Project:
Netfilter/iptables

Andrew Rimes - cs253030
Kreshnik Mati - cse93062
CSE4221 - F09

December 8, 2009

Contents

1 Overview **3**

2 Rule Tables **3**
2.1 Overview . . . 3
2.2 Data Structures . . . 3
2.2.1 ip_tables.h/ipt_ip . . . 3
2.2.2 ip_tables.h/ipt_entry . . . 3
2.2.3 x_tables.h/xt_entry_match (ipt_entry_match) . . . 4
2.2.4 x_tables.h/xt_match . . . 4
2.2.5 xt_mtchk_param . . . 5
2.2.6 x_tables/xt_target (ipt_entry_target) . . . 5
2.3 Diagram . . . 7
2.4 Execution . . . 7
2.4.1 ip_tables.h/IPT_MATCH_ITERATE . . . 7
2.4.2 XT_MATCH_ITERATE . . . 7
2.4.3 XT_ENTRY_ITERATE_CONTINUE and XT_ENTRY_ITERATE . . . 8
2.4.4 ip_tables.c/ip_packet_match . . . 8
2.4.5 do_match . . . 9
2.4.6 do_table . . . 9

3 ConnTrack **11**
3.1 Overview . . . 11
3.2 Data Structures . . . 11
3.2.1 nf_conntrack_tuple.h/nf_conntrack_tuple . . . 11
3.2.2 nf_conntrack_tuple.h/nf_conntrack_tuple_hash . . . 12
3.2.3 nf_conntrack_tuple.h/ip_conntrack_old_tuple . . . 12
3.2.4 nf_conntrack_tuple.h/xt_conntrack_info . . . 13
3.2.5 nf_conntrack.h/nf_conn . . . 13
3.2.6 conntrack.h/netns_ct . . . 14
3.2.7 nf_conntrack_expect.h/nf_conntrack_expect . . . 14
3.2.8 Diagram . . . 15
3.3 Execution . . . 17
3.3.1 Initiating a new connection . . . 20
3.3.2 Reply to new connection . . . 20

4 Registration **20**
4.1 Hook Object . . . 20
4.2 Hook List Object . . . 21
4.3 Hook Adding . . . 22
4.4 Hook List Adding . . . 23
4.5 Hook And Hook-List Removing . . . 24
4.6 Hooks Processing . . . 24

1 Overview

iptables usually refers to the framework and associated modules that make up the firewall and NAT subsystem in the Linux kernel. (It is more probably called X_tables since it supports several network protocols including IPv4, IPv6 and ARP.) iptables is also the name of the userland program used for manipulating the in-kernel data structures that define the firewall and NAT rules. The framework provides hooks for general *packet mangling* and can be used to implement kernel modules that in someway intercept, modify or track packets flowing through the Linux TCP/IP stack, e.g. an IPSec VPN could be implemented in this manner. This report will cover some of the major data structures that make up the firewalls rules and matching mechanism (2), the connection tracking system (3) and the registration system(4) that allows modules to be registered to event hooks.

2 Rule Tables

2.1 Overview

An IP table is made of multiple chains; a chain is a list of rules associated with one or more hooks. Each rule consists of one or more matches, terminated by a target. The target is the action performed when a rule is entirely matched. The data structures are designed such that they can be used in both kernel mode and user mode by the iptables command.

2.2 Data Structures

2.2.1 ip_tables.h/ipt_ip

This struct specifies the minimal IP header information needed to to identify a packet. The src and dst fields identify the source and destination IP addresses, and smsk and dmsk identify the source and destination masks so that ranges of IP addresses can be specified in the rule, e.g. 192.168.0.0/24 would match all IPs in the 192.168.0.0 subnet[3]. The initface and outiface fields specify inbound and outbound interfaces, and iniface_mask and outiface_mask allow the user to specify several interfaces at once. The protocol number is stored in proto, e.g. it would be set to 6 for TCP. The fields flags and invflags specified the IP header flags selected and not selected.

```
struct ipt_ip {
        /* Source and destination IP addr */
        struct in_addr src, dst;
        /* Mask for src and dest IP addr */
        struct in_addr smsk, dmsk;
        char iniface[IFNAMSIZ], outiface[IFNAMSIZ];
        unsigned char iniface_mask[IFNAMSIZ], outiface_mask[IFNAMSIZ];

        /* Protocol, 0 = ANY */
        u_int16_t proto;

        /* Flags word */
        u_int8_t flags;
        /* Inverse flags */
        u_int8_t invflags;
};
```

2.2.2 ip_tables.h/ipt_entry

This structure defines the starting point of each of the firewall rules, which are contained in arrays, i.e. tables. The nf_cache bitfield shows what parts of the packet this rule exams. The target_offset field indicates the offset between the beginning of the current rule (contained in the structure) and where the ipt_entry_target begins. As indicated in the comments, the target offset is equal to the size of the ipt_entry and the total number of matches. The next_offset is the sum of the entry, matches, and target, which determines the position of the next rule's ipt_entry. The target, described elsewhere, is executed when the rule matches. The comefrom field is a back pointer used by the kernel to track packet traversal. Obviously, counters stores packet and byte counts for the rule, i.e. how many have passed through. The final, variable length field `elems` is where matches, terminated by the target, are stored.

```
struct ipt_entry
{
        struct ipt_ip ip;

        /* Mark with fields that we care about. */
        unsigned int nfcache;

        /* Size of ipt_entry + matches */
        u_int16_t target_offset;
        /* Size of ipt_entry + matches + target */
        u_int16_t next_offset;

        /* Back pointer */
        unsigned int comefrom;

        /* Packet and byte counters. */
        struct xt_counters counters;

        /* The matches (if any), then the target. */
        unsigned char elems[0];
};
```

2.2.3 x_tables.h/xt_entry_match (ipt_entry_match)

This provides a thin layer of abstraction so that code can be reused regardless of whether it is executing in the kernel or in user space. We will discuss `xt_match` since we are concerned with the kernel[1].

```
struct xt_entry_match
{
        union {
                struct {
                        __u16 match_size;

                        /* Used by userspace */
                        char name[XT_FUNCTION_MAXNAMELEN-1];

                        __u8 revision;
                } user;
                struct {
                        __u16 match_size;

                        /* Used inside the kernel */
                        struct xt_match *match;
                } kernel;

                /* Total length */
                __u16 match_size;
        } u;

        unsigned char data[0];
};
```

2.2.4 x_tables.h/xt_match

This structure represents a match entry in a rule (possibly one of many). It is fairly abstract and meant to represent all types of matches.
The `list` is the standard doubly linked list used in the Linux kernel; this simple struct has two fields, `prev` and `nest`, and is defined in `include/linux/list.h`. In this case it is used to link together consecutive matches in the rule. The `name` field stores the name of the current table. The `revision` field stores the current revision number of the data structure so that if it changes in the future, backwards compatibility can be maintained.

[1] In some cases, structs have two names: one beginning with xt and one without. This is a result of the unification of common code in `ip_tables`, `ip6_tables` and `arp_tables` into `x_tables` (hence xt) – a table structure that can handle all three protocols.

The match field is a pointer to the Boolean function that determines whether or not the packet will be matched. The skb parameter is a pointer to a copy of the packet and xt_match_param is a pointer to additional parameters the match function can use to make the decision.
The function pointed to by checkentry is called when the user attempts to add this type of match to the rule (whatever that type is). The struct xt_mtchk_param (2.2.5) is passed to make that decision.

```
struct xt_match
{
        struct list_head list;

        const char name[XT_FUNCTION_MAXNAMELEN-1];
        u_int8_t revision;

        /* Return true or false: return FALSE and set *hotdrop = 1 to
           force immediate packet drop. */
       bool (*match)(const struct sk_buff *skb,
                     const struct xt_match_param *);

        /* Called when user tries to insert an entry of this type. */
        bool (*checkentry)(const struct xt_mtchk_param *);

        /* Called when entry of this type deleted. */
        void (*destroy)(const struct xt_mtdtor_param *);

        /* Called when userspace align differs from kernel space one */
        void (*compat_from_user)(void *dst, void *src);
        int (*compat_to_user)(void __user *dst, void *src);

        /* Set this to THIS_MODULE if you are a module, otherwise NULL */
        struct module *me;

        /* Free to use by each match */
        unsigned long data;

        const char *table;
        unsigned int matchsize;
        unsigned int compatsize;
        unsigned int hooks;
        unsigned short proto;

        unsigned short family;
};
```

