            return LOOP
        end
    end
end
unmark(chain)
return NO_LOOP
```

are initially marked white. After visiting a node, it is marked grey and after all edges are recursively processed, it is marked black. If a grey node is visited, the graph contains a loop. The running time is $\Theta(n+m)$ where n is the number of nodes and m is the number of edges since each node and edge is visited once. For each source node c (built-in chain), the iptables loop detection algorithm visits each node v (user-defined chain) reachable from c as many times as there are paths from c to v. Hence, the running time for a complete DAG, i.e. a DAG with n nodes and $\frac{1}{2}n(n-1)$ edges, is $\Theta(2^n)$. Another severe performance bottleneck is caused by the write lock used to protect the table from concurrent read operations which occur during `iptables_match`. As long as the write lock is acquired, no packet can be classified. This is not a problem for the socket option `IPT_SO_SET_REPLACE` where the operation during the lock is constant time but in option `IPT_SO_SET_ADD_COUNTERS`, the critical operation takes $\Theta(n)$ time and in option `IPT_SO_GET_ENTRIES`, the critical operation takes even $\Theta(s \cdot n)$ time where s is the number of cpus and n the number of rules.

The static interface and data structure are motivated by the need for atomic counter snapshots. HiPAC implements a scheme which also achieves atomic counter snapshots but requires only constant time locking and thus avoids stale traffic during rule updates and listings. After all, the iptables update performance is unnecessary bad, especially considering the simple classification algorithm. Installing a rule set consisting of n rules takes $\Theta(s \cdot n^2)$ time assuming that the loop detection scales linearly with the number of rules.

The user space implementation of iptables involves a library called libiptc which uses the kernel socket options interface. The following libiptc procedure applies to all iptables commands:

- `getsockopt(IPT_SO_GET_INFO)`
- allocate memory for table

- `getsockopt(IPT_SO_GET_ENTRIES)`
- apply rule set modification to table (may involve reallocation of the table)
- `setsockopt(IPT_SO_SET_REPLACE)`
- modify counter array if necessary
- `setsockopt(IPT_SO_SET_ADD_COUNTERS)`

The only exception is the rule listing which is finished after the second `getsockopt()` call.

3.1.4 PCP vs. iptables

Comparing iptables packet classification with definition 2.6 of PCP, the former reveals to be more expressive. The following list summarizes the extensions iptables provides over PCP:

Iptables targets may be non-terminal: PCP always returns the smallest priority rule applying to a given packet. Hence, all PCP actions (targets) are terminal by definition. Incorporating non-terminal actions into the rules of a PCP rule set leads to a more general problem which returns the smallest priority, *terminal* rule applying to a packet along with all non-terminal rules which also apply to the packet and whose priority is smaller than the one of the terminal rule. The policy rule of such a rule set must be terminal to guarantee that there is always a smallest priority, terminal rule. This generalized packet classification problem is called **non-terminal packet classification problem (NPCP)** (cf. section 3.3.1).

Iptables matches are general functions: Since iptables matches are potentially non-deterministic and stateful programs, each rule has to be evaluated at runtime for each packet until a rule with terminal action is found. Considering the match function as black box, it is impossible to implement a more efficient classification algorithm than the trivial one (`iptables_match`). However, in practice the majority of iptables rules consists of matches representing a numeric range of values for a certain packet header field. This observation leads to the distinction between **native matches** (numeric ranges) and **procedural matches** (iptables matches not expressible as numeric range).

Iptables matches may return HOTDROP: This ad hoc concept allows iptables matches to act like a restricted target. Every iptables match which potentially returns HOTDROP is considered a procedural match. Note that generalizing this idea would lead to matches which are allowed to return an arbitrary action. However, this functionality would hardly be useful in practice.

Iptables rule sets consist of several interconnected chains forming a DAG: This PCP extension is not covered in this thesis. [Bel04] shows that a DAG of chains

can always be expressed by a single chain which means that the DAG does not increase the expressiveness of the rule set.

3.2 HiPAC front-end

In this section, the design of HiPAC is described with focus on the OS dependent front-end and its integration into the linux 2.4 kernel. The front-end mainly implements an interface which allows the user to interact with the HiPAC core implementing the novel classification algorithm. Besides, it offers a number of statistical information concerning the algorithm and shows how to support the iptables network interface match, which is based on strings rather than numeric ranges, as native match.

3.2.1 Design overview

HiPAC provides high performance packet classification while at the same time maintaining the flexibility offered by the iptables framework. From the user's perspective, it is an alternative filter table with the same capabilities as the iptables filter table. Rule set updates and listings are handled by a user space tool named nf-hipac which is almost completely compatible to the iptables user space tool regarding both syntax and semantics. HiPAC overcomes all critical design mistakes of iptables illustrated in section 3.1.3 and achieves:

- minimal locking times, i.e. critical operations interrupting packet classification are always O(1).
- atomicity of counter snapshots without halting packet classification.
- dynamic rule set updates, i.e. instead of the whole rule set, only a single rule is submitted to the kernel per update.
- efficient loop detection.
- improved error handling by using netlink instead of the socket options interface.

Condensed, HiPAC encompasses the following features:

- Dynamic rule set updates: all iptables per rule and per chain operations are supported.
- Generic support for iptables matches and targets.
- Native (numeric range) matches: source / destination IP, transport layer protocol, source / destination port (TCP and UDP), fragment, TCP flags (only syn flag), ICMP type and code, TTL, connection tracking state (requires connection tracking kernel module) and network interface match (special case since interfaces are specified as strings).

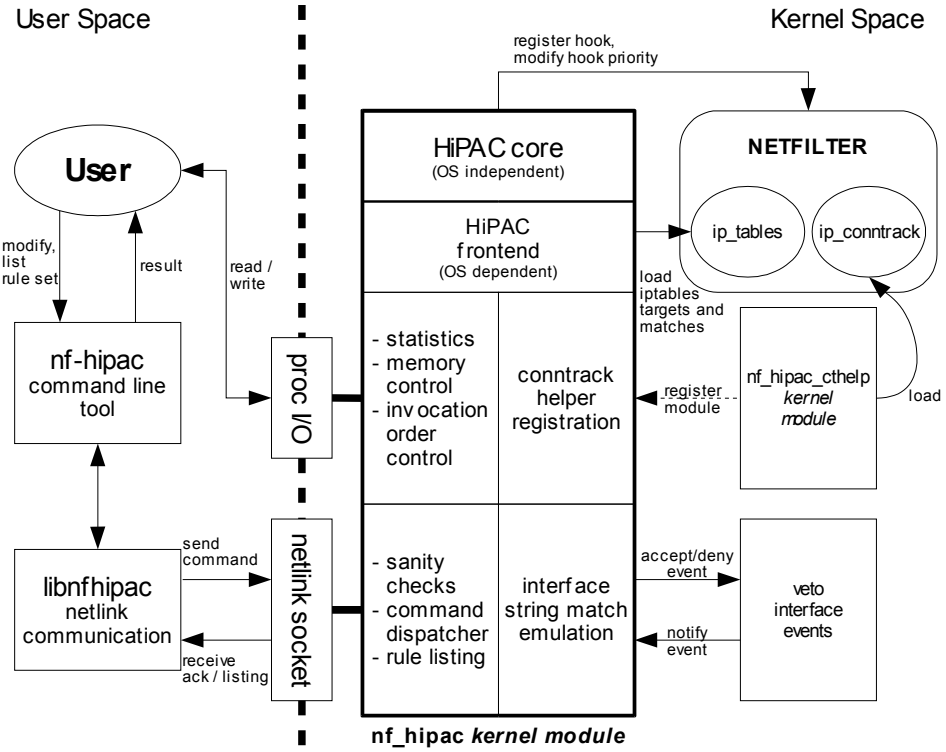

Figure 3.3: Overview of the integration of the HiPAC framework into the linux 2.4 kernel.

- Negation of native matches.
- 64 bit rule counters (packet and byte counter).
- User-defined chains support (cf. [Bel04]).
- Support for all linux 2.4 architectures including 64 bit and SMP architectures.
- Algorithm statistics (e.g. memory usage) available via proc files.
- Maximum memory usage is restrictable by the user.
- Coexistence with iptables, i.e. iptables rule sets may be used together with HiPAC.

Figure 3.3 illustrates the HiPAC components and their integration into linux 2.4. Similar to the iptables framework, there is a user space and a kernel space part. The user space part encompasses the program nf-hipac and the library libnfhipac which implements a netlink based protocol for exchanging rule set updates and listings between kernel and user space and provides a number of strings describing error codes.

The kernel space part consists of two kernel modules, namely `nf_hipac` and `nf_hipac_cthelp` which can also be statically compiled as part of the kernel image.

The latter is a very simple module which is solely used to dynamically load the connection tracking kernel module. The `nf_hipac` kernel module implements the entire functionality which is divided in two parts: the HiPAC core which implements the algorithm API and the HiPAC front-end which uses the API and implements the interfaces necessary to interact with the user space. The HiPAC core is designed with minimal assumptions about the execution environment to make the algorithm as OS independent as possible. Apart from memory allocation functions and kernel locks, the core has no further knowledge about the OS. In particular, kernel data structures like the representation of network packets are not present in the core. All operations on kernel specific data structures are implemented by the front-end and passed to the core via function pointers. This approach allows the core to be implanted in other OS's and environments and to be used for other purposes than firewalling, e.g. for problems reducible to instances of PCP. In particular, it is possible to test the core in user space which is shown in chapter 4.

The HiPAC front-end registers to the netfilter hooks **`NF_IP_LOCAL_IN`**, **`NF_IP_FORWARD`** and **`NF_IP_LOCAL_OUT`** like the iptables filter table does. Moreover, it implements the following functionalities:

Kernel-user communication: Netlink based protocol used to submit commands for rule set modifications from user space to kernel space and rule set listings in the opposite direction. All messages are checked for validity.

Proc user interface: Algorithm statistics computed by the core are translated into readable output available via proc files. One proc file (`/proc/net/nf-hipac/info`) can be written to, in order to change the upper bound for memory usage or the invocation order of the filter table and HiPAC, i.e. the priority HiPAC uses to register to the netfilter hooks may be dynamically modified so that either HiPAC is called before the filter table or vice versa.

Interface string match emulation: The iptables network interface match is based on strings. In particular, the interface referred to by a string need not to exist so that it is not possible to use the interface index (unique number associated with each interface) as a basis for a semantically equivalent native match. To overcome the problem, the interface string match emulation assigns a unique number to each interface string in use and maps all interface indices of active devices to the corresponding unique number. In this context, it is necessary to monitor and if necessary veto events changing the status of an interface.

Connection tracking registration: On module loading time, `nf_hipac_cthelp` registers to `nf_hipac`. Since `nf_hipac_cthelp` requires the connection tracking module `ip_conntrack` to be loaded first, `nf_hipac` can detect in this way whether connection tracking is enabled or not.

The HiPAC front-end consists of the following program files:

nfhp_com.h: data structures and error codes for netlink communication (shared between kernel and user space)

nfhp_mod.[ch]: kernel-user communication, connection tracking registration, use of HiPAC API, kernel module initialization and finalization

nfhp_dev.[ch]: interface string match emulation

nfhp_proc.[ch]: proc user interface

In addition to the kernel modules, a small number of linux kernel files need to be patched either to support a certain functionality or to improve performance. The overall size of the patch is very small: less than 200 lines of code are added to existing kernel files. The patch adds the following functions resp. macro (cf. appendix A for the corresponding files):

- `BR_HIPAC_LOCK`: Big reader lock used within the HiPAC core to protect the packet classification operation (improved performance compared to read-write locks).
- `register_netdevice_veto()`, `unregister_netdevice_veto()`: The registered function is called on the interface events `NETDEV_UP`, `NETDEV_DOWN`, `NETDEV_CHANGENAME` and decides whether the event is allowed to be completed apart from `NETDEV_DOWN` which cannot be canceled. The veto functionality is required for the interface string match emulation.
- `nf_change_prio_hook()`: Modifies the priority of functions already registered to netfilter hook(s). This function is required to dynamically change the invocation order of HiPAC and the filter table.
- `ipt_init_match()`, `ipt_init_target()`: Initializes an iptables match or target referred to by name which involves loading the corresponding kernel module if necessary. This functions is required for the generic iptables match and target support.
- `netlink_dump_start_cb()`: Delivers a number of netlink packets to a user space process which is required for the rule listing. The implementation provides a more flexible interface than the existing `netlink_dump_start()` in order to improve the listing performance by minimizing the number of packets sent to user space (by increasing their size).

