# 25. 报文分类及访问控制

DPDK提供了一个访问控制库，它能够根据一组分类规则对输入数据包进行分类。

ACL库用于对具有多个类别的一组规则执行N元组搜索，并为每个类别找到最佳匹配（最高优先级）。 库API提供以下基本操作：

* 创建一个新的访问控制（AC）上下文
* 向上下文中添加一条规则
* 对上下文中的所有规则，构建执行数据包分类所需的运行时结构
* 执行输入数据包分类
* 释放AC上下文及其运行时结构和相关的内存

## 25.1. 概述

### 25.1.1. 规则定义

当前的实现允许每个AC上下文的用户指定其自己的规则，通过该规则将执行数据包分类。 尽管规则字段布局几乎没有限制：

* 规则定义中的第一个字段必须是一个字节长。
* 所有后续字段必须分组为4个连续字节的集合。

这两个约束主要是出于性能原因 - 搜索函数将第一个输入字节作为流程设置的一部分进行处理，然后展开搜索函数的内部循环以一次处理四个输入字节。

为了定义AC规则中的各个字段，可以使用以下的数据结构：

struct rte\_acl\_field\_def {

uint8\_t type; /\*< type - ACL\_FIELD\_TYPE. \*/

uint8\_t size; /\*< size of field 1,2,4, or 8. \*/

uint8\_t field\_index; /\*< index of field inside the rule. \*/

uint8\_t input\_index; /\*< 0-N input index. \*/

uint32\_t offset; /\*< offset to start of field. \*/

};

各个字段的含义如下：

* type  
  这个字段可以有如下三种选择：
  + \_MASK - 用于像IP地址这种具有值和掩码字段
  + \_RANGE - 用于像port这种具有上下限期间的字段
  + \_BITMASK - 用于像协议标识符这种具有值和位掩码的字段
* size  
  size参数定义了字段的字节数，可用的值为1, 2, 4, 或8字节。 注意，由于输入字节分组，必须将1或2字节的字段定义为构成4个连续输入字节的连续字段。 并且，最好将8个或更多个字节的字段定义为4个字节字段，以便构建过程可以消除额外的字段。
* field\_index 表示规则中字段位置的基于零的值; 0到N-1为N字段。
* input\_index 如上所示，所有输入字段，除了第一个，必须是4个连续字节的组。 输入索引则指定了该字段属于哪个输入组。
* offset 偏移定义了该字段的偏移值。该偏移从buffer参数开始算起。

举例，为了定义IPv4 5元组分类结构：

struct ipv4\_5tuple {

uint8\_t proto;

uint32\_t ip\_src;

uint32\_t ip\_dst;

uint16\_t port\_src;

uint16\_t port\_dst;

};

可以使用以下数组字段定义：

struct rte\_acl\_field\_def ipv4\_defs[5] = {

/\* first input field - always one byte long. \*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof (uint8\_t),

.field\_index = 0,

.input\_index = 0,

.offset = offsetof (struct ipv4\_5tuple, proto),

},

/\* next input field (IPv4 source address) - 4 consecutive bytes. \*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 1,

.input\_index = 1,

.offset = offsetof (struct ipv4\_5tuple, ip\_src),

},

/\* next input field (IPv4 destination address) - 4 consecutive bytes. \*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 2,

.input\_index = 2,

.offset = offsetof (struct ipv4\_5tuple, ip\_dst),

},

/\*

\* Next 2 fields (src & dst ports) form 4 consecutive bytes.

\* They share the same input index.

\*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_RANGE,

.size = sizeof (uint16\_t),

.field\_index = 3,

.input\_index = 3,

.offset = offsetof (struct ipv4\_5tuple, port\_src),

},

{

.type = RTE\_ACL\_FIELD\_TYPE\_RANGE,

.size = sizeof (uint16\_t),

.field\_index = 4,

.input\_index = 3,

.offset = offsetof (struct ipv4\_5tuple, port\_dst),

},

};