2.2.5 xt_mtchk_param

This struct is passed to a match extension's checkentry function. The name of the table is in table. Protocol family information, (e.g. ipt_ip (2.2.1) for IPv4) is stored in entryinfo. The pointer match is the xt_match through which this function was invoked. The protocol family number is stored in family. Which hooks the new rule is reachable from is stored in hook_mask where each hook is defined by a bit. Any additional information is stored in matchinfo.

```
struct xt_mtchk_param {
        const char *table;
        const void *entryinfo;
        const struct xt_match *match;
        void *matchinfo;
        unsigned int hook_mask;
        u_int8_t family;
};
```

2.2.6 x_tables/xt_target (ipt_entry_target)

This structure represents the final target entry in a rule and is analogous to xt_match (2.2.4). Just as xt_match has a pointer to a match and checkentry functions, xt_target has pointers to a target function and its own `checkentry` function. Similarly, it has to deal with possible differences in memory alignment between kernel and user space. The `me` pointer is used to identify the entry within modules.

```
struct xt_target
{
        struct list_head list;

        const char name[XT_FUNCTION_MAXNAMELEN-1];

        /* Returns verdict. Argument order changed since 2.6.9, as this
           must now handle non-linear skbs, using skb_copy_bits and
           skb_ip_make_writable. */
        unsigned int (*target)(struct sk_buff *skb,
                               const struct xt_target_param *);

        /* Called when user tries to insert an entry of this type:
           hook_mask is a bitmask of hooks from which it can be
           called. */
        /* Should return true or false. */
        bool (*checkentry)(const struct xt_tgchk_param *);

        /* Called when entry of this type deleted. */
        void (*destroy)(const struct xt_tgdtor_param *);

        /* Called when userspace align differs from kernel space one */
        void (*compat_from_user)(void *dst, void *src);
        int (*compat_to_user)(void __user *dst, void *src);

        /* Set this to THIS_MODULE if you are a module, otherwise NULL */
        struct module *me;

        const char *table;
        unsigned int targetsize;
        unsigned int compatsize;
        unsigned int hooks;
        unsigned short proto;

        unsigned short family;
        u_int8_t revision;
};
```

2.3 Diagram

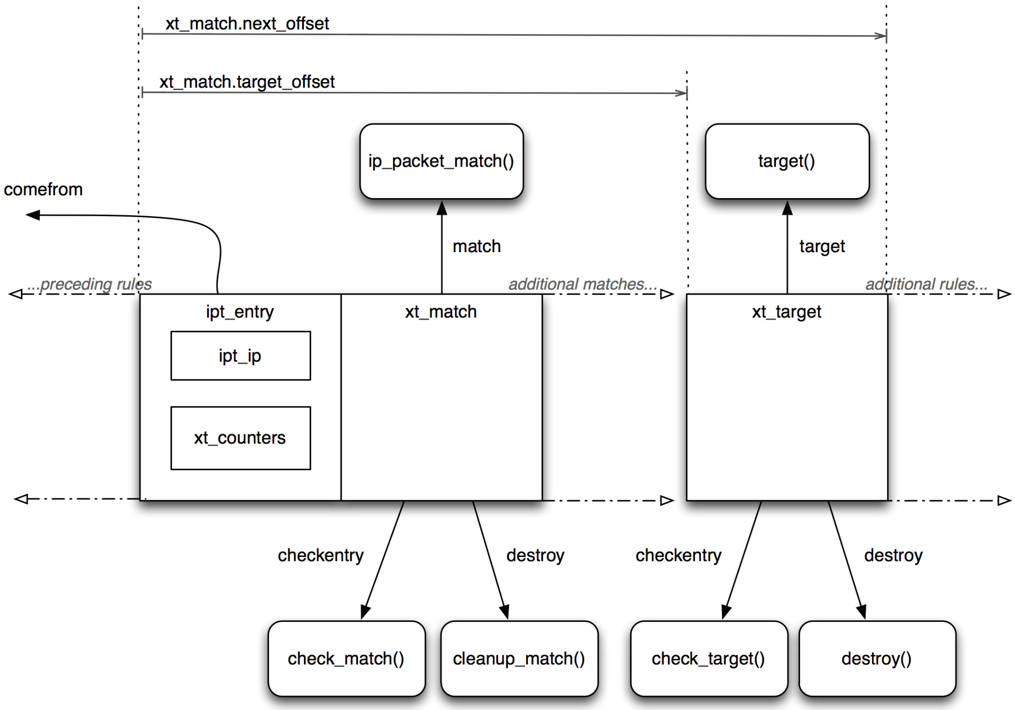

Figure 1: The structure of an `iptables` IPv4 table.

2.4 Execution

2.4.1 ip_tables.h/IPT_MATCH_ITERATE

For historical reasons, the IP specific `IPT_MATCH_ITERATE` macro calls the more general `XT_MATCH_ITERATE`.

```
  /* fn returns 0 to continue iteration */
#define IPT_MATCH_ITERATE(e, fn, args...) \
        XT_MATCH_ITERATE(struct ipt_entry, e, fn, ## args)
```

Similarly, the IPv4 specific `IPT_ENTRY_ITERATE` calls `XT_ENTRY_ITERATE` to iterate through the rule entries beginning at `ipt_entry`.

```
  /* fn returns 0 to continue iteration */
#define IPT_ENTRY_ITERATE(entries, size, fn, args...) \
        XT_ENTRY_ITERATE(struct ipt_entry, entries, size, fn, ## args)
```

2.4.2 XT_MATCH_ITERATE

The `XT_MATCH_ITERATE` macro expands to a for loop that iterates through the list of `xt_match` structures starting from the `ipt_entry`. With each step, the index into the rule is advanced by the length of the current `xp_match` object and the associated match function is called. If the match functions returns 0, i.e. the packet matches, the iteration continues.

```
/* fn returns 0 to continue iteration */
#define XT_MATCH_ITERATE(type, e, fn, args...)                \
({                                                            \
        unsigned int __i;                                     \
        int __ret = 0;                                        \
        struct xt_entry_match *__m;                           \
                                                              \
```

```
        for (__i = sizeof(type);                                    \
             __i < (e)->target_offset;                              \
             __i += __m->u.match_size) {                            \
                __m = (void *)e + __i;                              \
                                                                    \
                __ret = fn(__m , ## args);                          \
                if (__ret != 0)                                     \
                        break;                                      \
        }                                                           \
        __ret;                                                      \
})
```

2.4.3 XT_ENTRY_ITERATE_CONTINUE and XT_ENTRY_ITERATE

The `XT_ENTRY_ITERATE_CONTINUE` and `XT_ENTRY_ITERATE` macros are used whenever it is necessary to walk through a table applying the function `fn` on each entry[2].

```
/* fn returns 0 to continue iteration */
#define XT_ENTRY_ITERATE(type, entries, size, fn, args...) \
        XT_ENTRY_ITERATE_CONTINUE(type, entries, size, 0, fn, args)

/* fn returns 0 to continue iteration */
#define XT_ENTRY_ITERATE_CONTINUE(type, entries, size, n, fn, args...) \
({                                                                  \
        unsigned int __i, __n;                                      \
        int __ret = 0;                                              \
        type *__entry;                                              \
                                                                    \
        for (__i = 0, __n = 0; __i < (size);                        \
             __i += __entry->next_offset, __n++) {                  \
                __entry = (void *)(entries) + __i;                  \
                if (__n < n)                                        \
                        continue;                                   \
                                                                    \
                __ret = fn(__entry , ## args);                      \
                if (__ret != 0)                                     \
                        break;                                      \
        }                                                           \
        __ret;                                                      \
})
```

2.4.4 ip_tables.c/ip_packet_match

Below is the basic match function that is performed on the header of every incoming packet (with some debugging statements omitted). Lines 12-17 apply the netmask from the rule to the source and destination IPs of the packet header `iphdr` and verifies that they correspond to the fields in `ipt_ip`(2.2.1). It also verifies that the rule is checking source and destination addresses. Lines 19-29 verify that the incoming and outgoing interface of the packet correspond to the rule and that the rule checks for interfaces. Lines 31-35 verify the protocol matches and that the protocol is being checked. The final check on line 39 verifies that if the match applies to fragmented packets, that the packet is fragmented.

```
1  static inline bool
2  ip_packet_match(const struct iphdr *ip,
3                  const char *indev,
4                  const char *outdev,
5                  const struct ipt_ip *ipinfo,
6                  int isfrag)
7  {
8          unsigned long ret;
9
10 #define FWINV(bool, invflg) ((bool) ^ !!(ipinfo->invflags & (invflg)))
11
12         if (FWINV((ip->saddr&ipinfo->smsk.s_addr) != ipinfo->src.s_addr,
13                   IPT_INV_SRCIP)
```