3.2.2 Kernel-user communication

The communication between kernel and user space is based on netlink, a datagram oriented message transfer service available under linux since version 2.2. Netlink is not reliable since packets may be dropped due to memory shortage or other errors. However, packet reordering or corruption cannot occur. On the user space side, netlink is controlled via the standard socket interface. Creating a socket involves selecting the

so-called netlink family, i.e. the kernel entity for which the packets are destined. The netlink header (`include/linux/netlink.h`) defines the available netlink families, e.g. `NETLINK_ROUTE` which is responsible for communicating routing updates and tables. HiPAC defines a new netlink family (`NETLINK_NFHIPAC`) and implements a simple, yet efficient protocol for submitting rule set updates to the kernel and requesting rule set listings from the kernel. The protocol involves two steps. In the first step, the user sends a command message to the kernel which is then verified and processed in kernel space. The second step depends on the command. In case of a rule set update, the kernel sends a status message to the user space indicating whether the update was successful. On the other hand, if a rule listing is requested, the kernel sends a stream of chains, each followed by the rules contained in the chain. Figure 3.4 illustrates both protocol runs. The chain message contains the number of rules which follow, and each rule stores its size so that it is possible to navigate in the stream. Note that each rule must be correctly aligned in the stream because it may contain an arbitrary number of iptables matches (or an iptables target) whose alignment requirement may be different from the alignment requirement of the rule[7]. Generally, the stream is split across several packets where a context switch is performed after a packet has been sent by the kernel so that the user space process attached to the netlink socket is able to read the data. In order to achieve high listing performance, maximum sized packets are sent which minimizes the number of context switches. Linux provides with `netlink_dump_start()` a function suitable for transferring multi packet messages. Unfortunately, it does not allow to select the size of the packets. This functionality is added by `netlink_dump_start_cb()` which is part of the kernel patch.

The basic structure of the messages involved in the communication is illustrated in figure 3.5. The command message has always the same structure regardless which command is used but depending on the actual command, only certain parts of the message need to be filled in by the user space. Note that the command message reserves space for the maximum possible number of native matches for convenience. If the rule contains an iptables match or target or a chain target (jump rule) then the iptables header (`struct ipt_entry`) must be included in the message. This is necessary for compatibility with the iptables framework which demands each iptables match/target to provide a verification function (`checkentry()` function in `struct ipt_match` resp. `struct ipt_target`) which requires a part of the iptables header as argument. This function must be called before the rule is allowed to be included in the rule set. The rule part starting from the iptables header to the final iptables match resp. iptables target (not including the chain target if any) is exactly equal to the iptables rule repre-

[7] Primitive data types, i.e. pointers and 8, 16, 32, 64 bit integers, are stored at memory addresses which are usually a multiple of the size of the data type. This is necessary on some architectures and recommended on others to increase the performance. This alignment requirement for primitive data types also applies to C structs and unions. The maximum alignment requirement on a 32 bit system is usually 32 bit whereas on a 64 bit system, it is 64 bit, e.g. on 32 bit systems, 64 bit integers are aligned at addresses which are a multiple of 4 whereas on 64 bit systems, they are aligned at addresses which are a multiple of 8. Special care must be taken on sparc64 linux systems which have a 32 bit user space and 64 bit kernel space under linux.

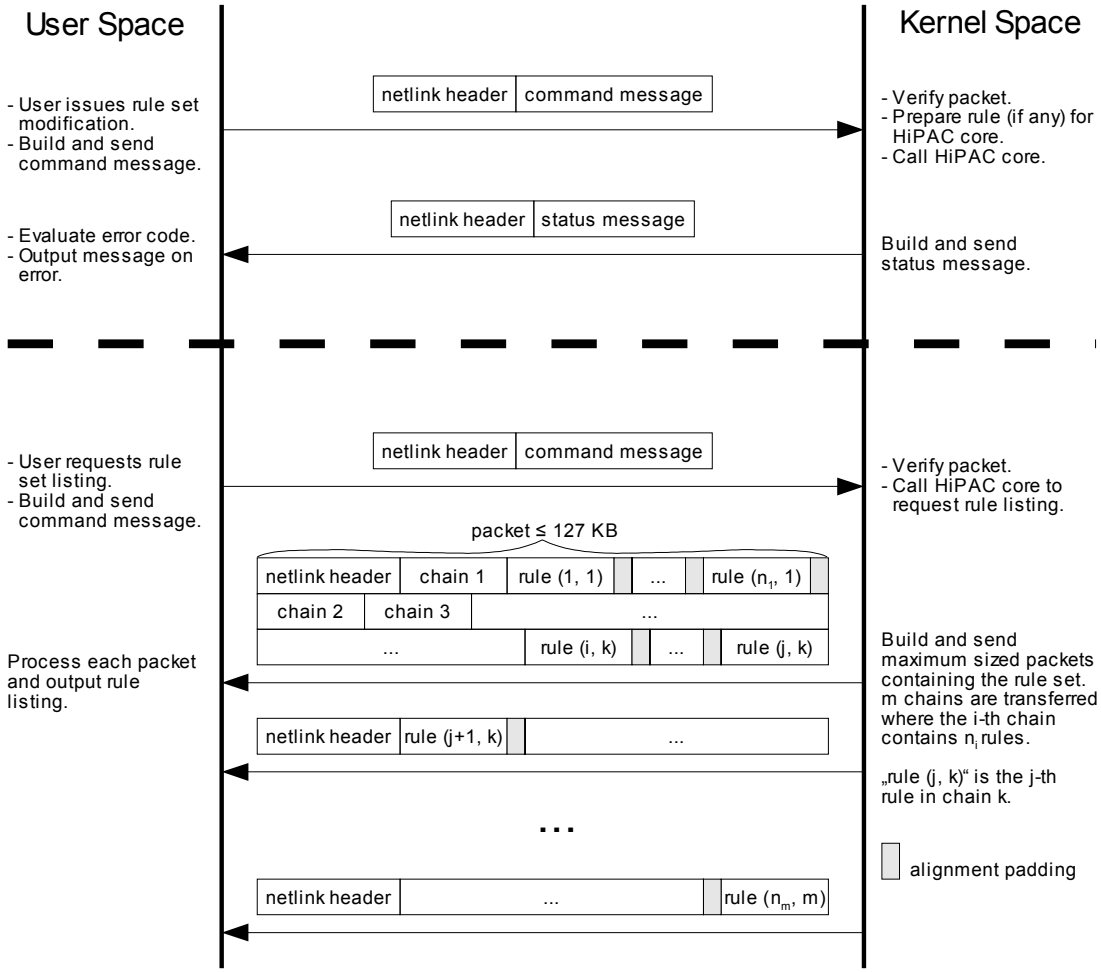

Figure 3.4: Two protocol runs, one for rule set modifying commands (above dashed line) and the other for the listing command (below dashed line).

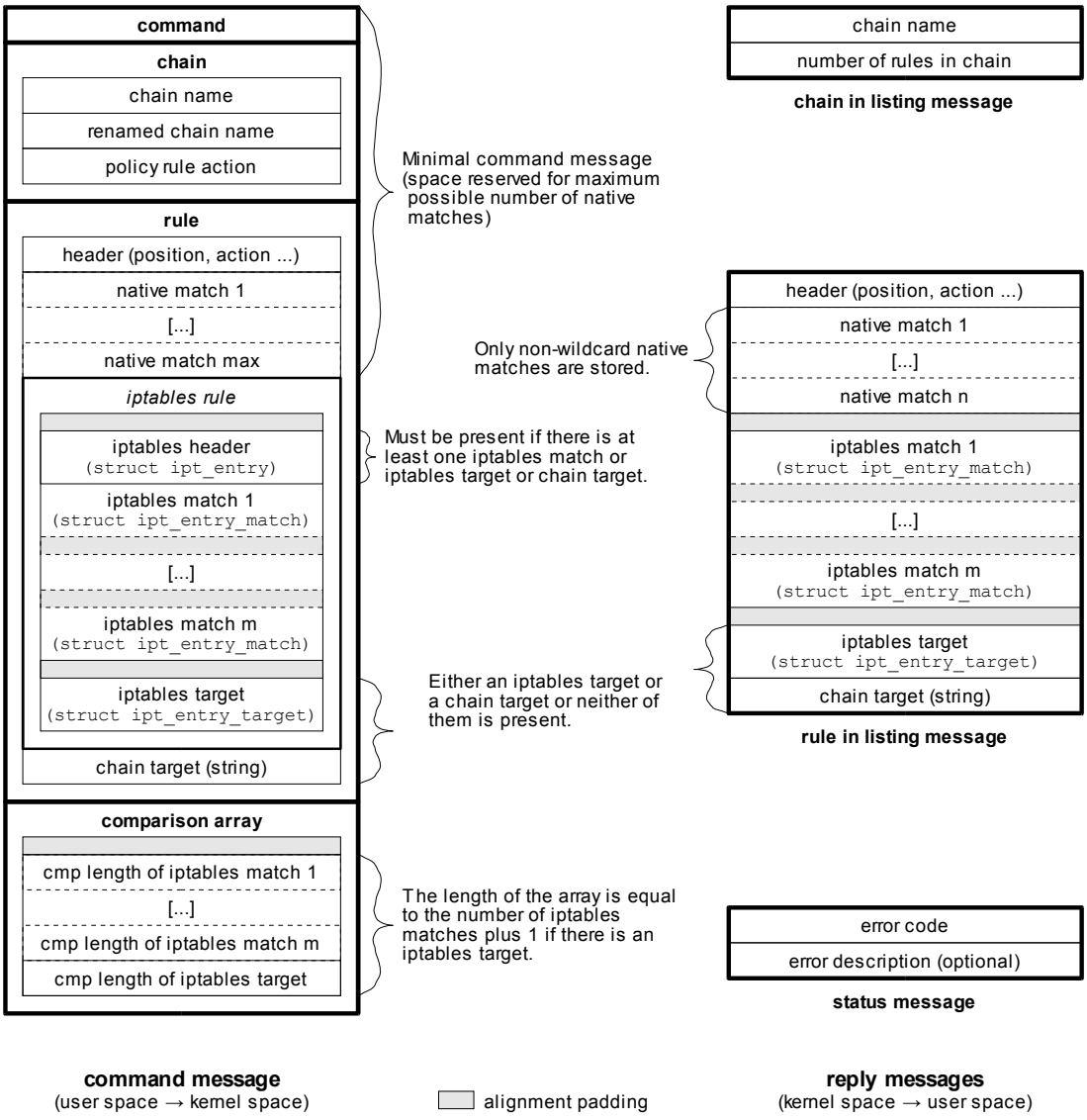

Figure 3.5: Structure of the command and status message and the components of the listing message.

sentation. Like the rules of the listing stream, the iptables matches / target require to be correctly aligned within memory. This may cause extra space to be appended to each iptables match (alignment padding).

Comparing the rule structure sent to the kernel with the one contained in the listing stream, there are two differences. Firstly, the "listing rule" provides only those native matches which are non-wildcard and secondly, it omits the iptables header.

At the end of the command message, a 16 bit unsigned array is placed, the so-called comparison array. It is only present if the command `CMD_DELETE_RULE` (cf. table 3.1) is used and at least one iptables match or an iptables target is contained in the rule. Each entry of the array corresponds to a certain iptables match / target and specifies the size of the match / target subject to the comparison of the given rule with the ones contained in the rule set. To understand this, one has to consider the structure of iptables matches and targets. All iptables matches share a common header (`struct ipt_entry_match`) followed by a match specific data structure. The same holds for iptables targets (common header: `struct ipt_entry_target`). The match / target specific data structure (`include/linux/netfilter_ipv4/ipt_*.h`) generally consists of a public component followed by a private component. The public component is shared between kernel and user space while the private component is only used in kernel space. Up to linux 2.4.24, there is only one match contained in the standard kernel[8] which includes a private component, namely the limit match (`include/linux/netfilter_ipv4/ipt_limit.h`). The values provided in the comparison array are thus used to restrict the comparison to the public component of each match / target.

Table 3.1 summarizes the commands provided by the kernel-user communication interface (`nfhp_com.h`) and states for each command which parts of the command message are required. Note that the commands coincide with those implemented by the user space tool nf-hipac.

Command processing in kernel space Processing commands in kernel space is a two step task. Firstly, the command message is checked for validity and all iptables matches and the iptables target (if any) are initialized. The validity check is crucial to be conducted in kernel space since linux is a monolithic kernel, i.e. a single erroneous kernel component may crash the whole kernel. In particular, kernel components must protect their interfaces to the user space. Hence, the data communicated by user space processes or untrusted components in general must be verified by the responsible kernel entity. HiPAC combines the validity check and the initialization in the function `cmd_check_init()` (`nfhp_mod.c`).

In the second step, the execution of the command is deferred to the HiPAC core which provides a specific function for each command. Since the HiPAC core does not allow concurrent operations, apart from the match function which may be called anytime, command execution must be serialized. This is achieved by the mutex (semaphore for mutual exclusion) `nlhp_lock` which is also used within the proc user interface. Note

[8] Additional iptables matches / targets from `http://www.netfilter.org/` are not considered.

command	description	required parts
CMD_APPEND	append rule at the end of the chain	chain name, rule
CMD_INSERT	insert rule at given position into the chain	chain name, rule
CMD_DELETE_RULE	delete the first rule from the chain which is equal to the given rule	chain name, rule, comparison array
CMD_DELETE_POS	delete rule at given position from the chain	chain name, rule position (header)
CMD_REPLACE	replace rule at given position by the provided one	chain name, rule
CMD_NEW_CHAIN	create new user-defined chain	chain name
CMD_RENAME_CHAIN	rename user-defined chain	chain name, renamed chain name
CMD_SET_POLICY	set action of policy rule contained in the chain	chain name, policy rule action
CMD_ZERO_COUNTERS	set counters of all rules of the chain resp. all chains to 0	chain name
CMD_FLUSH	delete all rules of the chain resp. all chains	chain name
CMD_DELETE_CHAIN	delete user-defined chain resp. all user-defined chains which are empty and not used as chain target of a rule	chain name
CMD_LIST	request listing for chain resp. all chains without rule counters	chain name
CMD_LIST_COUNTER	request listing for chain resp. all chains including atomic snapshot of rule counters	chain name

Table 3.1: Summary of all commands including the corresponding required parts of the command message. The commands in the second half of the table apply for all chains if the chain name is left empty.