这个IPv4 五元组的一个典型实例如下：

source addr/mask destination addr/mask source ports dest ports protocol/mask

192.168.1.0/24 192.168.2.31/32 0:65535 1234:1234 17/0xff

任何IPv4报文，具有协议ID为 17(UDP)，源IP为 192.168.1.[0-255]，目的IP为 192.168.2.31，源端口为 [0-65535] 且目的端口为 1234 的报文都匹配这个条目。

为了定义如下的IPv6 头部使用的2-元组分类: <protocol, IPv6 source address> ：

struct struct ipv6\_hdr {

uint32\_t vtc\_flow; /\* IP version, traffic class & flow label. \*/

uint16\_t payload\_len; /\* IP packet length - includes sizeof(ip\_header). \*/

uint8\_t proto; /\* Protocol, next header. \*/

uint8\_t hop\_limits; /\* Hop limits. \*/

uint8\_t src\_addr[16]; /\* IP address of source host. \*/

uint8\_t dst\_addr[16]; /\* IP address of destination host(s). \*/

} \_\_attribute\_\_((\_\_packed\_\_));

可以使用以下的数组字段：

struct struct rte\_acl\_field\_def ipv6\_2tuple\_defs[5] = {

{

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof (uint8\_t),

.field\_index = 0,

.input\_index = 0,

.offset = offsetof (struct ipv6\_hdr, proto),

},

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 1,

.input\_index = 1,

.offset = offsetof (struct ipv6\_hdr, src\_addr[0]),

},

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 2,

.input\_index = 2,

.offset = offsetof (struct ipv6\_hdr, src\_addr[4]),

},

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 3,

.input\_index = 3,

.offset = offsetof (struct ipv6\_hdr, src\_addr[8]),

},

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 4,

.input\_index = 4,

.offset = offsetof (struct ipv6\_hdr, src\_addr[12]),

},

};

典型实例如下：

source addr/mask protocol/mask

2001:db8:1234:0000:0000:0000:0000:0000/48 6/0xff

任何IPv6报文，具有协议ID为6 (TCP)，且源IP在范围 [2001:db8:1234:0000:0000:0000:0000:0000 - 2001:db8:1234:ffff:ffff:ffff:ffff:ffff] 内的报文都将匹配这个规则。

在下面的例子中，搜索键值的最后一个元素是8bit，因此，出现输入字段的4个字节未完全占用的情况。 分类结构为：

struct acl\_key {

uint8\_t ip\_proto;

uint32\_t ip\_src;

uint32\_t ip\_dst;

uint8\_t tos; /\*< 这里通常是32bit的元素 \*/

};

可以使用以下的数组字段：

struct rte\_acl\_field\_def ipv4\_defs[4] = {

/\* first input field - always one byte long. \*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof (uint8\_t),

.field\_index = 0,

.input\_index = 0,

.offset = offsetof (struct acl\_key, ip\_proto),

},

/\* next input field (IPv4 source address) - 4 consecutive bytes. \*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 1,

.input\_index = 1,

.offset = offsetof (struct acl\_key, ip\_src),

},

/\* next input field (IPv4 destination address) - 4 consecutive bytes. \*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_MASK,

.size = sizeof (uint32\_t),

.field\_index = 2,

.input\_index = 2,

.offset = offsetof (struct acl\_key, ip\_dst),

},

/\*

\* 尽管tos字段只需要1个字节，但是我们仍旧要申请4字节

\*/

{

.type = RTE\_ACL\_FIELD\_TYPE\_BITMASK,

.size = sizeof (uint32\_t), /\* All the 4 consecutive bytes are allocated \*/

.field\_index = 3,

.input\_index = 3,

.offset = offsetof (struct acl\_key, tos),

},

};