[2] `XT_ENTRY_ITERATE` skips the application of `fn` on the first entry.

```
                || FWINV((ip->daddr&ipinfo->dmsk.s_addr) != ipinfo->dst.s_addr,
                        IPT_INV_DSTIP)) {
                return false;
        }

        ret = ifname_compare_aligned(indev, ipinfo->iniface, ipinfo->iniface_mask);

        if (FWINV(ret != 0, IPT_INV_VIA_IN)) {
                return false;
        }

        ret = ifname_compare_aligned(outdev, ipinfo->outiface, ipinfo->outiface_mask);

        if (FWINV(ret != 0, IPT_INV_VIA_OUT)) {
                return false;
        }

        /* Check specific protocol */
        if (ipinfo->proto
            && FWINV(ip->protocol != ipinfo->proto, IPT_INV_PROTO)) {
                return false;
        }

        /* If we have a fragment rule but the packet is not a fragment
         * then we return zero */
        if (FWINV((ipinfo->flags&IPT_F_FRAG) && !isfrag, IPT_INV_FRAG)) {
                return false;
        }

        return true;
}
```

2.4.5 do_match

The `do_match` function is the function called by the `XT_MATCH_ITERATOR` macro. It executes the match routine associated with each match entry in the rule, for example it would call `ip_packet_match` above for a basic IPv4 match.

```
 static inline bool
do_match(struct ipt_entry_match *m, const struct sk_buff *skb,
         struct xt_match_param *par)
{
        par->match     = m->u.kernel.match;
        par->matchinfo = m->data;

        /* Stop iteration if it doesn't match */
        if (!m->u.kernel.match->match(skb, par))
                return true;
        else
                return false;
}
```

2.4.6 do_table

The `do_table` function is an important function that walks through the match table. The main `do` loop begins on line 53. The loop continues so long as the matches are successful. Eventually a target is called and it returns a `verdict` on whether or not the packet should be accepted. If the verdict is to continue, it moves on to the next rule. A hotdrop breaks out of the loop early if the packet is to be dropped immediately.

```
1   /* Returns one of the generic firewall policies, like NF_ACCEPT. */
2  unsigned int
3  ipt_do_table(struct sk_buff *skb,
4               unsigned int hook,
5               const struct net_device *in,
6               const struct net_device *out,
7               struct xt_table *table)
```

```
{
#define tb_comefrom ((struct ipt_entry *)table_base)->comefrom

	static const char nulldevname[IFNAMSIZ] __attribute__((aligned(sizeof(long))));
	const struct iphdr *ip;
	u_int16_t datalen;
	bool hotdrop = false;
	/* Initializing verdict to NF_DROP keeps gcc happy. */
	unsigned int verdict = NF_DROP;
	const char *indev, *outdev;
	void *table_base;
	struct ipt_entry *e, *back;
	struct xt_table_info *private;
	struct xt_match_param mtpar;
	struct xt_target_param tgpar;

	/* Initialization */
	ip = ip_hdr(skb);
	datalen = skb->len - ip->ihl * 4;
	indev = in ? in->name : nulldevname;
	outdev = out ? out->name : nulldevname;
	/* We handle fragments by dealing with the first fragment as
	 * if it was a normal packet.  All other fragments are treated
	 * normally, except that they will NEVER match rules that ask
	 * things we don't know, ie. tcp syn flag or ports).  If the
	 * rule is also a fragment-specific rule, non-fragments won't
	 * match it. */
	mtpar.fragoff = ntohs(ip->frag_off) & IP_OFFSET;
	mtpar.thoff   = ip_hdrlen(skb);
	mtpar.hotdrop = &hotdrop;
	mtpar.in      = tgpar.in  = in;
	mtpar.out     = tgpar.out = out;
	mtpar.family  = tgpar.family = NFPROTO_IPV4;
	mtpar.hooknum = tgpar.hooknum = hook;

	IP_NF_ASSERT(table->valid_hooks & (1 << hook));
	xt_info_rdlock_bh();
	private = table->private;
	table_base = private->entries[smp_processor_id()];

	e = get_entry(table_base, private->hook_entry[hook]);

	/* For return from builtin chain */
	back = get_entry(table_base, private->underflow[hook]);

	do {
		struct ipt_entry_target *t;

		IP_NF_ASSERT(e);
		IP_NF_ASSERT(back);
		if (!ip_packet_match(ip, indev, outdev,
		    &e->ip, mtpar.fragoff) ||
		    IPT_MATCH_ITERATE(e, do_match, skb, &mtpar) != 0) {
			e = ipt_next_entry(e);
			continue;
		}

		ADD_COUNTER(e->counters, ntohs(ip->tot_len), 1);

		t = ipt_get_target(e);
		IP_NF_ASSERT(t->u.kernel.target);

		/* Standard target? */
		if (!t->u.kernel.target->target) {
			int v;

			v = ((struct ipt_standard_target *)t)->verdict;
			if (v < 0) {
				/* Pop from stack? */
				if (v != IPT_RETURN) {
```

```
                                        verdict = (unsigned)(-v) - 1;
                                        break;
                                }
                                e = back;
                                back = get_entry(table_base, back->comefrom);
                                continue;
                        }
                        if (table_base + v != ipt_next_entry(e)
                            && !(e->ip.flags & IPT_F_GOTO)) {
                                /* Save old back ptr in next entry */
                                struct ipt_entry *next = ipt_next_entry(e);
                                next->comefrom = (void *)back - table_base;
                                /* set back pointer to next entry */
                                back = next;
                        }

                        e = get_entry(table_base, v);
                        continue;
                }

                /* Targets which reenter must return
                   abs. verdicts */
                tgpar.target   = t->u.kernel.target;
                tgpar.targinfo = t->data;

                verdict = t->u.kernel.target->target(skb, &tgpar);

                /* Target might have changed stuff. */
                ip = ip_hdr(skb);
                datalen = skb->len - ip->ihl * 4;

                if (verdict == IPT_CONTINUE)
                        e = ipt_next_entry(e);
                else
                        /* Verdict */
                        break;
        } while (!hotdrop);
        xt_info_rdunlock_bh();

        if (hotdrop)
                return NF_DROP;
        else return verdict;


#undef tb_comefrom
}
```

3 ConnTrack

3.1 Overview

The `conntrack` system provides connection tracking so that as each packet is processed, its context within in a connection is known. This allows for more intelligent filtering decisions to be made and is essential to implement a so-called *stateful firewall*, as opposed to a simple packet filter. Other modules that require connection tracking are built on top of the basic functionality provided by `conntrack`.

3.2 Data Structures

3.2.1 nf_conntrack_tuple.h/nf_conntrack_tuple

The `nf_conntrack_tuple` structure is used to uniquely identify a connection that has passed through the firewall and whose state is being tracked. The `src` field is the source address, which is considered manipulable (useful for NAT), unlike the remaining fields, which are immutable. The field `u3` represents the destination address in either IPv4 and IPv6, depending on what type of packet is being tracked. The union `u` with `tcp`, `udp`, `icmp`, etc., track layer 4 protocol state fields for the destination, such as the port number for TCP[3]. The `all` field is used when multiple fields need to be accessed at once, e.g. for hashing of the tuple. Finally, `protonum` stores the protocol in use, (TCP or UDP, etc.) and `dir` stores the direction.

```
struct nf_conntrack_tuple
{
        struct nf_conntrack_man src;

        struct {
                union nf_inet_addr u3;
                union {
                        /* Add other protocols here. */
                        __be16 all;

                        struct {
                                __be16 port;
                        } tcp;
                        struct {
                                __be16 port;
                        } udp;
                        struct {
                                u_int8_t type, code;
                        } icmp;
                        struct {
                                __be16 port;
                        } dccp;
                        struct {
                                __be16 port;
                        } sctp;
                        struct {
                                __be16 key;
                        } gre;
                } u;

                /* The protocol. */
                u_int8_t protonum;

                /* The direction (for tuplehash) */
                u_int8_t dir;
        } dst;
};
```