that if a listing is requested, the mutex is acquired until the communication is finished or aborted.

3.2.3 Proc user interface

The proc file system is a flexible user interface which allows access to and modification of kernel data structures. Proc files are not stored on any persistent media but instead their content is dynamically constructed on access. Each file provides a read and optionally a write function for this purpose. The read function stores the content in a supplied buffer whose size is one memory page (4 KB on x86). If the content exceeds the buffer size, multiple read calls are issued where an offset value is used to indicate the requested data position.

The HiPAC proc user interface is mainly used to supply statistical information about the HiPAC core in a human readable form. The implementation encompasses several files which are described in table 3.2. Additional files can be easily added by extending the data structure `nfhp_proc` (`nfhp_proc.c`). There are four generic output functions reflecting the different types of statistical data. They are stated in the following list along with the data structure used to represent the information:

- Description (string)
- Scalar value (64 bit unsigned)
- Mapping $[0, l-1] \rightarrow \mathcal{U}_{32}$ (32 bit unsigned array of length l)
- Distribution $\{[\lfloor 2^{i-1} \rfloor, 2^i - 1] \mid 0 \leq i < l-1\} \cup \{[2^{l-2}, \infty[\} \rightarrow \mathcal{U}_{32}$ (32 bit unsigned array a of length l where $a[i],\ 0 \leq i < l-1$ is the value associated with the range $[\lfloor 2^{i-1} \rfloor, 2^i - 1]$ and $a[l-1]$ is the value associated with the range $[2^{l-2}, \infty[$)

Since the text output of a proc file may exceed 4 KB, the maximum memory required for each file is allocated at module loading time, and each time the file is requested from its beginning (offset is 0), both statistics and output are recomputed. The statistics function calls are protected by the `nlhp_lock` mutex which is also used to serialize the netlink communication. It is necessary to ensure that at most one HiPAC core function is executed at a given time (apart from the match function which is allowed to run anytime).

In table 3.2 the proc files and their statistical output is described. One file – namely `/proc/net/nf-hipac/info` – can also be written to, in order to:

- dynamically change the memory bound:
  ‘`echo` *n* `> /proc/net/nf-hipac/info`‘ sets the memory bound to *n* MB.
- dynamically change the invocation order of the iptables filter table and HiPAC:
  ‘`echo nf-hipac-first > /proc/net/nf-hipac/info`‘ prioritizes HiPAC over iptables while
  ‘`echo iptables-first > /proc/net/nf-hipac/info`‘ does the opposite.

TYPE	DESCRIPTION
/proc/net/nf-hipac/info	
scalar	maximum memory bound
scalar	total requested memory (in bytes)
scalar	total allocated memory (in bytes)
scalar	number of built-in and user-defined chains
scalar	total number of rules (in all chains)
description	nf-hipac invoked before iptables or vice versa
/proc/net/nf-hipac/statistics/mem	
scalar	total requested memory (as in `/proc/net/nf-hipac/info`)
scalar	total allocated memory (as in `/proc/net/nf-hipac/info`)
scalar	number of allocated memory blocks (= number of dynamic objects)
scalar	number of buckets in memory hash (hash table used to maintain the addresses of dynamic objects and the amount of memory requested for each object)
scalar	number of entries in the smallest bucket of the memory hash
scalar	number of entries in largest bucket of the memory hash
distribution	$R \rightarrow$ number of memory hash buckets with k entries where $k \in R$
/proc/net/nf-hipac/statistics/rlp_{input,forward,output}	
description	built-in chain associated with the proc file
scalar	total requested memory (as in `/proc/net/nf-hipac/info`)
scalar	total allocated memory (as in `/proc/net/nf-hipac/info`)
scalar	requested memory for rlp objects (in bytes)
scalar	allocated memory for rlp objects (in bytes)
scalar	number of rlp objects
mapping	$i \rightarrow$ number of rlp objects with dimension id i
mapping	$i \rightarrow$ number of rlp objects in depth i
scalar	number of rlp keys (in all rlp objects)
distribution	for each dimension id i: $R \rightarrow$ number of rlp objects with dimension id i containing k keys where $k \in R$
scalar	number of leaves
mapping	$i \rightarrow$ number of leaves whose parent rlp object has dimension id i
mapping	$i \rightarrow$ number of leaves whose parent rlp object is in depth i
mapping	$i \rightarrow$ number of rlp objects whose parent rlp object has dimension id i
mapping	$i \rightarrow$ number of rlp objects whose parent rlp object is in depth i
scalar	number of rule blocks (leaves containing more than one rule)
scalar	number of rules in all rule blocks
distribution	$R \rightarrow$ number of rule blocks with k rules where $k \in R$
/proc/net/nf-hipac/statistics/dimtree_{input,forward,output}	
description	MRLP chain (one per built-in chain) associated with the proc file [this section refers to the rules of the corresponding MRLP chain]
scalar	requested memory for MRLP chain including rules
scalar	allocated memory for MRLP chain including rules
scalar	number of rules
scalar	number of rules with iptables matches
scalar	number of rules with iptables target

TYPE	DESCRIPTION
distribution	$R \rightarrow$ number of *equal priority rule subsets s* containing k_s rules where $k_s \in R$ [for each priority p, there is an *equal priority rule subset* which contains all rules with priority p; these rules are pairwise non-overlapping and created during the conversion of rules containing negated native matches (cf. section 3.3.3)]
mapping	$i \rightarrow$ number of rules with i native matches
/proc/net/nf-hipac/statistics/hipac_rules_{input,forward,output}	
description	built-in chain associated with the proc file [this section refers to the rules of the corresponding built-in chain c and all user-defined chains contained in the DAG of chains reachable from c]
scalar	number of rules
scalar	number of rules with iptables matches
scalar	number of rules with iptables target
scalar	number of jump rules
scalar	number of rules with action `IPT_RETURN`
mapping	$i \rightarrow$ number of rules with i native matches
mapping	$i \rightarrow$ number of rules with i negated native matches
/proc/net/nf-hipac/statistics/hipac_chains	
scalar	requested memory for all built-in and user-defined chains including rules
scalar	allocated memory for all built-in and user-defined chains including rules
scalar	number of built-in and user-defined chains
scalar	number of rules in all chains
distribution	$R \rightarrow$ number of chains where each chain c is referred to by k_c rules where $k_c \in R$
distribution	$R \rightarrow$ number of chains with k parent chains where $k \in R$
distribution	$R \rightarrow$ number of chains with k child chains where $k \in R$

Table 3.2: Statistical information contained in each proc file.

3.2.4 Interface string match emulation

The linux representation of network interfaces stores both the interface name (string) and a number (interface index) which is unique for all interfaces. Note that the interface index is not fixed for a certain name, e.g. if the interface is renamed, the index remains unchanged whereas if a registered network device is unregistered and later registered again with the same name, it is not guaranteed that its interface index remains unchanged.

The iptables in / out interface match is based on strings (at most 16 byte including the terminating '\0'). The interface index is ignored and instead, iptables performs a string compare for each interface match being processed per packet. It is also possible to use strings which don't refer to an existing network interface.

To support the iptables interface match semantics with HiPAC, the interface string emulation (`nfhp_dev.[ch]`) associates a unique number (virtual index) with the name of each IPv4 network interface and with the name used in each interface string match inserted during the lifetime of the `nf_hipac` kernel module. The bijective mapping is stored in two separate arrays, one sorted after the virtual index and the other sorted after the interface name. Mappings are queried using simple binary search on the corresponding array. Alternatively, two hashes could have been used but binary search scales well enough for the number of interface matches used in real world rule sets. Moreover, both arrays are not used during packet classification. For this purpose, another array is maintained which maps the interface index of each active IPv4 interface (interface in state `IFF_UP`) to the corresponding virtual index. Since the number of active interfaces is usually rather small, it is advantageous to use binary search too instead of a hash table.

On module loading time, all present IPv4 interfaces are assigned virtual indices and for each interface, a mapping of its interface index to the corresponding virtual index is created. During the lifetime of the `nf_hipac` kernel module, interface states may change, e.g. active interfaces may become inactive and vice versa. These events are communicated by so-called notifiers. In order to keep the arrays up-to-date, the following events are relevant:

`NETDEV_UP`: Inactive network interface becomes active.
⇒ Assign new virtual index if interface name appears for the first time. Insert mapping of interface index to virtual index.

`NETDEV_DOWN`: Active network interface becomes inactive.
⇒ Remove mapping of interface index to virtual index.

`NETDEV_CHANGENAME`: Network interface name changes.
⇒ Assign new virtual index if new interface name appears for the first time. Update mapping of interface index (same as before renaming) to virtual index.

Unfortunately, linux notifiers are passive, i.e. it is not possible to prevent a notified event from happening. Since this functionality is required because array updates may fail (memory shortage), the abstract network device layer (`net/ipv4/core/dev.c`) is extended by a simple veto notifier. Two functions (`register_netdevice_veto()`, `unregister_netdevice_veto()`) are added which allow the registration resp. deregistration of a single callback function. This function is called on one of the events mentioned above. If it returns a value unequal to 0, the event is canceled. Note that `NETDEV_DOWN` events cannot be canceled due to design limitations, i.e. the operation is implicitly assumed not to fail which results in the return value of `dev_close()` being ignored.

The initialization of the interface string match emulation performs two tasks. It registers a veto callback function and iterates over the global network interface list to setup the mappings. Since the mappings cannot be initialized atomically, the veto callback function must be registered at first to not miss a relevant event. Thus, it is possible

(on a SMP system) that the callback function is issued before the mappings are completely initialized. This race condition is resolved by a spinlock which is only used during initialization. There are also other locks necessary to protect the functions updating the mappings. Those are not discussed here. Note that the interface index to virtual index mapping may be accessed during the packet classification operation without locking because the critical parts of the array update are protected by the `BR_HIPAC_LOCK` write lock.

The interface string match emulation offers a simple, yet efficient way to support the iptables interface match semantics. However, iptables offers an extension of the string match which is not supported by HiPAC: interface string matches ending with the '+' character apply to all interface names beginning with the string excluding the '+', e.g. "ppp" and "ppp123" apply to "ppp+". This restriction can be easily worked around by specifying the corresponding interfaces individually. Note that this approach may involve rule set updates which are otherwise not necessary.

3.2.5 User space integration

As stated in section 3.2.1, the user space part of HiPAC consists of the command line tool nf-hipac and the library libnfhipac. Nf-hipac is almost syntax compatible to the iptables user space tool which is also called iptables. One noticeable incompatibility is that nf-hipac does not support non-contiguous IP network masks, e.g. 192.168.0.0/0.0.0.1. Instead, prefix network masks or IP ranges may be used. The command line options of nf-hipac are shown after executing `nf-hipac -h` in a shell. For further information, the iptables documentation can be consulted, e.g. [RBM+99], [And01]. The motivation behind the compatibility to iptables is to facilitate the migration to HiPAC and hide the complexity of the algorithm behind a well-known established user interface.

The lightweight library libnfhipac, encompassing the files `libnfhipac.[ch]`, hides the netlink based communication protocol behind a simple interface and translates the error codes of the status message into comprehensible descriptions. The main function `nlhp_send_cmd()` implements the whole protocol, i.e. it opens a netlink socket and sends the given command message to the kernel. If a listing is requested, a callback function must be passed to `nlhp_send_cmd()` which calls it once for each received listing packet. Note that the library does not provide any means to create or verify the command message. It relies upon the message being correctly constructed by the caller and the sanity checking implemented by the kernel module. However, future libnfhipac versions will probably provide functions to construct the command message in order to encourage the implementation of alternative front-ends.

Nf-hipac, encompassing the single file `nf-hipac.c`, parses the command line options and constructs the command message which is then passed to `nlhp_send_cmd()`. Moreover, it implements the listing callback function which directly outputs the content of each packet as it is received without buffering. Regarding the generic support for iptables matches and targets, nf-hipac is not only syntax compatible to iptables but also source compatible. Each iptables match / target consists of a kernel and a user space

part. The kernel space part is implemented as kernel module while the user space part is implemented as dynamically loadable library (DLL) which can also be statically compiled into the iptables binary. Each DLL exports a set of functions mainly used to output a help text, parse the command line options specific to the match / target and output a textual representation of the match / target as part of the listing. Generally, the DLL's are not fully self-contained in that they use a set of functions exported by the iptables binary. To be fully source compatible to iptables, these functions are also implemented by nf-hipac.

3.3 HiPAC core

This section generalizes the packet classification problem from chapter 2 to the non-terminal packet classification problem which reflects the extensions provided by iptables. Afterwards, the design of the HiPAC core is presented along with a number of implementation aspects.

3.3.1 Non-terminal packet classification problem (NPCP)

In section 3.1.4, the extensions are described which render iptables more expressive than PCP. These extensions motivate the formulation of a generalized version of PCP (definition 2.6) which is developed in this section. The foundation of the new problem is the extension of the match part and action of a rule. In addition to range based matches, which are henceforth called **native matches**, a rule may contain an arbitrary number of **procedural matches** which must be evaluated at runtime for each packet. Concerning the action, a rule may either contain a **terminal action**, which corresponds to the action type used with PCP, or a **non-terminal action**. Hence, a d-dimensional rule is specified by its priority, d native matches, an arbitrary number (possibly 0) of procedural matches and either a terminal or non-terminal action. One distinguishes two types of rules: terminal and non-terminal rules which are defined as follows.