典型实例如下：

source addr/mask destination addr/mask tos/mask protocol/mask

192.168.1.0/24 192.168.2.31/32 1/0xff 6/0xff

任何IPv4报文，协议ID为6 (TCP)，源IP为192.168.1.[0-255]，目的IP为192.168.2.31，ToS 为1都匹配该规则。

当创建一组规则时，对于每个规则，还必须提供附加信息：

* priority: 衡量规则优先级的权重值，该值越大，优先级越高。 如果输入元组匹配多个规则，则返回优先级较高的规则。 请注意，如果输入元组匹配多于一个规则，并且这些规则具有相同的优先级，则未定义哪个规则作为匹配返回。 建议为每个规则分配唯一的优先级。
* category\_mask: 每个规则使用位掩码值来选择规则的相关类别。 当执行查找时，返回每个类别的结果。 如果例如有四个不同的ACL规则集，一个用于访问控制，一个用于路由等，则通过使单个搜索能够返回多个结果来有效地提供“并行查找”。 每个集合可以被分配自己的类别，并且通过将它们组合成单个数据库，一个查找返回四个集合中的每一个的结果。
* userdata: 用户定义的数值。 对于每个类别，成功匹配返回最高优先级匹配规则的userdata字段。当没有规则匹配时，返回值为零。

注意：

* 将新规则添加到ACL上下文中时，所有字段必须是主机字节顺序（LSB）。
* 当为输入元组执行搜索时，该元组中的所有字段必须是网络字节顺序（MSB）。

### 25.1.2. RT 内存大小限制

构建阶段 (rte\_acl\_build()) 为给定的一组规则创建内部结构以供运行时遍历。 当前的实现是一组多分枝树，分枝为8. 根据规则集，可能会消耗大量的内存。 为了节省一些空间，ACL构建过程尝试将给定的规则集拆分为几个不相交的子集，并为每个子集构建一个单独的trie。 根据规则集，它可能会减少RT内存需求，但可能会增加分类时间。 在构建时有可能为给定的AC上下文指定内部RT结构的最大内存限制。 可以通过 rte\_acl\_config 结构的 max\_size 字段来完成。 将其设置为大于0的值以指示 rte\_acl\_build() ：

* 尝试最小化RT表中的尝试次数，但是
* 确保RT表的大小不会超过给定值。

将其设置为零可使rte\_acl\_build（）使用默认行为：尝试最小化RT结构的大小，但不会暴露任何硬限制。  
这使用户能够对性能/空间权衡做出决定。

例如：

struct rte\_acl\_ctx \* acx;struct rte\_acl\_config cfg;int ret;

/\*

\* assuming that acx points to already created and

\* populated with rules AC context and cfg filled properly.

\*/

/\* try to build AC context, with RT structures less then 8MB. \*/

cfg.max\_size = 0x800000;

ret = rte\_acl\_build(acx, &cfg);

/\*

\* RT structures can't fit into 8MB for given context.

\* Try to build without exposing any hard limit.

\*/

if (ret == -ERANGE) {

cfg.max\_size = 0;

ret = rte\_acl\_build(acx, &cfg);

}

### 25.1.3. Classification 方法

在给定的AC上下文成功完成rte\_acl\_build()之后，它可以用于执行分类 - 搜索比输入数据高优先级的规则。

有几种分类算法实现：

* RTE\_ACL\_CLASSIFY\_SCALAR: 通用实现，不需要任何特殊的硬件支持
* RTE\_ACL\_CLASSIFY\_SSE: vector实现，可以实现8条流并行，需要 SSE 4.1 支持
* RTE\_ACL\_CLASSIFY\_AVX2: vector实现，可以实现16条流并行，需要 AVX2 支持

纯粹是运行时决定哪种方法来选择，没有建立时间的差异。 所有实现都在相同的内部RT结构上运行，并使用类似的原理。 主要区别在于矢量实现可以手动利用IA SIMD指令并并行处理多个输入数据流。 在启动时，ACL库确定给定平台的最高可用分类方法，并将其设置为默认的。 虽然用户有能力覆盖给定ACL上下文的默认分类器功能，或使用非默认分类方法执行特定搜索。 在这种情况下，用户有责任确保给定的平台支持选定的分类实现。

25.2. API用法  
注意：关于 Access Control API 的更多纤细信息，请参考 DPDK API Reference 。  
以下示例演示了更详细的多个类别的上面定义的规则的IPv4，5元组分类。

25.2.1. 多类别报文分类

struct rte\_acl\_ctx \* acx;struct rte\_acl\_config cfg;int ret;

/\* define a structure for the rule with up to 5 fields. \*/

RTE\_ACL\_RULE\_DEF(acl\_ipv4\_rule, RTE\_DIM(ipv4\_defs));