3.2.2 nf_conntrack_tuple.h/nf_conntrack_tuple_hash

```
 /* Connections have two entries in the hash table: one for each way */
struct nf_conntrack_tuple_hash {
        struct hlist_nulls_node hnnode;
        struct nf_conntrack_tuple tuple;
};
```

3.2.3 nf_conntrack_tuple.h/ip_conntrack_old_tuple

This is an old IPv4 specific format for the tuple. It is kept for historical reasons since it has been exposed to userspace.

```
/* This is exposed to userspace, so remains frozen in time. */
struct ip_conntrack_old_tuple
{
        struct {
                __be32 ip;
                union {
                        __u16 all;
                } u;
        } src;

        struct {
                __be32 ip;
                union {
```

```
                        __u16 all;
                } u;

                /* The protocol. */
                __u16 protonum;
        } dst;
};
```

3.2.4 nf_conntrack_tuple.h/xt_conntrack_info

```
struct xt_conntrack_info
{
        unsigned int statemask, statusmask;

        struct ip_conntrack_old_tuple tuple[IP_CT_DIR_MAX];
        struct in_addr sipmsk[IP_CT_DIR_MAX], dipmsk[IP_CT_DIR_MAX];

        unsigned long expires_min, expires_max;

        /* Flags word */
        __u8 flags;
        /* Inverse flags */
        __u8 invflags;
};
```

3.2.5 nf_conntrack.h/nf_conn

The `ct_general` is an atomic counter that tracks the usage of the `nf_conn` structure. The spinlock is used to protect the connection on SMP systems. The `tuplehash` is a two element array that contains the tuple for the initiating side of the connection (index `IP_CT_DIR_ORIGINAL`) and a second tuple for the reply packet (index `IP_CT_DIR_ORIGINAL`). The reply tuple can be easily calculated by inverting fields. For example, to generate the reply tuple for a TCP connection, the source and destination IP addresses of the initial packet are inverted, as are the source and destination port numbers[3].

```
  struct nf_conn {
        /* Usage count in here is 1 for hash table/destruct timer, 1 per skb,
           plus 1 for any connection(s) we are 'master' for */
        struct nf_conntrack ct_general;

        spinlock_t lock;

        /* These are my tuples; original and reply */
        struct nf_conntrack_tuple_hash tuplehash[IP_CT_DIR_MAX];

        /* Have we seen traffic both ways yet? (bitset) */
        unsigned long status;

        /* If we were expected by an expectation, this will be it */
        struct nf_conn *master;

        /* Timer function; drops refcnt when it goes off. */
        struct timer_list timeout;

#if defined(CONFIG_NF_CONNTRACK_MARK)
        u_int32_t mark;
#endif

#ifdef CONFIG_NF_CONNTRACK_SECMARK
        u_int32_t secmark;
#endif

        /* Storage reserved for other modules: */
        union nf_conntrack_proto proto;

        /* Extensions */
        struct nf_ct_ext *ext;
```

```
#ifdef CONFIG_NET_NS
        struct net *ct_net;
#endif
};
```

3.2.6 conntrack.h/netns_ct

This points to global connection tracking entities like the `expect_hash` hash table for expected packets [fig. 3]. The atomic `count` refers to the number of active connections being tracked through the firewall, and `hash` points to the hash table for active connections. The `unconfirmed` table is where tuples are temporarily stashed before the corresponding packet leaves the firewall [fig. 4].

```
  struct netns_ct {
        atomic_t                count;
        unsigned int            expect_count;
        struct hlist_nulls_head *hash;
        struct hlist_head       *expect_hash;
        struct hlist_nulls_head unconfirmed;
        struct hlist_nulls_head dying;
        struct ip_conntrack_stat *stat;
        int                     sysctl_events;
        unsigned int            sysctl_events_retry_timeout;
        int                     sysctl_acct;
        int                     sysctl_checksum;
        unsigned int            sysctl_log_invalid; /* Log invalid packets */
#ifdef CONFIG_SYSCTL
        struct ctl_table_header *sysctl_header;
        struct ctl_table_header *acct_sysctl_header;
        struct ctl_table_header *event_sysctl_header;
#endif
        int                     hash_vmalloc;
        int                     expect_vmalloc;
};
```

3.2.7 nf_conntrack_expect.h/nf_conntrack_expect

```
  struct nf_conntrack_expect
{
        /* Conntrack expectation list member */
        struct hlist_node lnode;

        /* Hash member */
        struct hlist_node hnode;

        /* We expect this tuple, with the following mask */
        struct nf_conntrack_tuple tuple;
        struct nf_conntrack_tuple_mask mask;

        /* Function to call after setup and insertion */
        void (*expectfn)(struct nf_conn *new,
                         struct nf_conntrack_expect *this);

        /* Helper to assign to new connection */
        struct nf_conntrack_helper *helper;

        /* The conntrack of the master connection */
        struct nf_conn *master;

        /* Timer function; deletes the expectation. */
        struct timer_list timeout;

        /* Usage count. */
        atomic_t use;

        /* Flags */
        unsigned int flags;
```

```
        /* Expectation class */
        unsigned int class;

#ifdef CONFIG_NF_NAT_NEEDED
        __be32 saved_ip;
        /* This is the original per-proto part, used to map the
         * expected connection the way the recipient expects. */
        union nf_conntrack_man_proto saved_proto;
        /* Direction relative to the master connection. */
        enum ip_conntrack_dir dir;
#endif

        struct rcu_head rcu;
};
```