Definition 3.1 *A **terminal rule** R is a rule with terminal action and 0 procedural matches. This property is denoted by $term(R)$.*

Definition 3.2 *A **non-terminal rule** R is a rule which is not terminal, i.e. R contains a non-terminal action or at least one procedural match or both.*

A policy rule is a terminal rule which is defined as in definition 2.4. The definition of the rule set remains unchanged, i.e. a set of d-dimensional rules including a policy rule with pairwise disjoint priorities. Thus, the generalized packet classification problem is defined as follows.

Definition 3.3 *Given a d-dimensional rule set $\mathcal{R}$ and a packet P with d packet fields, the d-dimensional **non-terminal packet classification problem (NPCP)** returns the smallest priority, terminal rule R^T in $\mathcal{R}$ which applies to P and the set of non-terminal rules*

$R_1^{\mathrm{N}}, \ldots, R_k^{\mathrm{N}}$, $k \geq 0$ *with lower priority than* R^{T} *which also apply to P. That is:*

$$NPCP(\mathcal{R}, P) = \{R_1^{\mathrm{N}}, \ldots, R_k^{\mathrm{N}}, R^{\mathrm{T}}\},$$
$$prio(R^{\mathrm{T}}) = min\{prio(R'),\ R' \in \mathcal{R} \mid term(R') \wedge apply(R', P)\} \wedge$$
$$\forall 1 \leq i \leq k:\ prio(R_i^{\mathrm{N}}) < prio(R^{\mathrm{T}}) \wedge apply(R_i^{\mathrm{N}}, P)$$

Note that $apply(R, P)$ still obeys definition 2.3, i.e. it refers only to the native matches of R. Considering the result of NPCP, it is necessary to linearly evaluate the R_i^{N} at runtime for each packet in descending order of their priorities. If a non-terminal rule with terminal action matches a packet, the evaluation stops and the action is executed.

Similar to the reduction of PCP to MRLP, it is possible to reduce NPCP to MRLP. The only difference is that a leaf of the MRLP tree consists of a set of rules instead of a single rule. For each leaf, this set of rules is equal to the solution of NPCP for the packet representing the path leading to the leaf. This modification is formally expressed by extending the definition of $\mathcal{A}$ in claim 2.3:

Let $\mathcal{A}$ *be a total function which maps a path of length d to the set containing the smallest priority, terminal rule matching the path and the non-terminal rules with lower priority than the terminal rule which also match the path.*

$$\mathcal{A} : \{K \in \mathcal{U}^d \mid K^{[1:d-1]} \in dom(M_{d-1}) \ \wedge\ K(d) \in M_{d-1}(K^{[1:d-1]})\} \rightarrow \{\mathcal{R}' \subseteq \mathcal{R}\},$$
$$\mathcal{A}(K) = \{R_1^{\mathrm{N}}, \ldots, R_k^{\mathrm{N}}, R^{\mathrm{T}}\},\ prio(R^{\mathrm{T}}) = min\{prio(R'),\ R' \in O_d(K) \mid term(R')\} \wedge$$
$$\forall 1 \leq i \leq k:\ R_i^{\mathrm{N}} \in O_d(K) \wedge prio(R_i^{\mathrm{N}}) < prio(R^{\mathrm{T}})$$

Using the same reasoning as in proof 2.3, one can show that the construction yields: $NPCP(\mathcal{R}, P) = \mathcal{A}(MRLP(\mathcal{M}, P))$.

Concerning the implementation, the set of rules contained in a leaf is represented by an array of pointers to the corresponding rules. The non-terminal rules are stored in descending order of their priorities while the terminal rule is stored first to avoid an additional memory access if a rule is inserted which is not included in the array or a rule is deleted which is not contained in the array. These conditions are determined by comparing the priority of the rule with the priority of the terminal rule in the array. In general, the insertion resp. deletion of a rule into resp. from a leaf yields the following result:

Insert the rule R into the leaf $L := \{R_1^{\mathrm{N}}, \ldots, R_k^{\mathrm{N}}, R^{\mathrm{T}}\}$:
If $prio(R) > prio(R^{\mathrm{T}})$, the leaf remains unchanged. Otherwise, if R is terminal, the leaf becomes $\{R_1^{\mathrm{N}}, \ldots, R_j^{\mathrm{N}}, R\}$, $\forall 1 \leq i \leq j \leq k:\ prio(R_i^{\mathrm{N}}) < prio(R)$ and if R is non-terminal, the leaf becomes $L \cup \{R\}$.

Delete the rule R from the leaf $L := \{R_1^{\mathrm{N}}, \ldots, R_k^{\mathrm{N}}, R^{\mathrm{T}}\}$ where P is the packet representing the path to the leaf:
If $prio(R) > prio(R^{\mathrm{T}})$, the leaf remains unchanged since $R \notin L$. Otherwise, if $R = R^{\mathrm{T}}$, the leaf becomes $L' := NPCP(\mathcal{R} \backslash \{R^{\mathrm{T}}\}, P)$ where $L \cap L' = \{R_1^{\mathrm{N}}, \ldots, R_k^{\mathrm{N}}\}$ and if R is non-terminal, the leaf becomes $L \backslash \{R\}$.

Both insert and delete require a lookup on the part of the array containing the non-terminal rules which can be implemented using binary search. Note that L' may be naively computed by iterating over the rule set similarly to function `MRLP.get_rule`

(section 2.6). To avoid repeatedly iterating over the whole rule set, a set of candidates may be precomputed which contains all rules overlapping the rule to be deleted (cf. section 2.7).

For performance reasons, HiPAC omits the array of rule pointers if it consists solely of the terminal rule. In this case, the key of the parent rlp referring to the leaf directly points to the terminal rule in order to save one memory access.

3.3.2 Design overview

Figure 3.6 illustrates the components of the HiPAC core and their dependencies. The main constituents are the RLP, MRLP and chain layer which are built on top of each other. The chain layer implements the HiPAC core API which serves as external interface. The MRLP layer implements the dynamic operations on the tree including the classification function and the RLP layer implements an RLP solving data structure representing the nodes of the MRLP tree. The following sections provide an overview of the core components.

Kernel space / user space compatibility

The compatibility header `mode.h` allows HiPAC to be used both as part of the `nf_hipac` kernel module and as user space library. The latter enables the verification of the HiPAC core in user space which is described in chapter 4. The implementation encompasses wrappers for memory allocators, locks, output functions and certain information about SMP systems.

As for memory allocation, HiPAC uses `malloc()` if compiled as library and `kmalloc()` resp. `vmalloc()` if compiled as part of the kernel module. `kmalloc()` is a slab cache allocator which is implemented according to [Bon94]. It returns physically contiguous memory where the allocated block size is doubled starting from 32 byte (if the page size is 4 KB) resp. 64 byte (otherwise) up to at most 128 KB. Larger memory blocks may be allocated via `vmalloc()` which returns virtually contiguous memory which is not necessarily physically contiguous. The size of the virtual memory block is a multiple of the page size (4 KB on x86).

Concerning locks, the kernel version requires the big reader lock `BR_HIPAC_LOCK` and spinlocks. The former is used as efficient replacement for standard read-write locks to synchronize the classification operation with tree modifications. The spinlock is required within the scope of the atomic counter snapshot. The user space version does not need any locks since it is assumed to run in a concurrency free environment.

The output function of the user space version is `printf()` while the kernel version uses `printk()` which is mostly compatible to `printf()`.

On SMP systems, the number of cpus and the id of the currently running cpu is accessible which is relevant for the per cpu rule counters. The user space version assumes a single cpu.

Incremental hash

The incremental hash implements hashing with chaining using a single hash function per key type. The instantiation of a hash object requires an equality test function and a hash function so that it is possible to support arbitrary key types. Predefined equality and hash functions are available for strings and 32 / 64 bit integers. As for the operations, insertion, deletion, replacement of key / object pairs and key lookup is implemented. The name "incremental hash" is chosen because the number of buckets is doubled each time the average number of elements per bucket exceeds a predefined maximum which involves rehashing of all elements contained in the hash. Note that the number of buckets is never decreased.

To evaluate the distribution of keys, a number of statistical information may be computed including the number of elements and buckets, the number of elements contained in the smallest resp. largest bucket and a distribution (cf. section 3.2.3) which maps a numeric range N to the number of buckets containing $n \in N$ elements.

Common functions

HiPAC offers a dynamically adjustable parameter to limit the maximum memory usage. The motivation of this feature is to avoid system instabilities which are caused by memory shortage, e.g. the linux kernel might kill processes or even freeze if free memory becomes unavailable. The functionality is hidden behind a set of memory allocators based on the allocation primitives provided by the compatibility header. The allocator implements three functions similar to `malloc()`, `realloc()` and `free()`. The addresses of the allocated memory blocks are stored in a hash along with the requested size which is necessary to determine the size of a memory block which is freed. Additionally, the mapping allows the comparison between the amount of requested memory and effectively allocated memory revealing the amount of unused, allocated memory.

Besides the memory allocator, the implementation provides a dynamic array which is automatically resized if necessary and a doubly linked list data structure based on `include/linux/list.h`.

RLP layer

The RLP layer implements a static B+ tree (henceforth called btree) behind an interface hiding the actual implementation which facilitates the integration of alternative RLP solvers. The btree maps the nodes, which are completely filled unlike the nodes of a dynamic btree, to a single memory block. Hence, it is more memory efficient than the dynamic variant but insertion and deletion of a key requires the data structure to be rebuilt. The main motivation behind the static approach is revealed in the following section which presents the MRLP tree as partially persistent data structure. The btree supports 8, 16 and 32 bit keys to keep the nodes as small as possible. Considering the MRLP algorithm, at most two keys are added per node during insertion of a rule and

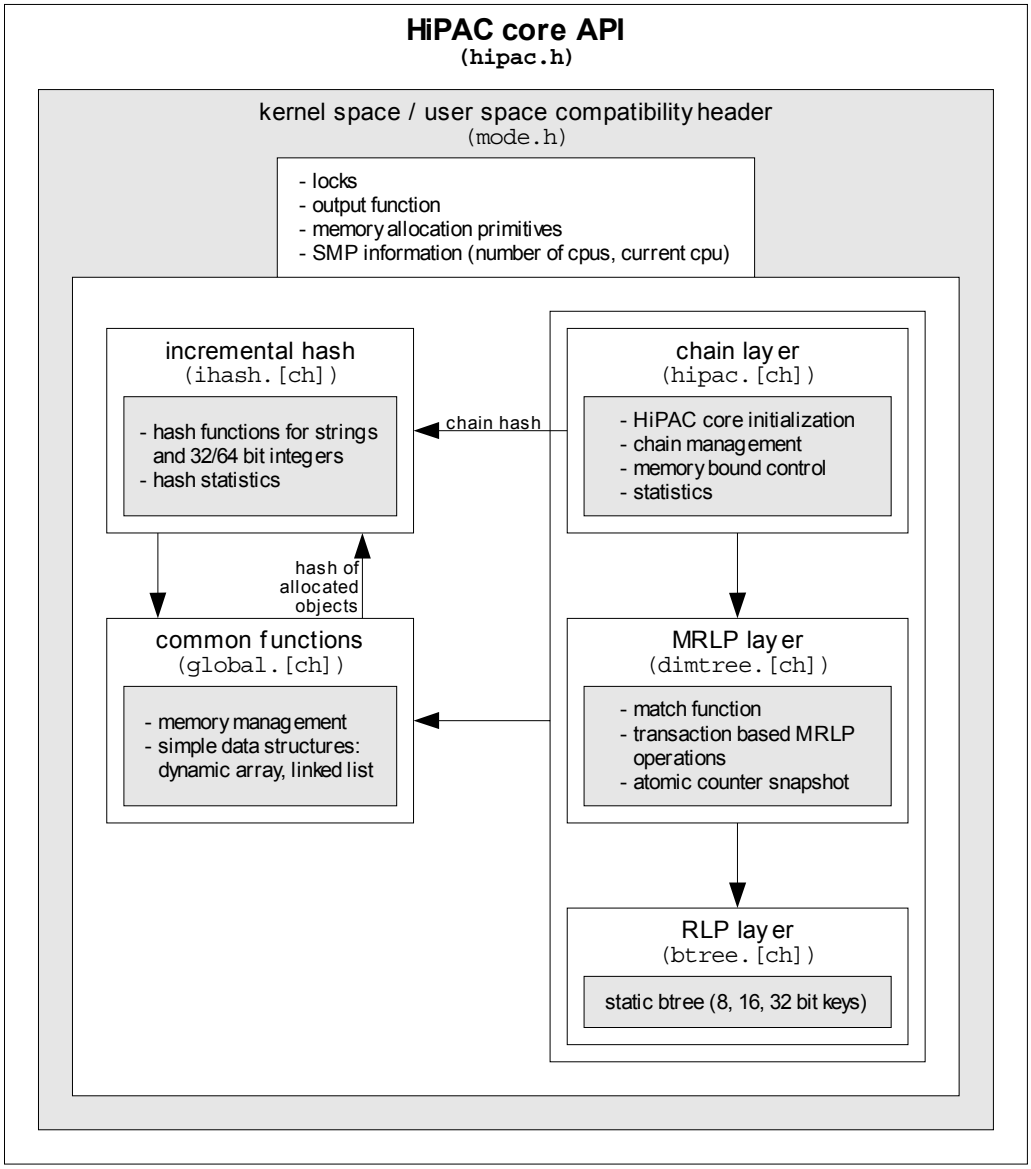

Figure 3.6: Overview of the HiPAC core design.