/\* AC context creation parameters. \*/

struct rte\_acl\_param prm = {

.name = "ACL\_example",

.socket\_id = SOCKET\_ID\_ANY,

.rule\_size = RTE\_ACL\_RULE\_SZ(RTE\_DIM(ipv4\_defs)),

/\* number of fields per rule. \*/

.max\_rule\_num = 8, /\* maximum number of rules in the AC context. \*/

};

struct acl\_ipv4\_rule acl\_rules[] = {

/\* matches all packets traveling to 192.168.0.0/16, applies for categories: 0,1 \*/

{

.data = {.userdata = 1, .category\_mask = 3, .priority = 1},

/\* destination IPv4 \*/

.field[2] = {.value.u32 = IPv4(192,168,0,0),. mask\_range.u32 = 16,},

/\* source port \*/

.field[3] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

/\* destination port \*/

.field[4] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

},

/\* matches all packets traveling to 192.168.1.0/24, applies for categories: 0 \*/

{

.data = {.userdata = 2, .category\_mask = 1, .priority = 2},

/\* destination IPv4 \*/

.field[2] = {.value.u32 = IPv4(192,168,1,0),. mask\_range.u32 = 24,},

/\* source port \*/

.field[3] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

/\* destination port \*/

.field[4] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

},

/\* matches all packets traveling from 10.1.1.1, applies for categories: 1 \*/

{

.data = {.userdata = 3, .category\_mask = 2, .priority = 3},

/\* source IPv4 \*/

.field[1] = {.value.u32 = IPv4(10,1,1,1),. mask\_range.u32 = 32,},

/\* source port \*/

.field[3] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

/\* destination port \*/

.field[4] = {.value.u16 = 0, .mask\_range.u16 = 0xffff,},

},

};

/\* create an empty AC context \*/

if ((acx = rte\_acl\_create(&prm)) == NULL) {

/\* handle context create failure. \*/

}

/\* add rules to the context \*/

ret = rte\_acl\_add\_rules(acx, acl\_rules, RTE\_DIM(acl\_rules));if (ret != 0) {

/\* handle error at adding ACL rules. \*/

}

/\* prepare AC build config. \*/

cfg.num\_categories = 2;

cfg.num\_fields = RTE\_DIM(ipv4\_defs);

memcpy(cfg.defs, ipv4\_defs, sizeof (ipv4\_defs));

/\* build the runtime structures for added rules, with 2 categories. \*/

ret = rte\_acl\_build(acx, &cfg);if (ret != 0) {

/\* handle error at build runtime structures for ACL context. \*/

}

对于源IP地址：10.1.1.1和目标IP地址：192.168.1.15的元组，一旦执行如下操作：

uint32\_t results[4]; /\* make classify for 4 categories. \*/

rte\_acl\_classify(acx, data, results, 1, 4);

结果数组包含：

results[4] = {2, 3, 0, 0};

* 对于类别0，规则1和2都匹配，但规则2具有较高的优先级，因此[0]包含规则2的用户数据。
* 对于类别1，规则1和3都匹配，但规则3具有较高的优先级，因此[1]包含规则3的用户数据。
* 对于类别2和3，没有匹配，结果[2]和结果[3]包含零，这表明没有找到匹配的那些类别。

对于源IP地址为192.168.1.1和目标IP地址：192.168.2.11的元组，一旦执行：

uint32\_t results[4]; /\* make classify by 4 categories. \*/

rte\_acl\_classify(acx, data, results, 1, 4);

结果数组包含：

results[4] = {1, 1, 0, 0};

* 对于0和1类，只有规则1匹配。
* 对于类别2和3，没有匹配。

对于源IP地址：10.1.1.1和目标IP地址：201.212.111.12的元组，一旦执行：

uint32\_t results[4]; /\* make classify by 4 categories. \*/

rte\_acl\_classify(acx, data, results, 1, 4);

结果数组包含：

results[4] = {0, 3, 0, 0};

* 对于类别1，只有规则3匹配。
* 对于0,2和3类，没有匹配。