3.2.8 Diagram

The hash table shown in diagram [fig. 2] is used to store tuples for established connections. Each entry points to a list of tuples with the same hash key. New tuples are inserted at the head.
The efficiency of this structure is extremely important since a packet's tuple must be looked up every time it arrives on an interface or is sent from an internal process. Space must also be considered because the table can grow quite large on a busy firewall handling thousands of connections concurrently.
One space optimization was the use of `hlist_node` and `hlist_head` over the more general `list_head`. The latter has a pointer to the tail of the list, which means the tail can be reached in $O(N)$ time. However, since this is not needed for tuple look-ups[3], the extra pointer in `list_head` is superfluous. The `hlist_head` contains only a `first` pointer field. Although, this creates another complication: when accessing the previous node (e.g. for insertion) it is no longer known whether it is a `hlist_node` or a `hlist_head`. To get around this, the `pprev` field is a pointer to a pointer – a pointer to whatever field is pointing to the current node. Thus, for insertions and deletions, the previous node/head can be updated regardless of type.
To access the containing structure of an `hlist_node`, in this case an `nf_conntrack_tuple_hash`, the macro `container_of` is used to cleverly calculate the address containing structure address from the member address.

```
#define container_of(ptr, type, member) ({                    \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);  \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

[3]Tuple collisions are inserted in front – there is no reason to jump to the tail.

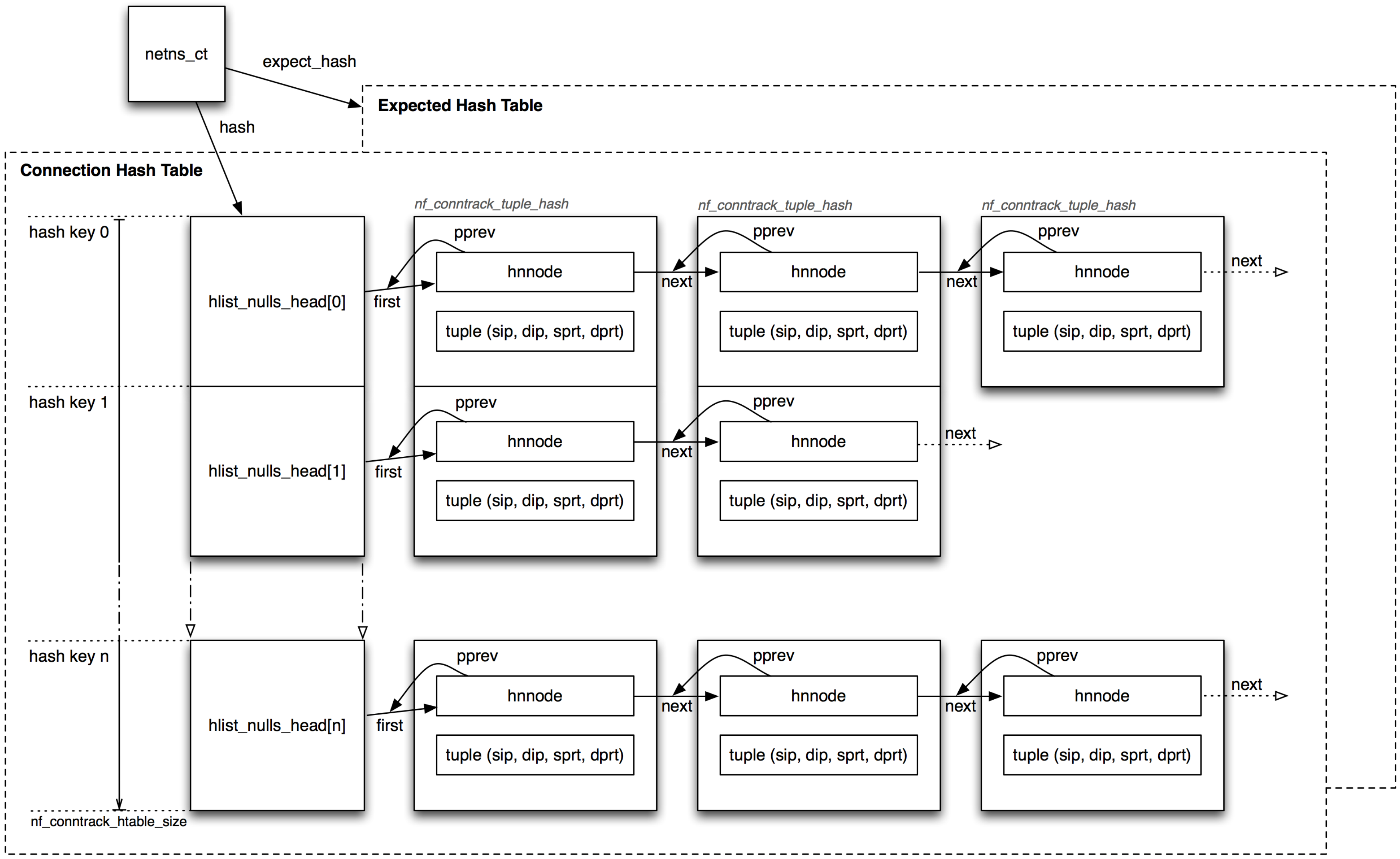

Figure 2: The structure of the hash tables that store active connections (`hash` pointer) and expected connections (`expect_hash`).

3.3 Execution

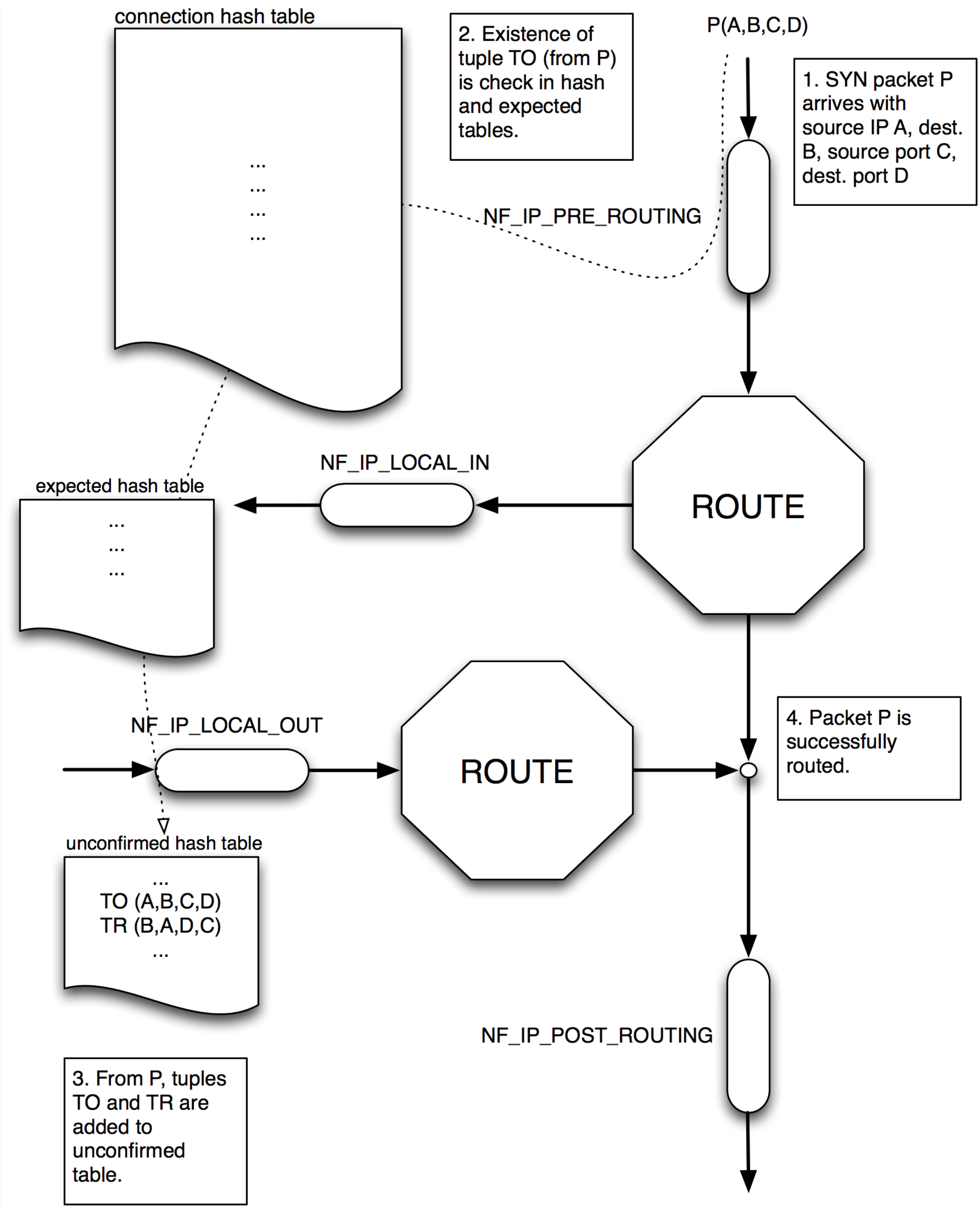

Figure 3: Part 1. New connection packet P enters the firewall.

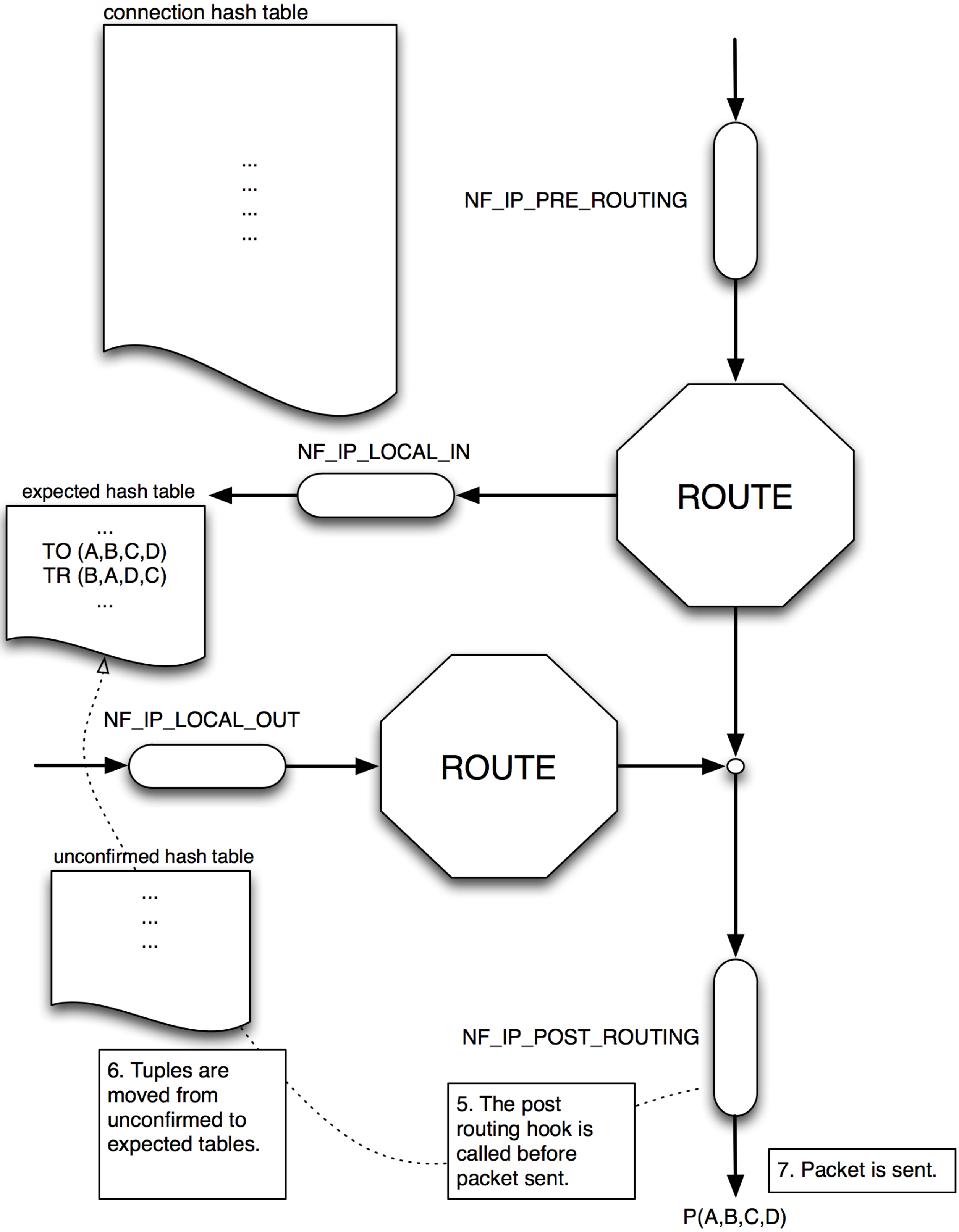

Figure 4: Part 2. Packet P is confirmed and leaves the firewall.

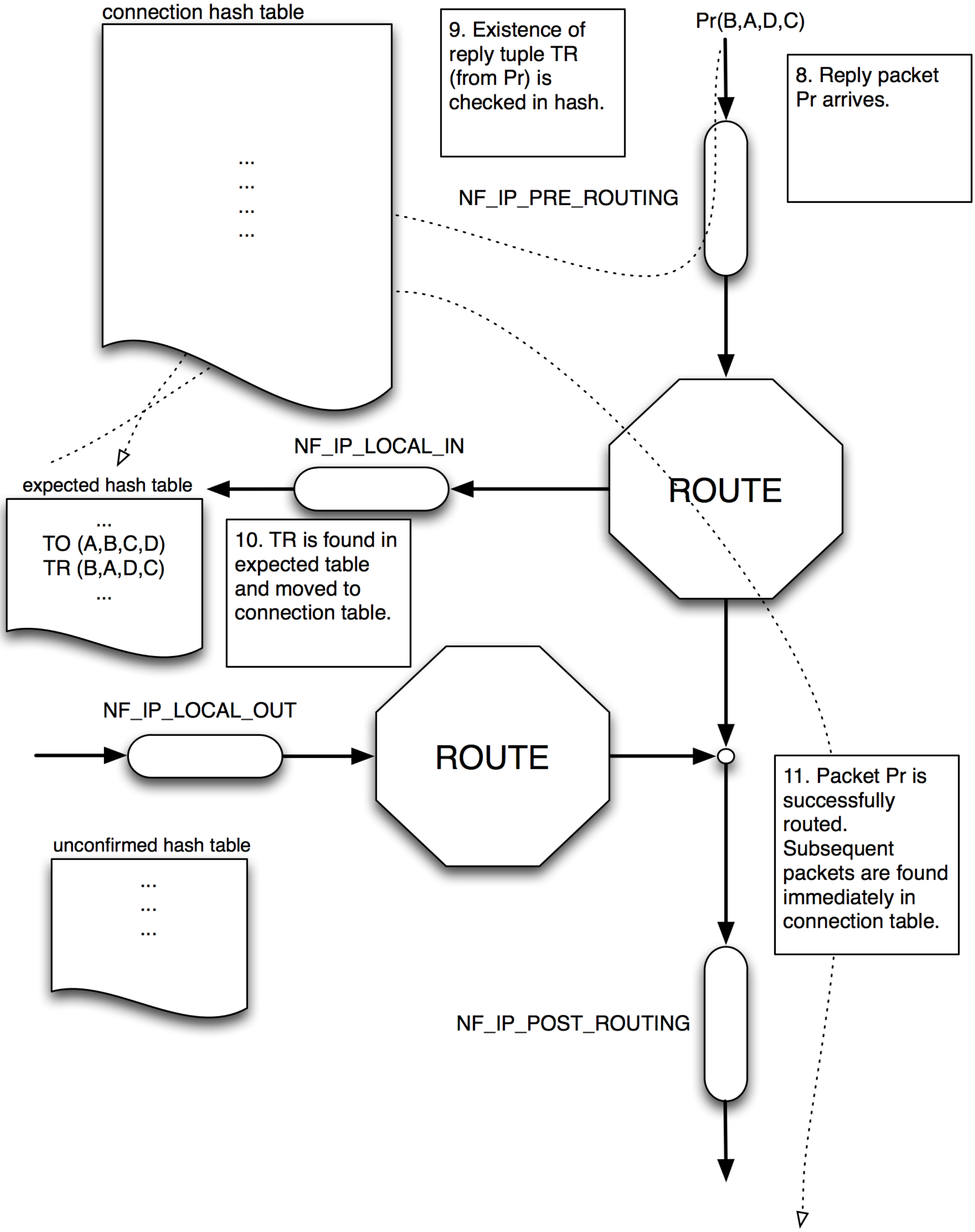

Figure 5: Part 3. Reply packet Pr returns and the connection is established.

3.3.1 Initiating a new connection