at most two keys are removed per node during deletion of a rule. This is reflected by the RLP insert and delete function which allows at most two keys to be added resp. removed in a single operation.

MRLP layer

The MRLP layer is responsible for managing NPCP rule sets. Each rule set is represented by the corresponding MRLP tree and a list of rules sorted after the rule priorities in ascending order. Besides insertion and deletion of rules, the MRLP layer implements a way to compute an atomic counter snapshot and the match function which performs the packet classification. To allow tree updates and packet classification to be performed simultaneously, the MRLP tree is maintained as partially persistent data structure, i.e. a data structure which preserves two versions where only the latest version may be updated. Hence, the match function operates on the original tree (version before update) while insert and delete operations build a new tree. Instead of copying the whole tree, insert and delete operate on the original tree and create a new node each time a node of the original tree is modified. Thus, both trees share all nodes which are not affected by the modification. This approach explains why a static data structure is used as RLP solver since an RLP update requires the node to be rebuilt anyway [9].

The insert and delete functions are embedded in a transaction based interface which allows an arbitrary number of rule set updates to be atomically committed. The updates apply each to the latest version of the tree. Nodes which are not contained in the original tree may be directly replaced on an update. At the end of a series of insert and delete operations, the new tree is made available for packet classification by setting the root pointer from the root node of the original tree to the new root node.

Since an update may fail due to memory shortage, it is necessary to implement a recovery mechanism which enables fallback to the original tree while the modifications are discarded. Section 3.3.3 outlines the implementation behind.

HiPAC core API (chain layer)

The chain layer implements the API offered by the HiPAC core which essentially consists of a set of functions implementing the netlink commands stated in table 3.1 (section 3.2.2). Moreover, the API encompasses the MRLP match function, a number of functions computing the statistical information stated in table 3.2 (section 3.2.3) and it allows runtime modification of the memory bound limiting the maximum memory usage. The layer manages a set of built-in and user-defined chains which obey the iptables rule set semantics (cf. section 3.1). Each built-in chain along with the reachable user-defined chains are dynamically converted into an equivalent MRLP rule set which involves a number of rules being atomically inserted resp. deleted via the MRLP transaction interface. Disregarding the conversion of user-defined chains which is covered in [Bel04],

[9] Alternatively, a dynamic, partially persistent RLP data structure could be used. However, static RLP data structures are favored because in practice most MRLP nodes are quite small and thus updates are fast. Moreover, static data structures are more memory efficient.

the MRLP rules are equal to the rules of the corresponding built-in chain apart from the rules containing negated native matches. These must be converted into an equivalent set of rules without negation which is described in section 3.3.3. The structure of a rule is the same as the representation of a rule contained in a netlink listing message (cf. figure 3.5 in section 3.2.2).

Before the HiPAC API may be used, an initialization function must be called which defines certain PCP parameters and provides the core with functions accessing OS specific data structures. The following parameters must be supplied:

- number of dimensions (matches)
- mapping which associates each dimension with the bit width of the corresponding packet field (required for the RLP layer), e.g. source IP dimension → 32 bit, TCP source port dimension → 16 bit, transport protocol dimension → 8 bit
- extractor function for each dimension which returns the value of the corresponding packet field
- constructor / destructor function which is called before a rule will be inserted resp. after it has been deleted
- procedural match / target executor function which evaluates a procedural match / target on a given packet
- equality test function which compares two rules (required for deletion of rules according to the netlink command `CMD_DELETE_RULE`)
- initial memory bound limiting the maximum memory usage

Since the HiPAC core is designed to cope with runtime environments involving concurrency, it must be specified which API functions are allowed run at the same time. For this, there are two rules:

1. There may be at most one API function running at a given time excluding the match function.
2. The match function may be called anytime.

Hence, an arbitrary number of classification operations and at most another single API function are processed simultaneously at a given time.

3.3.3 Implementation aspects

This section covers a number of implementation aspects concerning the HiPAC core including MRLP tree recovery, concurrency issues, atomic counter snapshot and negation of native matches.

MRLP tree recovery

Considering a series of insert and delete operations performed on a MRLP tree, two cases are possible. Either the operation is successful and the original tree is replaced by the new one or the series fails due to memory shortage which results in a fallback to the original tree representing the state before the first operation of the series. The MRLP layer implements a history data structure which is capable to handle both situations. The history is represented by a list which stores the nodes of the original tree which are no longer present in the new tree and a hash which stores the nodes which are newly introduced during the series. Considering the dynamic operations, the list and hash are updated according to the following rules:

- Each node of the original tree which is modified, i.e. replaced by a new node, is added to the list while the new node is added to the hash.
- Each node of the new tree which is modified, i.e. replaced by a new node, is freed and replaced in the hash by the new node.
- Each node which is newly created within the scope of cloning a subtree is added to the hash.
- Each node of the original tree which is removed during deletion of a subtree is added to the list.
- Each node of the new tree which is removed during deletion of a subtree is freed and deleted from the hash.

If a series is successful, the nodes contained in the list are freed. Otherwise, if a series fails, the nodes contained in the hash are freed. Note that the fallback to the original tree must be carefully implemented since the recovery itself must not fail under any circumstances. Otherwise, the data structure would become unusable.

Locking within the HiPAC core

Disregarding the match function, the HiPAC core is free of concurrency since API function calls must be serialized externally. The match function introduces concurrency since it may be called anytime and particularly, it may be running simultaneously on multiple cpus. Parallel classification operations don't interfere with each other since the shared MRLP tree on which they operate is accessed read-only. An exception is the counter update which is discussed in the following paragraph. The remaining case is that the match function runs at the same time as another API function modifying the MRLP tree. Due to the tree being implemented as partially persistent data structure, packet classification may operate on the original tree during the update without any locking. After the tree update has been completed, the pointer to the root node (root pointer) is set to reference the new tree and the unreferenced nodes of the original tree

are removed. This causes a problematic race since there might be an ongoing match operation on the original tree while at the same time some nodes of this tree are removed. This situation occurs if the match operation starts before the root pointer is adjusted.

The race is resolved by protecting the critical parts with a read-write lock which allows multiple readers at the same time but only a single writer. Thus, the match function acquires the read lock before the first tree access and releases the lock immediately before the function returns. The tree update works as follows:

1. Update partially persistent tree.
2. Acquire write lock.
3. Set root pointer to new root node.
4. Release write lock.
5. Delete unreferenced nodes of original tree.

Hence, the critical section solely consists of a single value assignment which renders packet classification virtually unaffected by tree updates. Note that instead of a traditional read-write lock, HiPAC uses the big reader lock `BR_HIPAC_LOCK` which achieves higher performance for the readers at the cost of penalizing the writer.

In fact, it is even possible to omit the lock if the root pointer modification is followed a sleep period which lasts until the final match operation started before the assignment is finished.

Atomic counter snapshot

As described in section 3.1.3, iptables uses a naive approach to compute atomic counter snapshots, namely by halting the packet classification during the snapshot. HiPAC implements a simple trick which allows packet classification and atomic counter snapshot to run simultaneously. Each rule contains **two** 64 bit counter pairs (packet and byte counter) and according to a global flag, which is defined per MRLP tree, the match function updates either the first or the second counter pair. The computation of an atomic counter snapshot works as follows:

1. Set flag such that the second counter pair is updated instead of the first one.
2. Read atomic counter snapshot represented by the first counter pair of each rule.
3. Set flag such that the first counter pair is updated again.
4. For each rule: add second counter pair to first counter pair and set second counter pair to 0.

Note that step 4 involves a race since it might happen (though very unlikely) that the first counter pair is updated by both the match function and due to the addition of the second counter pair. Since 64 bit addition is not atomic on most architectures, a spinlock

is used to synchronize the race. On uni-processor systems this lock is omitted because it is ensured that the critical addition is not interrupted.

On SMP systems, there is a separate counter unit (first and second counter pair) for each cpu to allow parallel counter updates. If only a single counter unit were used, the counter update of a single rule by n cpus would be serialized by the spinlock. The counter units are arranged such that each occupies a separate cache line in order to avoid cache thrashing during the counter update.

Negated native matches

The native matches of the chain layer differ from the ones of the MRLP layer in that they may be negated. Hence, a chain layer rule containing negated native matches must be converted into an equivalent set of MRLP rules. Clearly, a single, negated range match $\neg[a,b]$ can be expressed by at most two ranges, namely $[0,a-1]$ if $a > 0$ and $[b+1, max(\mathcal{U})]$ if $b < max(\mathcal{U})$. To convert a rule with more than one negated match, it is necessary to compute the set of "positive" rules representing all possible combinations of the equivalent matches without negation, e.g. a rule with match part $(\neg[a,b], \neg[c,d])$, $0 < a \leq b < max(\mathcal{U})$, $0 < c \leq d < max(\mathcal{U})$ is converted into the set of rules whose match parts are as follows:

$$\begin{aligned}\{&([0,a-1],[0,c-1]),\\ &([0,a-1],[d+1,max(\mathcal{U})]),\\ &([b+1,max(\mathcal{U})],[0,c-1]),\\ &([b+1,max(\mathcal{U})],\ [d+1,max(\mathcal{U})])\}\end{aligned}$$

Note that the rules are pairwise non-overlapping, i.e. there is no packet which matches two converted rules. In general, the conversion of a rule containing i negated matches yields at most 2^i positive rules. In practice, this is not a problem since the number of negations per rule is quite small.

4 Verification of the HiPAC core

Software verification or testing is an essential part of software development. Different approaches are possible ranging from single test cases to formal verification methods. As for HiPAC, verification plays an important role since the software permanently runs in kernel space. Hence, programming errors could lead to memory leaks[1] and even system crashes. The main verification goals are to ensure both correctness and robustness. Here, the latter means that HiPAC has to cope with lack of system resources or runtime faults without entering an inconsistent state. Robustness particularly includes error recovery, e.g. if a rule set update cannot be finished due to memory shortage HiPAC has to undo all modifications and return to the state before the update. Clearly, to achieve the verification goals it is not feasible to manually construct test cases and debug them. This chapter describes a simple generic test suite which has been implemented to meet the goals accompanying the development process. It is used to configure and run verification modules implemented as dynamically loadable libraries. The verification modules randomly generate a test case for a certain HiPAC component and perform a number of checks for each operation. Subject to the verification is only the HiPAC core which runs both in kernel and user space. The HiPAC front-end is considered simple enough to be manually tested.

Since both HiPAC core and verification suite are implemented in C, runtime errors and undefined behavior may occur as a result of programming faults. These errors don't necessarily lead to a program crash which can be easily ascribed to the responsible lines of code. In fact, the program may not crash at all and still contains subtle bugs like reading undefined values which are accidentally suitable for the test case. To achieve reliable results, the verification sketched above must be executed in an environment which is able to detect these illegal conditions. In the beginning of 2002, an excellent tool has been released – **valgrind** – which perfectly meets this requirement. Its benefits are outlined in the following paragraph.

Valgrind [Sew02] is a very flexible tool for debugging and profiling linux x86 executables. It implements a x86 to x86 JIT (just-in-time) compiler and a virtual x86 cpu which executes the code and detects the use of illegal memory addresses and undefined values. All read and write operations to the memory are checked. In particular, the following programming errors are detected.

- use of uninitialized memory

[1] Memory leaks are allocated memory blocks which are no longer referenced so that it is impossible to deallocate them.

- reading / writing memory location after it has been freed
- reading / writing off the end of allocated memory blocks
- reading / writing inappropriate areas on the stack
- memory leaks
- overlapping source and destination memory areas in `memcpy` and related functions

Valgrind is independent from the programming language and requires no modifications of the binary like recompiling or relinking. However, the executable must be dynamically linked since valgrind relies on the dynamic linking mechanism to gain control at startup. If the executable is compiled with debugging information, valgrind shows the file and line of code responsible for an error. It is recommended to disable compiler optimizations to avoid rare situations causing valgrind to wrongly report uninitialized value errors.

4.1 Design of the verification suite

Figure 4.1 illustrates the design of the verification suite. It encompasses three main components, the verification manager, the verification modules and the helper libraries. The manager configures, runs and monitors the verification modules. There are two kinds of modules, *active* and *passive* ones. Active modules implement a test case generator whereas passive modules don't. Passive modules are essentially configurable libraries which are used by active modules. The implementation provides a packet and rule generator as passive modules. The dummy module implements the skeleton of a verification module. As for the active modules, there is a test case generator for the incremental hash, the RLP, MRLP and chain layer of the HiPAC core. Each module is described in section 4.2. To allow the verification manager to interact with arbitrary modules, it relies on the common verification module interface (`icdll.h`) which must be implemented by each module. The interface contains only three functions which can be used to configure the module, print the current configuration and run the test case generator.

On startup, the manager parses the config file, loads the mentioned verification modules (dynamically loadable libraries), passes the relevant config information to the appropriate modules and acts as scheduler for the active modules. The test case generators are started as separate processes where the number of processes running at a given time may be limited by the user. If any process dies or a module indicates that the test is unsuccessful, the manager terminates the remaining processes if any and outputs the failure cause. The config file uses a very simple syntax which is described in the following paragraph. It contains the configuration for the modules and the manager. The verification modules offer a set of parameters which can be used to tweak

the test case generator. The manager offers the program options as parameters and additionally the parameter `run` which specifies the active modules being scheduled.

The helper libraries provide a set of common functions used by both the verification manager and the modules. There is a library providing elementary parsing functions, a library with several randomization functions and a library which implements the naive iptables algorithm including user-defined chains. The latter serves as reference implementation for the MRLP and chain layer test case generators.

To run the verification manager along with valgrind, the shell script `verify.sh` can be used. It is a simple wrapper which interprets the output of valgrind and the verification modules and provides some additional features like running the manager in an endless loop and sending error reports via e-mail if necessary.

Config file structure The file consists of a series of white space separated blocks with the following structure.

```
<module>
{
     <parameter 1> = <value 1>;
     [...]
     <parameter n> = <value n>;
}
```

Each block contains the configuration information which is passed to the verification module given by `<module>`. To configure the verification manager, the name `main` must be used. The parameters are strings which refer to module variables controlling the test case generator. The structure of the value depends on the parameter. It can be one of the following.

- unsigned integer, e.g. `123`
- series of white space separated unsigned integers, e.g. `1 2 3 4`
- series of white space separated ranges of unsigned integers, e.g. `[0, 100] [4, 4]`
- series of white space separated pairs where the first component is a string and the second one is an unsigned integer, e.g. `append 100  delete 20`

Both parameter and value are passed to the corresponding module for further processing. Note that module name and parameter must be specified in a single line while the value is allowed to encompass multiple lines. When passing the value to the module, each string of white space characters within the value is replaced by a single space character.

4.2 Verification modules

This section describes the test case generators including packet and rule generator. Both verification modules covering the chain layer are not discussed here. Their main purpose is to verify the handling of user-defined chains which is explained in [Bel04].

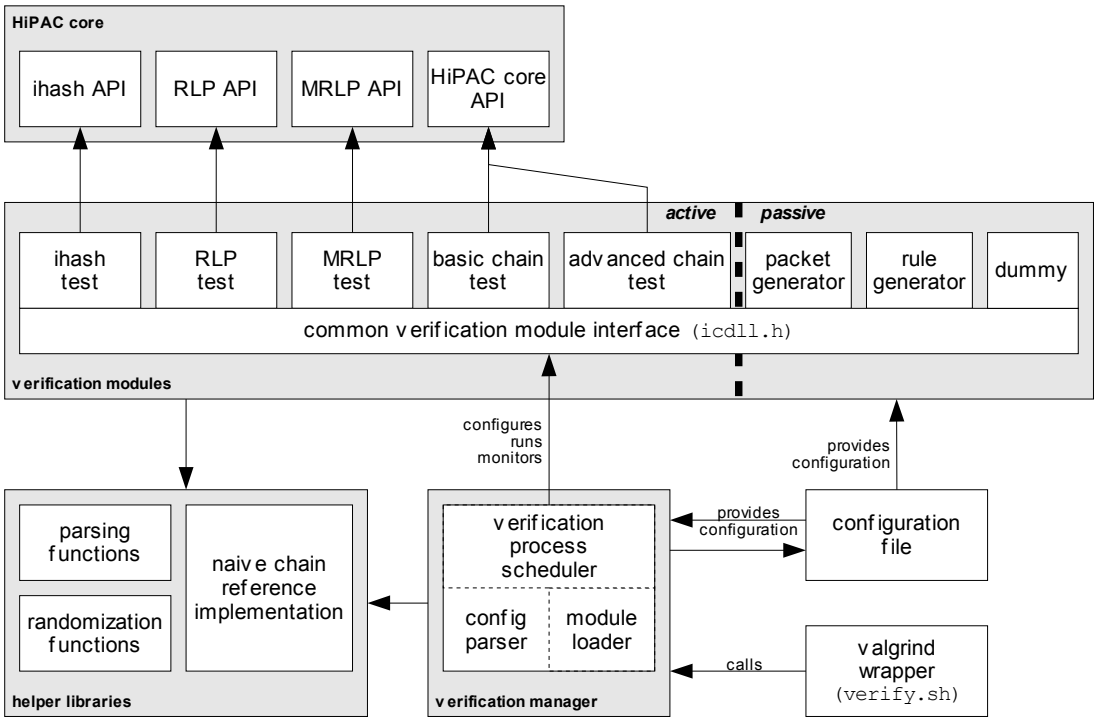

Figure 4.1: Design overview of the verification suite. The arrows which are not labeled state dependencies between components.

A technique commonly used in the test case generators is the so-called **failure simulation**. Its goal is to trigger all error cases for a given operation in order to maximize the code coverage. Since virtually all errors are caused by memory shortage (if not due to invalid function arguments), failure simulation is simply implemented by setting the memory bound, which limits the maximum memory usage, to the current memory usage and afterwards increasing the bound step-by-step while repeatedly calling the function subject to the simulation. Whether or not an operation is subject to the failure simulation is decided at random where the probability is determined by the user.

The methods implemented in the verification modules are summarized in the following sections.

4.2.1 Incremental hash test

The incremental hash test uses unsigned integers and strings representing unsigned integers as keys. The procedure is the same for both key types. The object associated with the key is equal to the key itself. This property is checked at lookup time. The test consists of three phases: insert, replace and delete phase. Each phase is followed by a hash statistics request.

insert: Given a random value r and a number n, insert all keys in $[r, r+n-1]$. Each single insert operation is followed by a number of hash lookups, one for each key contained in the hash. The insert operation may be subject to a failure simulation.

replace: Replace all keys i by $n+i$. Each single replace operation is followed by a check that i is no longer contained in the hash and a number of hash lookups, one for each key contained in the hash. The replace operation may be subject to a failure simulation.

delete: Delete all keys in $[r+n, r+2n-1]$. Each single delete operation is followed by a check that the deleted key is no longer contained in the hash and a number of hash lookups, one for each key contained in the hash. The delete operation may be subject to a failure simulation.

4.2.2 RLP layer test

The RLP layer test uses 8 bit, 16 bit and 32 bit keys. The procedure is the same for all key types. The pointer associated with each key points to an array location containing the key value. This property is checked at lookup time. Since the `rlp` operations are based on random keys, a naive reference implementation (array, linear search) is used to check the lookup results. The test consists of two phases: insert and delete phase. During the former phase a number of random keys is inserted where duplicates are avoided. Each single insert operation is followed by two query procedures. The first performs a lookup on all keys contained in the RLP data structure and the second issues a user-defined number of lookups using random keys. During the delete phase, all keys are removed in random order. Like insert, each single delete operation is followed by

the query procedures. Since the RLP API supports inserting / deleting of at most two keys at the same time, half of the operations insert / delete two keys.

4.2.3 Packet generator

A packet is represented by an array of 32 bit unsigned integers. Given a rule set, the packet generator produces a sequence of packets where a user-defined fraction is chosen to match certain rules and the residual packets are constructed at random. The former are simply built by repeatedly iterating over the rule set and constructing a random packet matching a given rule.

4.2.4 Rule generator

The rule generator implements a simple, yet flexible probabilistic model to construct randomized rules. Compared to a naive rule generator creating totally random rules, this approach has the advantage that certain special cases can be easily configured and HiPAC's space complexity worst case may be avoided. The generator can be used for both chain layer and MRLP rules. The main difference is that the rules of the chain layer support negated native matches and user-defined chain targets.

The generator offers a set of parameters which are used to tweak the construction of the matches (dimensions). Each dimension is treated independent from the others so that it is not possible to specify inter-match dependencies apart from the consistency check (see below). The following parameters are offered per dimension.

`pw[i]:` probability of dimension *i* being a wildcard match

`pp[i]:` probability of dimension *i* being a point match (range containing one element)

`pn[i]:` probability of dimension *i* being a negated match (ignored for MRLP rules)

`part[i]:` number of equally sized ranges dimension *i* is partitioned into

Additionally, there is a probability distribution for network prefix lengths which is represented by the array **`ndist`**. The value of **`ndist[i]`** determines the probability of the prefix length *i*. Note that the user may choose to enable the aforementioned consistency check which ensures that the rule meets some basic inter-match dependencies, e.g. the source / destination port match may only be non-wildcard if the protocol match is non-wildcard. The coherence between the parameters is revealed in function `gen_dim` which describes the construction of a single match given the dimension. The function calls `prob(`*i*`)` which returns `TRUE` with probability *i*.

The rule generator also supports procedural matches and targets. The behavior of such a match or target is determined at rule construction time, i.e. the return value of the match / target is randomly chosen according to a user-defined probability and hardcoded into the rule. The number of procedural matches per rule is determined according to a probability distribution. Thus, there is an upper bound for the number of procedural matches per rule.

Function `gen_dim(`*dim*`)`

```
m ← new match
if prob(pn[dim]) then
    negate m
end
if prob(pw[dim]) then
    m ← wildcard

else if prob(pp[dim]) then
    m ← random point

else if dim is source / destination IP dimension then
    m ← random network subject to ndist

else if part[dim] ≠ 0 then
    m ← random range from partition

else
    m ← random range
end
return m
```

4.2.5 MRLP layer test

The MRLP layer test depends on the rule and packet generator. It consists of two phases: insert and delete phase. In the simplest case, randomized rules are generated and inserted during the first phase. During the second phase, they are deleted again in random order until the rule set is empty. Alternatively to using random rule priorities during insertion, the user may enforce that a new rule is always appended to the chain, i.e. the rule priority is maximal with respect to the rule set excluding the policy rule. A similar option exists for the delete phase which ensures that always the largest priority rule (not the policy rule) is deleted. Each insert / delete operation may be subject to a failure simulation. In terms of the MRLP API, the operations are single transactions, i.e. they are committed immediately. In order to provide an adequate test of the transaction API, the verification module generates a series of operations which are not committed until the final operation of the series is completed. Series are executed according to a user-defined probability during both insert and delete phase. The number of operations in a series is determined by the user. There are three series modes: insert, delete and mixed mode. Like the names suggest, the first one contains only insertions, the second one only deletions and the third one insertions and deletions one half each. Each operation in a series may be subject to a failure simulation which causes the whole series to be canceled. The probability of an insertion / deletion being executed in a failure simulation is defined by the user.

Each operation, whether part of a series or not, is followed by a user-defined number of packet classifications. For this purpose, a set of packets is constructed by the packet generator where each packet is passed to the HiPAC match function and the match function of the naive chain implementation. HiPAC implements a special match function for this end which differs from the “normal” match function in that it collects the matching rules instead of executing their actions. In order to pass the test, both HiPAC and the naive implementation must yield the same set of rules. Additionally, it is checked whether uncommitted insertions change the original tree. Recall that the MRLP layer maintains two versions of the tree, the original one which is subject to packet classification and the new one which contains the uncommitted updates. Both versions share the nodes which are not affected by the updates. In the course of the check, the original tree is traversed after a rule has been inserted and before the operation will be committed. If the rule occurs in the original tree, the tree must have been modified which is not allowed. Unfortunately, a similar check is not possible for an uncommitted delete operation. Instead, it is checked whether the deleted rule still occurs in the tree after the operation has been committed.

5 Performance evaluation

Although packet filters and firewalls in general are widely deployed in today's Internet, there exists no standard performance evaluation tools or techniques to benchmark packet classification systems (PCS). The design of a PCS benchmark involves specifying the desired performance metrics, the test setup, the construction of the rule set and the network traffic. [Bra91] defines a number of terms and metrics used in the context of performance evaluation of general network interconnect devices. Particularly interesting performance metrics for PCS benchmarking are frame loss rate, latency and throughput. The test setup for benchmarking a single PCS generally includes a set of senders and receivers exchanging packets which traverse the PCS. Senders and receivers don't necessarily have to be different physical hosts so that the minimal test setup solely consists of two hosts, the PCS host and another host being sender and receiver at the same time.

Regarding rule set and traffic construction, there are no simple answers. In principle, there are two different approaches to PCS performance evaluation: *algorithm agnostic* and *algorithm aware* benchmarks. The former does not assume any knowledge about the implementation of the PCS and usually involves statistical methods to model rule set and network traffic. The latter uses decided knowledge about the packet classification algorithm and usually attempts to evaluate its worst case performance. However, since the multi-dimensional PCP is an inherently hard problem, all present algorithms either involve high space or time complexity worst cases if more than two dimensions are used. As for HiPAC, it is infeasible to base the evaluation on the space complexity worst case because the data structure grows too fast. Anyway, this test would not be very meaningful in practice since this kind of rule set does not occur in real world applications. In contrast, HiPAC's time complexity worst case is meaningful in practice to give performance guarantees. To construct this worst case, recall that given n rules (including policy rule), each node of the MRLP tree contains at most $2n-1$ keys. Hence, the rule set must be chosen such that the number of keys contained in the nodes which are accessed during packet classification is $O(n)$.