When a packet arrives on an interface or is sent by a local process, it triggers the PRE_ROUTING or LOCAL_OUT hooks respectively[fig. 7]. The conntrack module is always called first because it is registered with the highest priority. The packet's tuple is calculated[2] and looked-up in the hash [fig. 2] table for existing connections and expect_hash for expected connections (explained in 3.3.2). If the packet is the initiating packet of a new connection (e.g. a SYN packet for a TCP connection[3]) it will not be found. Instead, its tuple (in nf_conntrack_tuple_hash [3.2.2]) and reverse tuple(3.2.5) are added to the unconfirmed table in the netns_ns structure. The packet is then passed on for routing. However, there is some chance that the packet may not emerge, for instance, if there is no valid route or it is dropped by a subsequent rule[4]. In this scenario, the tuples will eventually time-out and be culled from unconfirmed. On the other hand, if it is routed successfully the POST_ROUTING hook is called. The initiating tuple and the reverse tuple is transferred from unconfirmed to hash_expect. At this point it can be assumed the packet will leave the host.

3.3.2 Reply to new connection

When the reply packet returns to the firewall, it again triggers the PRE_ROUTING hook. This time the reply tuple is found in expect_hash and then transferred to hash for established connections. Any subsequent packets related to this connection will found immediately in the hash table. The state of any given packet is made available to the rest of the kernel through the socket buffer struct sk_buff.nfct.

4 Registration

4.1 Hook Object

To be able to register a hook, the nf_register_hooktakes a struct of type nf_hook_ops, which is a structure hook to be added to the list of hooks and its options. From Victor Castro's description: "The first thing we see in the struct is the list_head struct, which is used to keep a linked list of hooks, but it's not necessary for our firewall. The nf_hookfn* struct member is the name of the hook function that we define. The pf integer member is used to identify the protocol family; it's PF_INET for IPv4. The next field is the hooknum int, and this is for the hook we want to use. The last field is the priority int. The priorities are specified in linux/netfilter_ipv4.h, but for our situation we want NF_IP_PRI_FIRST."[1] The list_head struct inside of the nf_hook_ops will be covered in more detail in section 4.3 and why it's needed.[1] [fig. 6]