Concerning algorithm agnostic benchmarks, there is very little research. [BM99] centers around general network interconnect devices and describes how the benchmarks should be performed while mainly focussing on traffic patterns. More specialized towards performance evaluation of PCS are [New99], [HNTM03]. The latter presents ten benchmarking tests which are largely based on http requests being exchanged between a set of virtual clients and servers. However, rule set characteristics and precise traffic patterns are not discussed. These topics are addressed by [TT03], a recent benchmark proposal for PCS. The paper presents a technique to generate synthetic rule sets retaining essential statistical characteristics of real world rule sets. The traffic patterns of

the benchmark are tersely sketched. Since PCS are stateless and only concerned with packet headers, it is not necessary to construct the traffic as a mix of valid protocol runs but instead the packet headers are varied such that they apply to different rules. [TT03] proposes phased traffic including two parameters to vary the traffic patterns: **locality** and **frequency** of reference. For each phase, the locality determines the set of possible packet headers in terms of a subset of the geometric space and the frequency determines the average period between packet headers matching the same rule.

The following sections present the structure and results of a simple performance test which serves as preliminary evaluation of HiPAC versus naive PCS represented by iptables. The benchmark is algorithm aware and compares the time complexity worst case of both approaches.

5.1 Test setup

The goal of the performance test comparing HiPAC and iptables is to measure the throughput subject to the packet size and the number of rules. According to [BM99], three of the recommended frame sizes are used: 128 byte, 512 byte and 1518 byte. For each frame size, the number of rules is varied. Starting from 25 rules, the number is doubled until the rule set contains 25600 rules. Each test case is performed using HiPAC and iptables.

The rule set consists of different kinds of rules matching the five tuple source / destination IP, protocol, source / destination port. The rules involve wildcard, range and prefix matches. Table 5.1 summarizes the rule types which are evenly used in the rule set. The random IP prefixes are subject to the probability distribution stated in table 5.2[1]. The rules are chosen such that they don't match the iperf traffic which triggers the worst case of the iptables algorithm. As for HiPAC, the rule set represents the time complexity worst case, i.e. it ensures that the size of the MRLP tree nodes accessed during packet classification scales linearly with the number of rules (apart from the protocol dimension).

The test setup used for performance evaluation is minimal, i.e. two hosts directly connected via ethernet crossover cable. The PCS host, an AMD Thunderbird 1.33 GHz with 512 MB DDR RAM (CL2) and 100 MBit ethernet NIC (VIA Rhine chipset), runs a linux system with kernel version 2.4.18 (optimized for athlon architecture). During the test, the PCS runs only three processes: init, getty and the iperf server. Iperf (`http://dast.nlanr.net/Projects/Iperf/`) is a client/server based network performance measurement tool which is used as packet generator and for throughput measurement. The generated traffic consists of a single TCP stream from client to server. The PCS does not forward the traffic but instead delivers the packets to the local iperf server. Hence, the rule set is installed such that locally destined packets are classified.

[1] The probability distribution does not claim to be representative for real world rule sets.

source IP	**destination IP**	**protocol**	**source port**	**destination port**
random IP	–	–	–	–
random IP prefix	–	–	–	–
–	random IP	–	–	–
–	random IP prefix	–	–	–
client IP	random IP	–	–	–
client IP	random IP prefix	–	–	–
–	–	TCP	random port	–
–	–	TCP	random port range	–
–	–	TCP	–	random port
–	–	TCP	–	random port range
client IP	–	TCP	random port	–
client IP	–	TCP	random port range	–
client IP	–	TCP	–	random port
client IP	–	TCP	–	random port range
–	server IP	TCP	random port	–
–	server IP	TCP	random port range	–
–	server IP	TCP	–	random port
–	server IP	TCP	–	random port range
client IP	server IP	TCP	random port	–
client IP	server IP	TCP	random port range	–
client IP	server IP	TCP	–	random port
client IP	server IP	TCP	–	random port range

Table 5.1: Each row of the table describes a certain rule type. The rule set consists of equal portions of each rule type. *Client IP* refers to the host running the iperf client while *server IP* refers to the host running the iperf server (PCS host).

prefix length	probability
8	5%
12	1%
16	25%
20	6%
22	1%
24	45%
27	1%
28	15%
30	1%

Table 5.2: Probability distribution of the occurrence of different IP prefix lengths.

5.2 Performance results

Figures 5.1, 5.2 and 5.3 illustrate the results of the performance test. Clearly, HiPAC outperforms iptables in all test cases. With 512 byte and 1518 byte sized frames, HiPAC retains maximum performance regardless of the number of rules while iptables causes a performance decrease if more than 300 resp. 800 rules are used. With 128 byte sized frames, HiPAC causes a slight performance decrease which evolves almost linearly while the number of rules is doubled. However, the iptables performance with 50 rules is already lower than the HiPAC performance with 25600 rules. Starting from 3200 rules, iptables renders the system (almost) completely unresponsive. Note that HiPAC does not involve a significant overhead compared to iptables. Otherwise, it would be outperformed by iptables for the smallest rule set.

Regarding the throughput of the test case involving 128 byte sized frames, the discrepance between measured value and theoretically maximum value becomes apparent. The theoretically maximum throughput for 100 MBit ethernet assumes back-to-back frames. The size of each packet in the stream of back-to-back frames is the sum of the frame size plus the size of the ethernet preamble (8 bytes) plus the minimum inter-frame gap (960 nanoseconds for 100 MBit ethernet which corresponds to 12 bytes). Hence, the theoretically maximum throughput for 100 MBit ethernet using 128 byte sized frames is $\frac{100 \cdot 10^6}{8 \cdot (128+8+12)} \simeq 84459$ frames per second. Yet, only a throughput of 56964 frames per second is achieved in the test. This may be due to the system's network parameters set to standard values and not especially tuned for high performance networking.

Obviously, the performance test has several shortcomings which are listed below.

- Instead of TCP, it would be better to use UDP or raw packets to avoid the protocol overhead.
- Rule set generation should be more involved, i.e. it should be based on statistical patterns incorporating inter-match dependencies.
- Network traffic generation should be more involved, i.e. instead of using a single connection, the packet headers should be varied according to a rule set oriented statistical model.
- The PCS should forward the traffic instead of delivering it locally in order to attain the real use case and to avoid the user space overhead.
- Latency and frame loss rate should be measured.
- GBit NIC's should be used instead of 100 MBit ones.
- System network parameters should be tuned for high performance networking.
- It would be interesting to compare the performance of SMP (symmetric multiprocessing) systems with uni-processor systems.

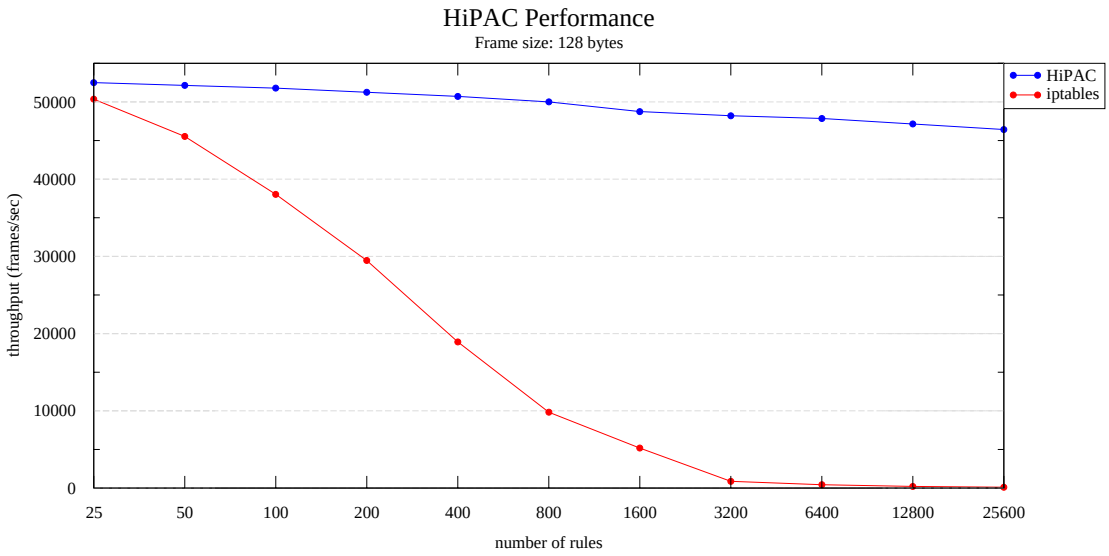

Figure 5.1: Results of the performance test involving 128 byte sized frames. Note the logarithmic scale of the x-axis.

Altogether, the presented performance evaluation gives only a preliminary rating of HiPAC's capabilities compared to naive PCS represented by iptables. More comprehensive tests are necessary to provide meaningful results for practical applications of PCS. It is desirable that the research community intensifies its efforts to establish standard techniques and tools for PCS benchmarking in order to facilitate the development of new algorithms and the optimization of existing packet classifiers.

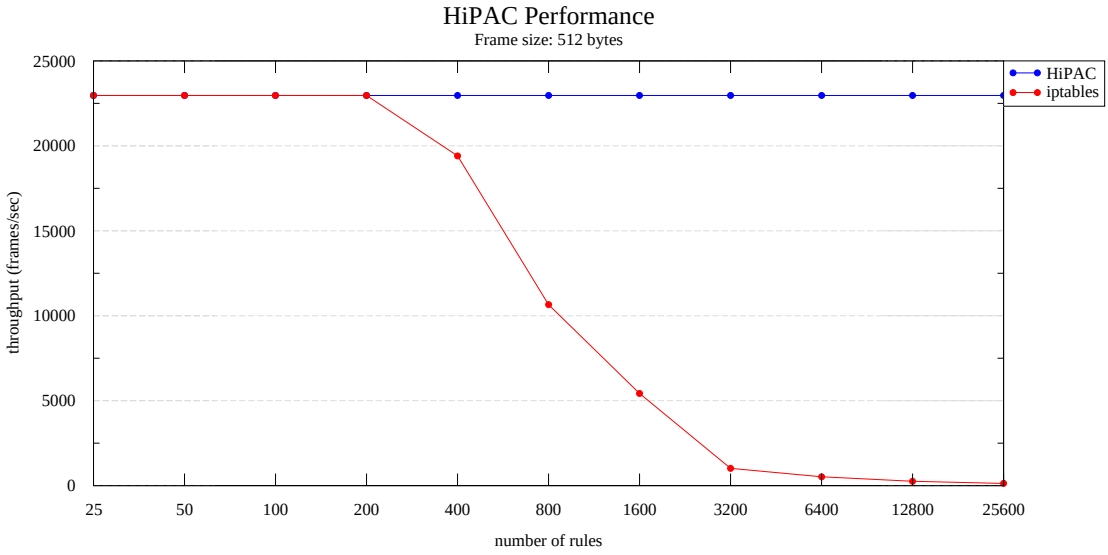

Figure 5.2: Results of the performance test involving 512 byte sized frames. Note the logarithmic scale of the x-axis.

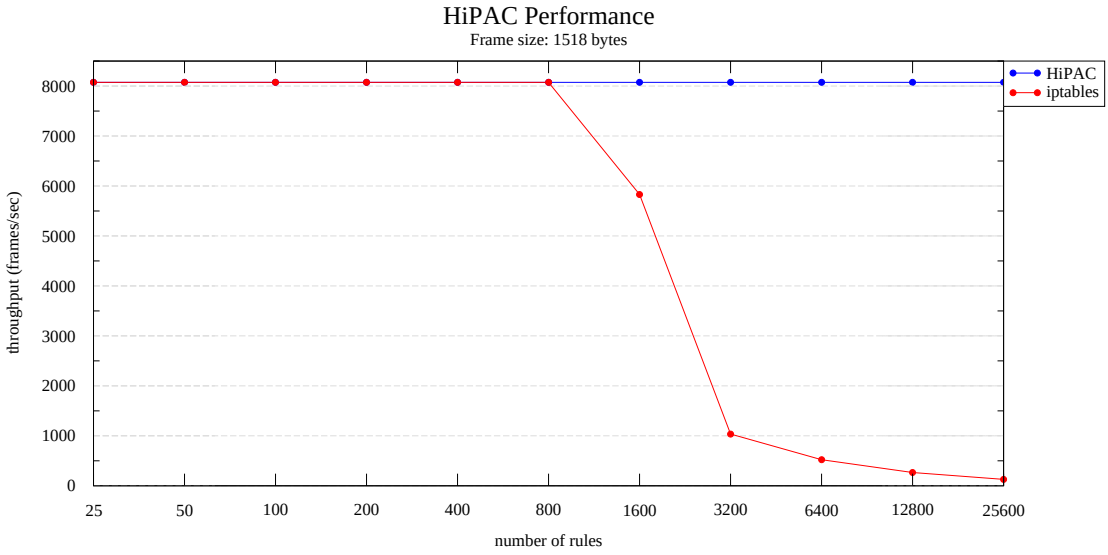

Figure 5.3: Results of the performance test involving 1518 byte sized frames. Note the logarithmic scale of the x-axis.

6 Outlook

HiPAC is a packet classification framework which goes far beyond a simple proof-of-concept implementation of a novel packet classification algorithm. It provides the same features as iptables – the well established packet filter of linux 2.4 – and is thus readily applicable as drop-in replacement without the user having to forgo any functionality. A high value is set on correctness and robustness of implementation to create a solid basis for incorporating a comparably complex algorithm into the linux kernel space. However, both HiPAC and PCP in general leave enough space for further optimizations and investigations. The following sections summarize some practical and theoretical aspects concerning these topics.