```
struct nf_hook_ops
{
        struct list_head list;

        /* User fills in from here down. */
        nf_hookfn       *hook;
        struct module   *owner;
        void            *priv;
        u_int8_t        pf;
        unsigned int    hooknum;
        /* Hooks are ordered in ascending priority. */
        int             priority;
};
```

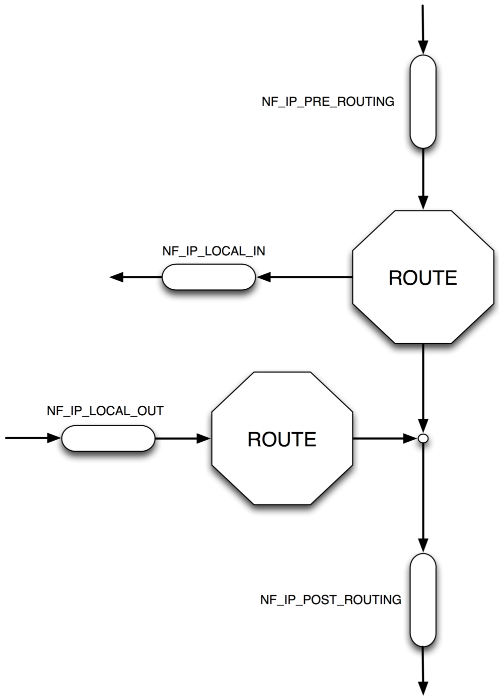

Figure 7: The packet is passed into `iptables`' hooks at certain points before and after routing.

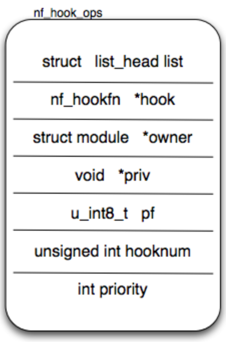

Figure 6: Hook Object

4.2 Hook List Object

The hook list object is a standard double linked list object used throughout the kernel. It's definition `list_head` and utility functions are defined in `include/linux/list.h` and allows for standard linked list functionality. A linked list is a perfect data structure in this case since there is no specific element indexing, but rather traversing through the list and calling all the available hooks [fig. 8]. There are 6 hooks lists [fig. 7]. One for each of `NF_INET_PRE_ROUTING` , `NF_INET_LOCAL_IN`, `NF_INET_FORWARD`, `NF_INET_LOCAL_OUT`, `NF_INET_POST_ROUTING`. By breaking the lists into one for each hook spot, on every spot where the hooks need to be called only the hooks responsible for that spot are iterated.[fig. 9]

```
extern struct list_head nf_hooks[NFPROTO_NUMPROTO][NF_MAX_HOOKS];
```

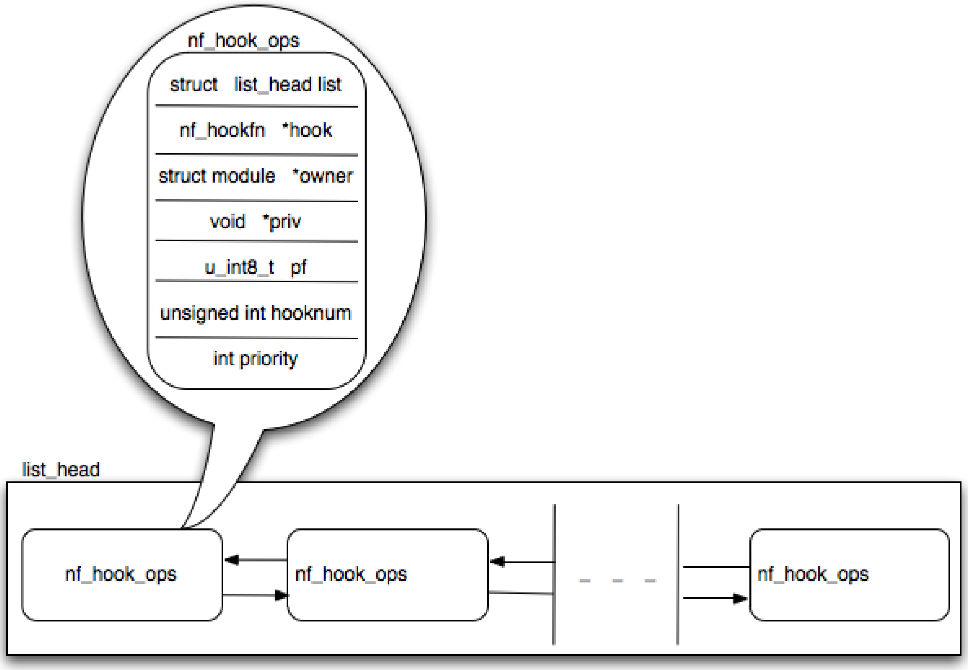

Figure 8: The list of hook objects

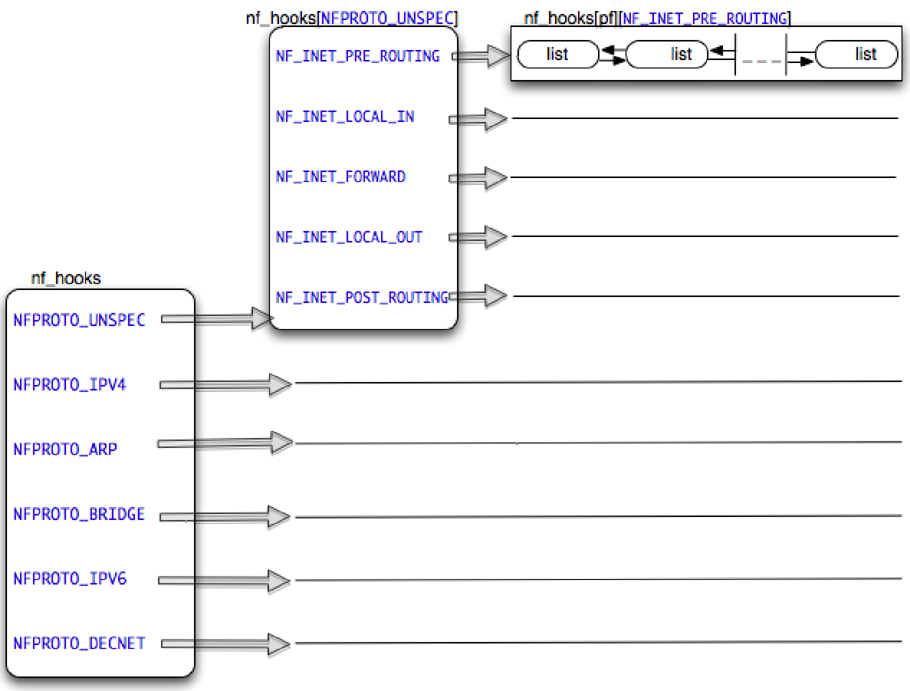

Figure 9: Two dimensional array of [protocol family] [hook type], and the corresponding hook lists

4.3 Hook Adding

The kernel attempts to add the hook object passed, by first creating a `nf_hook_mutex` object, locking it until it has finnished the hook adding process. It then adds the given hook object by using the `struct list_head list` as the reference pointer inside of the linked list but before adding it, loops through the list so that the list remains sorted by priority. That way when the hooks are called there is no need for sorting and the hooks with higher priority are called first. It unlocks the mutex and finally it adds the `nf_register_hook` function to the modules API. This allows the function to be used from loaded modules.[fig. 10]

```
int nf_register_hook(struct nf_hook_ops *reg)
{
        struct nf_hook_ops *elem;
        int err;

        err = mutex_lock_interruptible(&nf_hook_mutex);
        if (err < 0)
                return err;
        list_for_each_entry(elem, &nf_hooks[reg->pf][reg->hooknum], list) {
                if (reg->priority < elem->priority)
                        break;
        }
        list_add_rcu(&reg->list, elem->list.prev);
        mutex_unlock(&nf_hook_mutex);
        return 0;
}
EXPORT_SYMBOL(nf_register_hook);
```

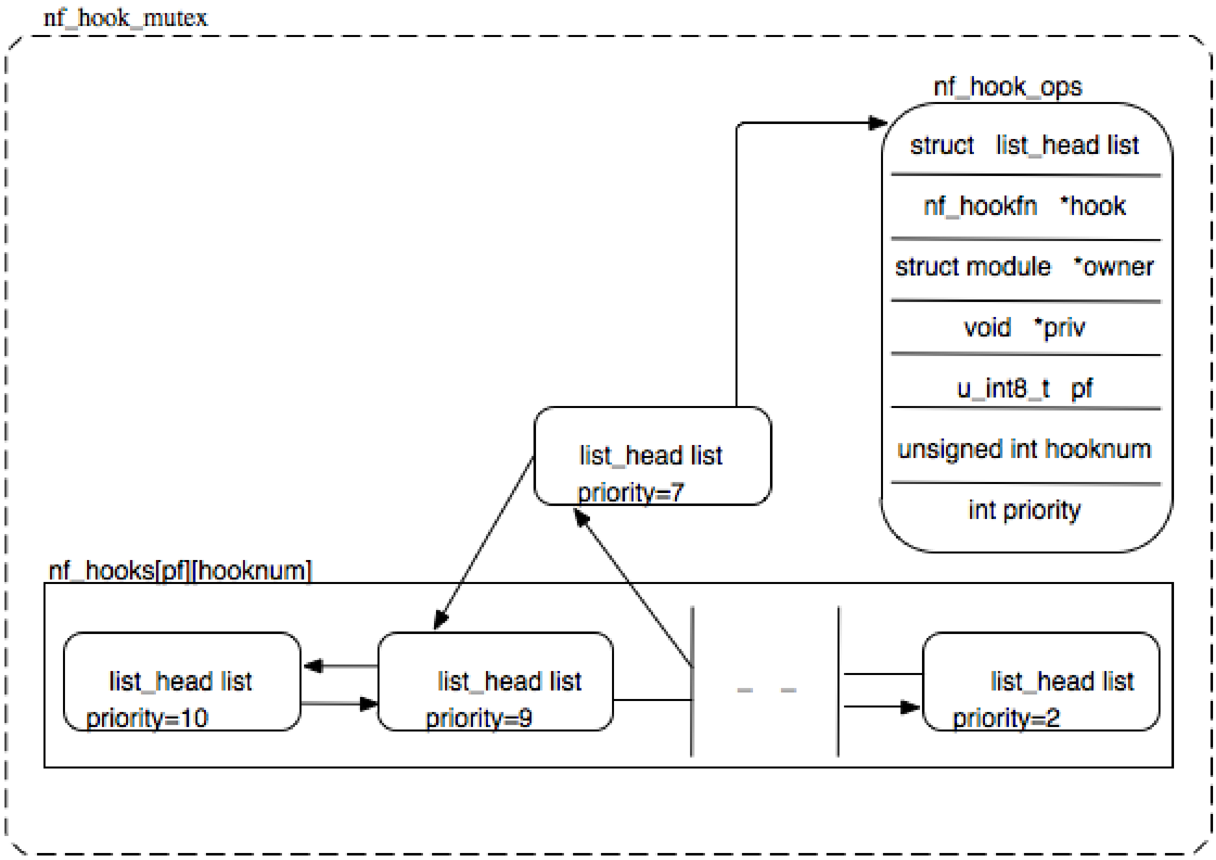

Figure 10: Adding a hook object

4.4 Hook List Adding

The functionality to add several hooks at once also is available by letting the user provide an array of `nf_hook_ops` and loop-add them to the linked list.

```
int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n)
{
        unsigned int i;
        int err = 0;

        for (i = 0; i < n; i++) {
                err = nf_register_hook(&reg[i]);
                if (err)
                        goto err;
        }
        return err;

err:
        if (i > 0)
                nf_unregister_hooks(reg, i);
        return err;
}
```

4.5 Hook And Hook-List Removing

The hooks can also be removed from the linked list in similar fashion to the way they are added.

```
void nf_unregister_hook(struct nf_hook_ops *reg);
void nf_unregister_hooks(struct nf_hook_ops *reg);
```

4.6 Hooks Processing

Finally there exist facilities inside `netfilter.h` that allow the kernel to process the hooks that are registered at the various steps. Many are in the form of macros that add certain small functionality before the function `nf_hook_slow` is called. This function iterates through all the hooks registered for the current step, executes them and proceeds according to the result returned by the hook.[fig. 11]

```
int nf_hook_slow(u_int8_t pf, unsigned int hook, struct sk_buff *skb,
                 struct net_device *indev, struct net_device *outdev,
                 int (*okfn)(struct sk_buff *), int thresh);
```

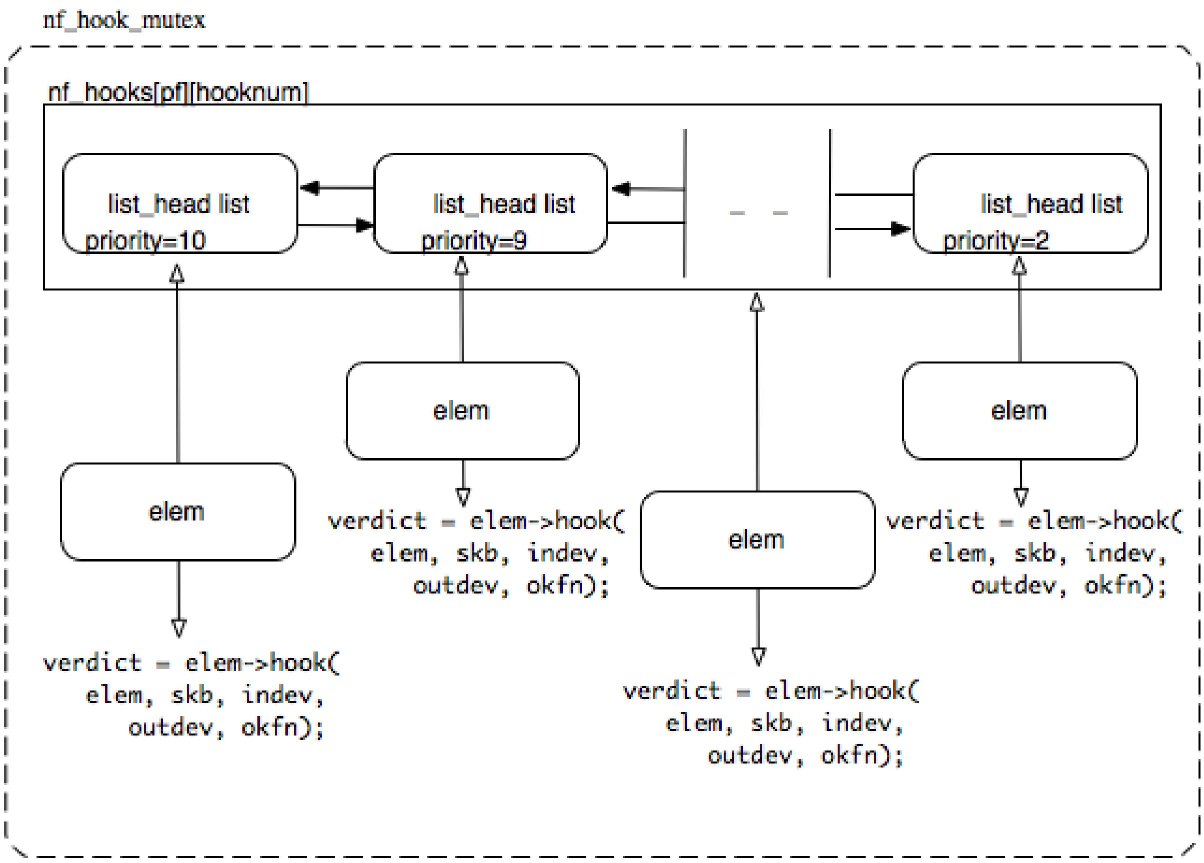

Figure 11: Hooks Processing

5 Conclusion

The `iptables` is a highly optimized and capable firewall. We were often surprised by the elegance of the code despite the need for efficiency. If time permitted, we would like to have tried implementing an `iptables` module, which seemed relatively straightforward. The most confusing aspect was dealing with the remnants of the code unification that was performed in 2006-2007, but largely undocumented. This left several artifacts in the code referring to the older IPT data structures that were often redefined to XT equivalents. For navigating the code, we found `global` for Emacs and the C/C++ module for Eclipse extremely helpful. They allowed for quick definition look-ups.

References

[1] Victor Castro. Roll your own firewall with netfilter. http://www.linuxjournal.com/article/7184, october 2003.

[2] Bob Jenkins. Hash functions and block ciphers. http://burtleburtle.net/bob/hash/, March 2009.

[3] W. Richard Stevens. *TCP/IP Illustrated*. Addison-Wesley, 1994.

[4] Harald Welte. Linux 2.4.x netfilter/iptables firewalling internals. Technical report, netfilter.org, 2002.