6.1 Practical aspects

[Bel04] presents a number of optimizations embarking on two strategies: optimizing the fundamental RLP data structure and increasing the memory efficiency of the whole decision data structure. The main proposition is to turn the MRLP tree into an equivalent, redundancy-free directed acyclic graph while the update operations remain dynamic. This improvement is expected to result in significantly smaller representations of real world rule sets which is critical to attain high cache performance.

Regarding the implementation, several small enhancements are desirable, e.g. support of arbitrary key lengths to enable native IPv6 and MAC matches. Moreover, the HiPAC core API should become transaction based like the MRLP layer to allow larger rule set modifications to be atomically committed. To increase the expressiveness of a single rule, the match representation could be generalized in the sense that instead of a single range, a set of ranges is used to represent a single match. This may lead to a significant reduction of rules in certain cases since a d-dimensional rule containing k ranges per match translates to k^d different rules containing a single range per match. Finally, HiPAC's data structure may be used to detect never matching rules, i.e. rules which don't occur in any leaf of the tree.

Although the HiPAC algorithm is primarily targeted towards a software based implementation, it might be reasonable to implement the RLP lookup in hardware. It would also be possible to implement the whole classification algorithm in hardware. However, the RLPs on the lookup path must be solved one after the other, so parallelism is not possible here.

Considering the evaluation of packet classification algorithms, research definitely has to be intensified. The development of general, algorithm independent and practically meaningful benchmarks has received very little attention of the research community

so far quite contrary to PCP itself. There are dozens of papers and articles about PCP, but in order to render a commonly accepted performance evaluation and to facilitate locating weak spots of the algorithms, a profound benchmark suite is indispensable.

6.2 Theoretical aspects

A fundamental property of HiPAC's decision data structure is the fixed order of dimensions, i.e. the sequence of dimensions (e.g. source IP, destination IP, protocol ...) is the same for any path. For a given rule set, the order may have tremendous consequences on the size of the data structure. Consider the following rule sets $\mathcal{R}_1$ and $\mathcal{R}_2$.

$$\mathcal{R}_1 = R^{def} \cup \{(i, M_i, 0),\ 1 \leq i < n \mid\ \forall 1 \leq j < d:\ M_i(j) = [i,\ max(\mathcal{U}) + i - n],\ M_i(d) = [2i-1,\ 2i-1]\}$$

$$\mathcal{R}_2 = R^{def} \cup \{(i, M_i, 0),\ 1 \leq i < n \mid\ \forall 2 \leq j \leq d:\ M_i(j) = [i,\ max(\mathcal{U}) + i - n],\ M_i(1) = [2i-1,\ 2i-1]\}$$

Note that $\mathcal{R}_1$ is equal to $\mathcal{R}_2$ apart from that the point match of each rule in $\mathcal{R}_1$ is in the d-th dimension while the point match of each rule in $\mathcal{R}_2$ is in the first dimension. Nevertheless, the space complexity of the MRLP tree representing $\mathcal{R}_1$ is $O(n^d)$ whereas the space complexity of the MRLP tree representing $\mathcal{R}_2$ is $O(nd)$ because in the root node (first dimension), each range of the partition is overlapped by a single rule. Although this is a marginal example, it shows that it would be interesting to have an estimate for the memory gain which is achievable by modifying the dimension order. Clearly, this cannot be done by trying all $d!$ different orders but some heuristics could be helpful.

In fact, the dimension order is a limitation of freedom since it would be possible for each node to choose its dimension among the remaining ones, i.e. the dimensions which are not already used on the path leading to the node. Although this new degree of freedom does not lower the worst case space complexity, it allows a more fine-grained tuning compared to changing the global dimension order which may lead to significant memory savings in certain rule sets. However, this approach impacts dynamic operations since changing the dimension of the source node (root) of a subgraph requires the whole subgraph to be rebuilt. Still, the main problem is to find an appropriate heuristics to decide which nodes should be assigned another dimension to increase the memory efficiency of the whole data structure.

In order to attack the problem of the high space complexity worst case, it could be useful to have a heuristics to collapse certain subgraphs and thus reducing the number of nodes in the decision data structure. The sink nodes in such a collapsed graph then contain a list of rules which must be linearly evaluated for each packet. Obviously, this makes only sense as long as the list is very small.

Since all software based PCP algorithms aiming at maximum classification performance suffer from a high space complexity worst case, it would be interesting to further investigate the "mystical" average case. Ideally, an abstract model would be developed which allows the instantiation of real world rule sets and traffic patterns. Such a model would also serve as foundation for a benchmark suite.

Finally, the question must be raised whether the commonly established formulation of PCP (or NPCP) is really suitable in practice. From an algorithmic point of view, the answer is probably yes since it is rather expressive. Contrarily, from a management point of view, the answer is presumably no simply because large rule sets are hard to understand. Hence, it may be useful to gain a better understanding of the requirements of different packet classification applications to develop a novel abstraction which is more suitable for rule set designers. Ideally, this abstraction yields a computationally simpler problem than PCP. However, if this approach turns out to be infeasible, PCP will remain an important and interesting challenge in the future.

A Linux kernel data structures

The following table lists previously mentioned linux 2.4 kernel data structures, functions and macros in the context of netfilter and iptables along with the file containing the definition. A • indicates that the data structure is introduced by HiPAC. The path in the second column is stated relative to the kernel source directory, e.g. `/usr/src/linux/`. `inc` is used as shortcut for `include`. [GG01] may be used to quickly search and browse the files and definitions.

function / data structure	location
BR_HIPAC_LOCK •	inc/linux/brlock.h
dev_close()	net/core/dev.c
IFF_UP	inc/linux/if.h
IPT_CONTINUE	inc/linux/netfilter_ipv4/ip_tables.h
IPT_RETURN	inc/linux/netfilter_ipv4/ip_tables.h
IPT_SO_GET_ENTRIES	inc/linux/netfilter_ipv4/ip_tables.h
IPT_SO_GET_INFO	inc/linux/netfilter_ipv4/ip_tables.h
IPT_SO_SET_ADD_COUNTERS	inc/linux/netfilter_ipv4/ip_tables.h
IPT_SO_SET_REPLACE	inc/linux/netfilter_ipv4/ip_tables.h
ipt_do_table()	net/ipv4/netfilter/ip_tables.c
ipt_init_match() •	net/ipv4/netfilter/ip_tables.c
ipt_init_target() •	net/ipv4/netfilter/ip_tables.c
kmalloc()	mm/slab.c
mark_source_chains()	net/ipv4/netfilter/ip_tables.c
NETDEV_CHANGENAME	inc/linux/notifier.h
NETDEV_DOWN	inc/linux/notifier.h
NETDEV_UP	inc/linux/notifier.h
NETLINK_NFHIPAC •	inc/linux/netlink.h
NETLINK_ROUTE	inc/linux/netlink.h
NF_ACCEPT	inc/linux/netfilter.h
NF_DROP	inc/linux/netfilter.h
NF_IP_FORWARD	inc/linux/netfilter_ipv4.h
NF_IP_LOCAL_IN	inc/linux/netfilter_ipv4.h
NF_IP_LOCAL_OUT	inc/linux/netfilter_ipv4.h
NF_IP_POST_ROUTING	inc/linux/netfilter_ipv4.h
NF_IP_PRE_ROUTING	inc/linux/netfilter_ipv4.h
NF_QUEUE	inc/linux/netfilter.h
NF_REPEAT	inc/linux/netfilter.h
NF_STOLEN	inc/linux/netfilter.h
netlink_dump_start()	net/netlink/af_netlink.c
netlink_dump_start_cb() •	net/netlink/af_netlink.c

nf_change_prio_hook() •	net/core/netfilter.c
printk()	kernel/printk.c
register_netdevice_veto() •	net/core/dev.c
struct ipt_entry	inc/linux/netfilter_ipv4/ip_tables.h
struct ipt_entry_match	inc/linux/netfilter_ipv4/ip_tables.h
struct ipt_entry_target	inc/linux/netfilter_ipv4/ip_tables.h
struct ipt_match	inc/linux/netfilter_ipv4/ip_tables.h
struct ipt_target	inc/linux/netfilter_ipv4/ip_tables.h
struct ip_conntrack	inc/linux/netfilter_ipv4/ip_conntrack.h
struct ip_conntrack_max	net/ipv4/netfilter/ip_conntrack_core.c
struct sk_buff	inc/linux/skbuff.h
unregister_netdevice_veto() •	net/core/dev.c
vmalloc()	inc/linux/vmalloc.h

List of Figures

Bibliography

[And01] Oskar Andreasson. Iptables tutorial. `http://iptables-tutorial.frozentux.net/`, 2001.

[ASP98] H. Adiseshu, S. Suri, and G. Parulkar. Packet filter management for layer 4 switching, 1998.

[BBD+01] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Claus Schröter, and Dirk Verworner. *Linux-Kernelprogrammierung*. Addison-Wesley, 2001.

[Bel04] Michael Bellion. Optimizing HiPAC for policy routing, packet filtering and traffic control, February 2004.

[BGP+94] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.

[BM99] S. Bradner and J. McQuaid. RFC 2544: Benchmarking methodology for network interconnect devices, March 1999. Status: INFORMATIONAL.

[BMG99] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[Bra91] S. Bradner. RFC 1242: Benchmarking terminology for network interconnection devices, July 1991. Status: INFORMATIONAL.

[BSW99] Milind M. Buddhikot, Subhash Suri, and Marcel Waldvogel. Space decomposition techniques for fast Layer-4 switching. In *Protocols for High Speed Networks IV (Proceedings of PfHSN '99)*, pages 25–41, Salem, MA, USA, 1999.

[BV01] Florin Baboescu and George Varghese. Scalable packet classification. In *SIGCOMM*, pages 199–210, 2001.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[EM01] David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Symposium on Discrete Algorithms*, pages 827–835, 2001.

[FM00] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *INFOCOM (3)*, pages 1193–1202, 2000.

[GG01] Arne Georg Gleditsch and Per Kristian Gjermshus. Cross-referencing linux. `http://lxr.linux.no/`, 2001.

[GM99a] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM*, pages 147–160, 1999.

[GM99b] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Proceedings Hot Interconnects VII*, Stanford, 1999.

[GM00] Pankaj Gupta and Nick McKeown. Dynamic algorithms with worst-case performance for packet classification. In *NETWORKING*, pages 528–539, 2000.

[GM01] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. IEEE Network, pp. 24-32, vol: 15:2, 2001.

[HNTM03] B. Hickman, D. Newman, S. Tadjudin, and T. Martin. RFC 3511: Benchmarking methodology for firewall performance, April 2003. Status: INFORMATIONAL.

[JCSV94] M. Jayaram, R. Cytron, D. Schmidt, and G. Varghese. Efficient demultiplexing of network packets by automatic parsing, 1994.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[kt00] kernelnewbies.org team. The kernelnewbies project. `http://kernelnewbies.org/`, 2000.

[LS98] T. V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, pages 203–214, 1998.

[MB01] Michael Mitzenmacher and Andrei Broder. Using multiple hash functions to improve IP lookups. In *INFOCOM*, pages 1454–1463, 2001.

[Mes02] Hans-Peter Messmer. *The Indispensable PC Hardware Book*. Addison-Wesley, 2002.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 39–51, 1987.

[New99] D. Newman. RFC 2647: Benchmarking terminology for firewall performance, August 1999. Status: INFORMATIONAL.

[Org97] Kernel.Org Organization. The linux kernel archives. `http://www.kernel.org/`, 1997.

[PA01] A. Prakash and A. Aziz. OC-3072 packet classification using BDDs and pipelined SRAMS, 2001.

[RBM+99] Paul "Rusty" Russell, Marc Boucher, James Morris, Harald Welte, Jozsef Kadlecsik, and Martin Josefsson. The netfilter/iptables project. `http://www.netfilter.org/`, 1999.

[SBVW03] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224. ACM Press, 2003.

[Sew02] Julian Seward. Valgrind. `http://valgrind.kde.org/`, 2002.

[SSV99] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *SIGCOMM*, pages 135–146, 1999.

[SV99] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, 1999.

[SVSW98] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM '98*, pages 191–202, 1998.

[SVW01] S. Suri, G. Varghese, and P. Warkhede. Multiway range trees: Scalable ip lookup with fast updates, 2001.

[Tan02] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2002.

[TT03] David E. Taylor and Jonathan S. Turner. Towards a packet classification benchmark. Technical Report WUCSE-2003-42, Department of Computer Science and Engineering, Washington University in Saint Louis, May 2003.

[Woo00] Thomas Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM (3)*, pages 1213–1222, 2000.

[WPR+02] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, and Marc Bechler. *Linux-Netzwerkarchitektur*. Addison-Wesley, 2002.

[WSV01] Priyank Ramesh Warkhede, Subhash Suri, and George Varghese. Fast packet classification for two-dimensional conflict-free filters. In *INFOCOM*, pages 1434–1443, 2001.

[WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM '97*, pages 25–36, September 1997.

[YBMM94] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter*, pages 153–165, 1994.